

# openUTM V6.1

Programming Applications with KDCS for COBOL, C and C++

## **Comments... Suggestions... Corrections...**

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:

[manuals@ts.fujitsu.com](mailto:manuals@ts.fujitsu.com)

## **Certified documentation according to DIN EN ISO 9001:2008**

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH

[www.cognitas.de](http://www.cognitas.de)

## **Copyright and Trademarks**

Copyright © Fujitsu Technology Solutions GmbH 2011.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

---

# Contents

<b>1</b>	<b>Preface . . . . .</b>	<b>11</b>
<b>1.1</b>	<b>Summary of contents and target group . . . . .</b>	<b>13</b>
<b>1.2</b>	<b>Summary of contents of the openUTM documentation . . . . .</b>	<b>14</b>
1.2.1	openUTM documentation . . . . .	14
1.2.2	Documentation for the openSEAS product environment . . . . .	18
1.2.3	README files . . . . .	19
<b>1.3</b>	<b>Innovations in openUTM V6.1 . . . . .</b>	<b>20</b>
1.3.1	New server functions . . . . .	20
1.3.1.1	New functions operative in all UTM applications . . . . .	20
1.3.1.2	New functions in UTM cluster applications . . . . .	21
1.3.2	New client functions . . . . .	24
1.3.3	New and modified functions for openUTM WinAdmin . . . . .	25
<b>1.4</b>	<b>Notational conventions . . . . .</b>	<b>27</b>
<b>2</b>	<b>Structure and use of UTM programs . . . . .</b>	<b>29</b>
<b>2.1</b>	<b>The openUTM service concept . . . . .</b>	<b>32</b>
<b>2.2</b>	<b>Structure of a program unit . . . . .</b>	<b>35</b>
2.2.1	Program framework . . . . .	35
2.2.2	Structure of a dialog program unit . . . . .	37
2.2.3	Reentrant capability of program units . . . . .	39
<b>2.3</b>	<b>Structuring services . . . . .</b>	<b>40</b>
2.3.1	Multi-step services . . . . .	40
2.3.2	Multiple program units in one processing step . . . . .	44
2.3.3	Multiple processing steps in a single program unit . . . . .	46
2.3.4	Subprogram calls from program units . . . . .	47
2.3.5	Chaining services . . . . .	47
2.3.6	Stacking services . . . . .	48

<b>2.4</b>	<b>Message Queuing (asynchronous processing)</b>	<b>50</b>
2.4.1	Messages to UTM-controlled queues	51
2.4.1.1	Output jobs	51
2.4.1.2	Background jobs	52
2.4.1.3	MQ calls of the KDCS interface	52
2.4.1.4	Structure of an asynchronous service	53
2.4.1.5	Redelivery with background jobs	60
2.4.1.6	Saving incorrectly processed messages in the dead letter queue	60
2.4.2	Messages to service-controlled queues	61
2.4.2.1	USER queues	61
2.4.2.2	TAC queues	62
2.4.2.3	Temporary queues	63
2.4.2.4	MQ calls of the KDCS interface	64
2.4.2.5	Lifetime of queues and queue messages	64
2.4.2.6	Deleting USER and TAC queues by means of programmed administration	65
2.4.2.7	Examples	66
<b>2.5</b>	<b>KDCS storage areas in openUTM</b>	<b>74</b>
2.5.1	Standard primary working area (SPAB)	78
2.5.2	Communication area (KB)	80
2.5.3	Local secondary storage area (LSSB)	82
2.5.4	Global secondary storage area (GSSB)	83
2.5.5	Terminal-specific long-term storage area (TLS)	84
2.5.6	User-specific long-term storage area (ULS)	85
2.5.7	User log file	86
2.5.8	Other areas	87
2.5.9	Action with locked storage areas (TLS, ULS and GSSB)	88
<b>2.6</b>	<b>Programming error routines</b>	<b>89</b>
<b>2.7</b>	<b>Message segments</b>	<b>90</b>
<b>2.8</b>	<b>Communication partners of a UTM application</b>	<b>92</b>
<b>2.9</b>	<b>Output to printers</b>	<b>94</b>
2.9.1	Hardcopy mode with openUTM	94
2.9.2	Print jobs	95
2.9.3	Output in line mode	97
2.9.4	Outputs in format mode	97
<b>2.10</b>	<b>Support for ID card readers</b>	<b>99</b>
2.10.1	Signing on to the application via ID card reader	99
2.10.2	Data input via ID card	100

---

<b>3</b>	<b>Interaction with databases</b>	<b>101</b>
<b>3.1</b>	<b>UTM transaction and DB transaction</b>	<b>103</b>
<b>3.2</b>	<b>Programming ESQL program units</b>	<b>105</b>
<b>3.3</b>	<b>Error processing with connected databases</b>	<b>106</b>
<b>4</b>	<b>Using formats</b>	<b>107</b>
<b>4.1</b>	<b>Format names for message exchange with UPIC clients</b>	<b>107</b>
<b>4.2</b>	<b>Use of formats in openUTM in BS2000/OSD</b>	<b>108</b>
4.2.1	Screen output functions in format mode	111
4.2.2	Starting services using basic formats	112
4.2.3	Using multiple partial formats	113
4.2.3.1	Output formatting with partial formats	113
4.2.3.2	Input formatting with partial formats	114
4.2.4	Message flow in openUTM (BS2000/OSD)	116
4.2.5	Controlling the output in line mode (BS2000/OSD)	117
<b>4.3</b>	<b>Screen restart</b>	<b>119</b>
<b>5</b>	<b>Program structure in distributed processing</b>	<b>121</b>
<b>5.1</b>	<b>Addressing remote services</b>	<b>122</b>
<b>5.2</b>	<b>Distributed dialogs</b>	<b>124</b>
5.2.1	Controlling communication in the program	124
5.2.2	Error handling by the program unit	125
5.2.2.1	Programmed reset	125
5.2.2.2	Error handling after service restart	128
5.2.3	Load distribution using LPAP bundles	131
<b>5.3</b>	<b>Distributed dialogs via LU6.1</b>	<b>132</b>
5.3.1	Programming aids	132
5.3.2	Programming rules and recommendations	134
5.3.3	Existing program units as LU6.1 job receivers	141
5.3.4	Example: distributed dialog via LU6.1	143
<b>5.4</b>	<b>Distributed dialogs via OSI TP</b>	<b>147</b>
5.4.1	Functional units	147
5.4.2	Programming aids	149
5.4.3	Programming rules for dialogs without the functional unit commit	154
5.4.4	Programming rules with the functional unit commit	154

## Contents

---

5.4.5	Programming rules for communications with BeanConnect . . . . .	157
5.4.6	Particularities of rollback and restart . . . . .	158
5.4.7	Using existing program units for OSI TP communication . . . . .	160
5.4.8	Particularities with heterogeneous coupling . . . . .	162
5.4.9	Examples: distributed dialogs via OSI TP . . . . .	164
5.4.9.1	One job receiver . . . . .	165
5.4.9.2	Multiple job receivers . . . . .	177
5.4.9.3	More complex dialog trees . . . . .	179
5.4.9.4	Using CTRL AB to terminate a job receiver . . . . .	188
<b>5.5</b>	<b>UTM-controlled queues in distributed processing . . . . .</b>	<b>190</b>
5.5.1	Job submitter side . . . . .	191
5.5.2	Job receiver side . . . . .	192
<b>5.6</b>	<b>Service-controlled queues in distributed processing . . . . .</b>	<b>193</b>
<b>6</b>	<b>Program structure in communication with transport system applications . . . . .</b>	<b>195</b>
<hr/>		
<b>6.1</b>	<b>Communication with TS applications of the type APPLI . . . . .</b>	<b>195</b>
<b>6.2</b>	<b>Communication via socket connections . . . . .</b>	<b>196</b>
6.2.1	Input messages for openUTM . . . . .	196
6.2.2	Output messages of openUTM . . . . .	197
6.2.3	Structure of the socket protocol header . . . . .	199
<b>7</b>	<b>KDCS calls . . . . .</b>	<b>203</b>
<hr/>		
	Complete overview of KDCS calls . . . . .	204
	Comments on the description of the KDCS calls . . . . .	207
	APRO Address job-receiving service . . . . .	208
	CTRL Control OSI TP Dialog . . . . .	219
	DADM Administer message queues . . . . .	224
	DGET Read a message from a service-controlled queue . . . . .	233
	DPUT Generate time-driven asynchronous messages . . . . .	244
	DPUT call without job complex . . . . .	245
	DPUT call in a job complex . . . . .	258
	FGET Receive asynchronous message . . . . .	266
	FPUT Generating asynchronous messages . . . . .	272
	GTDA Read from TLS . . . . .	282
	INFO Request information . . . . .	286
	INFO CK call . . . . .	298
	INIT Initialize program unit . . . . .	302
	LPUT Write to log file . . . . .	326

MCOM	Define job complex . . . . .	329
MGET	Receive dialog message . . . . .	334
MPUT	Send dialog message . . . . .	351
PADM	Administer printouts and printers . . . . .	364
PEND	Terminate program unit . . . . .	372
PGWT	Set wait point in program without terminating program unit . . . . .	385
PTDA	Write to TLS . . . . .	395
QCRE	Create temporary queue . . . . .	399
QREL	Delete temporary queue . . . . .	404
RSET	Reset transaction . . . . .	407
SGET	Read from secondary storage area . . . . .	411
SIGN	Control sign-on and sign-off, check authorization data, change passwords . . . . .	417
SIGN CL	- Change locale of user ID . . . . .	428
SPUT	Write to secondary storage area . . . . .	432
SREL	Delete secondary storage area . . . . .	438
UNLK	Unlock TLS, ULS or GSSB . . . . .	443
<b>8</b>	<b>Event functions . . . . .</b>	<b>447</b>
<b>8.1</b>	<b>Event exits . . . . .</b>	<b>449</b>
8.1.1	Event exit INPUT . . . . .	449
8.1.2	Event exit START . . . . .	457
8.1.3	Event exit SHUT . . . . .	458
8.1.4	Event exit VORGANG . . . . .	459
8.1.5	Event exit FORMAT (BS2000/OSD) . . . . .	461
<b>8.2</b>	<b>STXIT routines (BS2000/OSD) . . . . .</b>	<b>469</b>
<b>8.3</b>	<b>Event handling in ILCS programs (BS2000/OSD) . . . . .</b>	<b>470</b>
<b>8.4</b>	<b>Event services . . . . .</b>	<b>472</b>
8.4.1	Dialog service BADTACS . . . . .	472
8.4.2	Asynchronous service MSGTAC . . . . .	473
8.4.3	The SIGNON service . . . . .	476
8.4.3.1	Programming notes . . . . .	477
8.4.3.2	Sign-on service for terminals . . . . .	478
8.4.3.3	Sign-on service for UPIC clients or transport system clients . . . . .	481

<b>9</b>	<b>Additional information for C/C++</b>	<b>485</b>
<b>9.1</b>	<b>Program structure for C/C++ program units</b>	<b>485</b>
9.1.1	C/C++ program units as subroutines	485
9.1.2	Parameters of a C/C++ program unit	487
9.1.3	Declaring data	488
9.1.3.1	Communication area	488
9.1.3.2	Standard primary working area	488
9.1.3.3	Other data areas (AREAs)	489
9.1.4	Data structures for C/C++ program units	494
9.1.5	Command section of a C/C++ program unit	496
9.1.6	C/C++ macro interface	497
9.1.7	Event exits	502
9.1.8	Programming the KDCS error handling routines	503
9.1.9	Modifying KDCS attributes	504
9.1.10	Platform-specific characteristics for BS2000/OSD	505
9.1.11	Platform-specific characteristics in Unix systems	507
9.1.12	Platform-specific characteristics in Windows systems	508
<b>9.2</b>	<b>Programming examples in C/C++</b>	<b>510</b>
9.2.1	Examples of individual KDCS calls	510
9.2.2	Example of a complete C program unit	515
9.2.3	Example: INPUT exit	516
9.2.4	Example: MSGTAC event service	519
9.2.5	Example of a complete UTM application	524
<b>10</b>	<b>Additional information for COBOL</b>	<b>543</b>
<b>10.1</b>	<b>Structure of COBOL program units</b>	<b>543</b>
10.1.1	COBOL program units as subroutines	543
10.1.2	Data structures for COBOL program units	549
10.1.3	KDCS calls in COBOL program units	552
10.1.4	Platform-specific features in BS2000/OSD	554
10.1.5	Platform-specific features in Unix systems	558
10.1.6	Platform-specific features in Windows systems	560
<b>10.2</b>	<b>Programming examples in COBOL</b>	<b>563</b>
10.2.1	Examples of individual KDCS calls	563
10.2.2	Example of an INPUT exit	570
10.2.3	Example of an asynchronous MSGTAC program unit	572
10.2.4	Example of a complete UTM application	577



---

<b>11</b>	<b>Appendix</b> . . . . .	<b>593</b>
<b>11.1</b>	<b>Overview of all KDCS calls</b> . . . . .	<b>593</b>
<b>11.2</b>	<b>Different field names for C/C++ and COBOL</b> . . . . .	<b>603</b>
<b>11.3</b>	<b>ASCII-EBCDIC code conversion</b> . . . . .	<b>608</b>
11.3.1	BS2000/OSD . . . . .	608
11.3.2	Unix systems and Windows systems . . . . .	608
11.3.2.1	Modifying the code table in Unix systems . . . . .	609
11.3.2.2	Modifying the code table in Windows systems . . . . .	609
	 <b>Glossary</b> . . . . .	 <b>611</b>
	 <b>Abbreviations</b> . . . . .	 <b>645</b>
	 <b>Related publications</b> . . . . .	 <b>651</b>
	 <b>Index</b> . . . . .	 <b>661</b>

---



---

# 1 Preface

Modern enterprise-wide IT environments are subjected to many challenges of an increasingly explosive nature. This is the result of:

- heterogeneous system landscapes
- different hardware platforms
- different networks and different types of network access (TCP/IP, SNA, HTTP)
- the applications used by companies

Consequently, problems arise – whether as a result of mergers, joint ventures or labor-saving measures. Companies are demanding flexible, scalable applications, as well as transaction processing capability for processes and data, while business processes are becoming more and more complex. The growth of globalization means, of course, that applications are expected to run 24 hours a day, seven days a week, and must offer high availability in order to enable Internet access to existing applications across time zones.

openUTM is a transaction-oriented middleware platform that offers a runtime environment that meets all these requirements of modern, business-critical applications, because openUTM combines all the standards and advantages of transaction monitor middleware platforms and message queuing systems:

- consistency of data and processing
- high availability of the applications (not just the hardware)
- high throughput even when there are large numbers of users (i.e. highly scalable)
- flexibility as regards changes to and adaptation of the IT system

An UTM application can be run as a standalone UTM application or on several different computers as a UTM cluster application.

openUTM forms part of the comprehensive **openSEAS** offering. In conjunction with the Oracle Fusion middleware, openSEAS delivers all the functions required for application innovation and modern application development. Innovative products use the sophisticated technology of openUTM in the context of the **openSEAS** product offering:

- BeanConnect is an adapter that conforms to Oracle/Sun's Java Connector Architecture (JCA) and supports standardized connection of UTM applications to J2EE application servers. This makes it possible to integrate tried-and-tested legacy applications in new business processes.
- The WebTransactions member of the openSEAS family is a product that allows tried-and-tested host applications to be used flexibly in new business processes and modern application scenarios. Existing UTM applications can be migrated to the Web without modification.

## 1.1 Summary of contents and target group

The openUTM manual "Programming Applications with KDCS" is intended for anyone who wants to use the KDCS programming interface to program UTM applications.

The vast majority of the information presented in this manual is operating system-independent, i.e. it applies to both Unix systems and Windows systems, as well as to BS2000/OSD. In addition, descriptions of platform-specific features are also provided. Any such platform-specific information is clearly indicated by means of markings in the margins. As a result you can, if you so desire, write applications that can run under Unix systems, Windows systems and BS2000/OSD. The form of presentation used throughout this manual will also simplify matters if you are wanting to port existing UTM (BS2000) applications to Unix systems or Windows systems, or vice versa.

The chapters in this manual can be subdivided into three blocks:

- Chapter 2 - 6 contain introductory information. They explain basic concepts, such as the way program units are structured into processing step modules, message queuing functionality and distributed processing in their respective contexts.
- Chapters 7 and 8 provide reference materials in which you can look things up. Chapter 7 lists the KDCS calls in alphabetical order, and chapter 8 lists the event functions.
- Chapters 9 and 10 contain programming language-specific information - chapter 9 for C/C++, and chapter 10 for COBOL.

The appendix includes a series of tables which provide overviews of the entries that must be made in the parameter area for the individual KDCS calls, for example, or of the values returned to the communication area.

The detailed reference section at the back of the manual – including a glossary, abbreviations, a list of related publications and a keyword index – is intended to help you get the most out of this manual.



Wherever the term Unix system or Unix platform is used in the following, then this should be understood to mean both a Unix-based operating system such as Solaris or HP-UX and a Linux distribution such as SUSE or Red Hat.

Wherever the term Windows system or Windows platform is used below, this should be understood to mean all the variants of Windows under which openUTM runs.

## 1.2 Summary of contents of the openUTM documentation

This section provides an overview of the manuals in the openUTM suite and of the various related products.

### 1.2.1 openUTM documentation

The openUTM documentation consists of manuals, an online help system for openUTM WinAdmin, which is the graphical administration workstation, and a release note for each platform on which openUTM is released.

Some manuals are valid for all platforms, and others apply specifically to BS2000/OSD, Unix systems or Windows systems.

All the manuals are available as PDF files on the internet at

<http://manuals.ts.fujitsu.com>

On this site, enter the search term “openUTM V6.1” in the **Search by product** field to display all openUTM manuals of version 6.1.

The manuals are included on the Enterprise DVD with open platforms and are also available on the WinAdmin DVD (for BS2000/OSD).

The following sections provide a task-oriented overview of the openUTM V6.1 documentation. You will find a complete list of documentation for openUTM in the chapter on related publications at the back of the manual on [page 651](#).

#### Introduction and overview

The **Concepts and Functions** manual gives a coherent overview of the essential functions, features and areas of application of openUTM. It contains all the information required to plan a UTM operation and to design an UTM application. The manual explains what openUTM is, how it is used, and how it is integrated in the BS2000/OSD, Unix based and Windows based platforms.

## Programming

- You will require the **Programming Applications with KDCS for COBOL, C and C++** manual to create server applications via the KDCS interface. This manual describes the KDCS interface as used for COBOL, C and C++. This interface provides the basic functions of the universal transaction monitor, as well as the calls for distributed processing. The manual also describes interaction with databases.
- You will require the **Creating Applications with X/Open Interfaces** manual if you want to use the X/Open interface. This manual contains descriptions of the UTM-specific extensions to the X/Open program interfaces TX, CPI-C and XATMI as well as notes on configuring and operating UTM applications which use X/Open interfaces. In addition, you will require the X/Open-CAE specification for the corresponding X/Open interface.
- If you want to interchange data on the basis of XML, you will need the document entitled **openUTM XML for openUTM**. This describes the C and COBOL calls required to work with XML documents.
- For BS2000/OSD there is supplementary documentation on the programming languages Assembler, Fortran, Pascal-XT and PL/1. T

## Configuration

The **Generating Applications** manual is available to you for defining configurations. This describes for both standalone UTM applications and UTM cluster applications how to use the UTM tool KDCDEF to

- define the configuration
- generate the KDCFILE
- and generate the UTM cluster files for UTM cluster applications

In addition, it also shows you how to transfer important administration and user data to a new KDCFILE using the KDCUPD tool. You do this, for example, when moving to a new openUTM version or after changes have been made to the configuration. In the case of UTM cluster applications, it also indicates how you can use the KDCUPD tool to transfer this data to the new UTM cluster files.

## Linking, starting and using UTM applications

In order to be able to use UTM applications, you will need the **Using openUTM Applications** manual for the relevant operating system (BS2000/OSD or Unix systems/Windows systems). This describes how to link and start a UTM application program, how to sign on and off to and from a UTM application and how to replace application programs dynamically and in a structured manner. It also contains the UTM commands that are available to the terminal user. Additionally, those issues are described in detail that need to be considered when operating UTM cluster applications.

## Administering applications and changing configurations dynamically

- The **Administering Applications** manual describes the program interface for administration and the UTM administration commands. It provides information on how to create your own administration programs for operating a standalone UTM application or a UTM cluster application and on the facilities for administering several different applications centrally. It also describes how to administer message queues and printers using the KDCS calls DADM and PADM.
- If you are using **openUTM WinAdmin**, the graphical administration workstation, the following documentation is available to you:
  - A **description of WinAdmin**, which provides a comprehensive overview of the functional scope and handling of WinAdmin. This document is shipped with the software and is also available online as a PDF file.
  - The **online help system**, which provides context-sensitive help information on all dialog boxes and associated parameters offered by the graphical user interface. In addition, it also tells you how to configure WinAdmin in order to administer standalone UTM applications and UTM cluster applications.

## Testing and diagnosing errors

You will also require the **Messages, Debugging and Diagnostics** manuals (there are separate manuals for Unix systems / Windows systems and for BS2000/OSD) to carry out the tasks mentioned above. These manuals describe how to debug a UTM application, the contents and evaluation of a UTM dump, the behavior in the event of an error, and the openUTM message system, and also lists all messages and return codes output by openUTM.



## Creating openUTM clients

The following manuals are available to you if you want to create client applications for communication with UTM applications:

- The **openUTM-Client for the UPIC Carrier System** describes the creation and operation of client applications based on UPIC. In addition to the description of the CPI-C and XATMI interfaces, you will find information on how you can use the C++ classes to create programs quickly and easily.
- The **openUTM-Client for the OpenCPIC Carrier System** manual describes how to install and configure OpenCPIC and configure an OpenCPIC application. It describes how to install OpenCPIC and how to configure an OpenCPIC application. It indicates what needs to be taken into account when programming a CPI-C application and what restrictions apply compared with the X/Open CPI-C interface.
- The documentation for the **JUpic-Java classes** shipped with BeanConnect is supplied with the software. This documentation consists of Word and PDF files that describe its introduction and installation and of Java documentation with a description of the Java classes.
- The **BizXML2Cobol** manual describes how you can extend existing COBOL programs of a UTM application in such a way that they can be used as an XML-based standard Web service. How to work with the graphical user interface is described in the **online Help system**.
- If you want to provide UTM services on the Web quickly and easily then you need the manual **WebServices for openUTM**. The manual describes how to use the software product WS4UTM (WebServices for openUTM) to make the services of UTM applications available as Web services. The use of the graphical user interface is described in the corresponding **online help system**.

## Communicating with the IBM world

If you want to communicate with IBM transaction systems, then you will also require the manual **Distributed Transaction Processing between openUTM and CICS, IMS and LU6.2 Applications**. This describes the CICS commands, IMS macros and UTM calls that are required to link UTM applications to CICS and IMS applications. The link capabilities are described using detailed configuration and generation examples. The manual also describes communication via openUTM-LU62 as well as its installation, generation and administration.

## 1.2.2 Documentation for the openSEAS product environment

This manual briefly describes how openUTM is connected to the openSEAS product environment in chapter 2. The following sections indicate which openSEAS documentation is relevant to openUTM.

### Integrating J2EE application servers and UTM applications

The BeanConnect adapter forms part of the openSEAS product suite. The BeanConnect adapter implements the connection between conventional transaction monitors and J2EE application servers and thus permits the efficient integration of legacy applications in Java applications.

- The manual **BeanConnect** describes the product BeanConnect, that provides a JCA 1.5-compliant adapter which connects UTM applications with applications based on J2EE, e.g. the Oracle application server.  
The manuals for the Oracle application server can be obtained from Oracle.

### Connecting to the web and application integration

You require the WebTransactions manuals to connect new and existing UTM applications to the Web using the product **WebTransactions**.

The manuals will also be supplemented by JavaDocs.

### 1.2.3 README files

Information on any functional changes and additions to the current product version described in this manual can be found in the product-specific README files.

#### *Readme files online*

Readme files are available to you online in addition to the product manuals under the various products at <http://manuals.ts.fujitsu.com>.

#### *Readme files under BS2000/OSD*

On your BS2000 system you will find Readme files for the installed products under the file name:

```
SYSRME.<product>.<version>.E
```

Please refer to your system administrator for the user ID under which the required Readme file can be found. You can also obtain the path name of the Readme file directly by entering the following command:

```
/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>, LOGICAL-ID=SYSRME.E
```

You can view the Readme file on screen with `/SHOW-FILE` or by opening it in an editor, or print it on a standard printer using the following command:

```
/PRINT-DOCUMENT <filename>, LINE-SPACING=*BY-EBCDIC-CONTROL
```

#### *Readme files under Unix systems:*

The README file and any other files, such as a manual supplement file, can be found in the *utmpath* under `/docs/language`.

#### *Readme files under Windows systems:*

The README file and any other files, such as a manual supplement file, can be found in the *utmpath* under `\Docs\language`.

## 1.3 Innovations in openUTM V6.1

The functionality of UTM cluster applications has been greatly extended. In addition, openUTM offers a number of new server-side and client-side functions. The graphical administration workstation WinAdmin has been converted to Java technology and has also been extended to include new functions.

The following sections provide more detail on the innovations in the individual areas.

### 1.3.1 New server functions

Some of the enhancements work in all UTM applications whereas others can only be used in UTM cluster applications.

#### 1.3.1.1 New functions operative in all UTM applications

The functions listed below work in both standalone applications and in UTM cluster applications.

##### Generation

- New default value for MAX REQNR in BS2000. For performance reasons, this has been increased from 2 to 20.

##### KDCADMI enhancements

The following new and modified object types, operation codes and data structures are now available for administration tasks:

- New object type KC\_PTC for the operation code KC\_GET\_OBJECT with the data structure *kc\_ptc\_str* for the display of distributed transactions with the state PTC (Prepare to Commit).
- New operation code KC\_PTC\_TA with sub-opcode KC\_ROLLBACK for the rollback of transactions with the state PTC.
- New object type KC\_DB\_INFO for operation code KC\_GET\_OBJECT with new data structure *kc\_db\_info\_str* for the output of information on the database connection.
- New sub-opcode KC\_SAME in the operation code KC\_CHANGE\_APPLICATION to make it possible to reload the application program on open platforms without it being necessary to use a new version of the program when FGs are employed.

- When the operation code `KC_GET_OBJECT` is used with the object type `KC_USER` or `KC_USER_DYN2`, the system also indicates whether the user has an open service with a transaction in the state `PTC`.
- Data structure `kc_tac_str`  
In the case of an `XA` connection, the fields `db_counter` and `db_elap_msec` no longer return the value 0 but instead binary zero since the database values cannot be captured in the case of an `XA` connection.

### Extensions to commands

- The command `KDCAPPL PROG=SAME` can be used to reload the application program on open platforms without it being necessary to use a new version of the program when `FGGs` are employed (see also the `KDCADMI` program interface).
- The output from `KDCINF SYSPARM` now contains additional information.

### 1.3.1.2 New functions in UTM cluster applications

#### Functional enhancements for UTM cluster applications

- The global storage areas `GSSB` and `ULS` can be used globally in the cluster  
`GSSB` and `ULS` are administered in such a way that they are available for all users in all node applications. I.e. all node applications have the same view of the data in the storage area. If a storage area is set up (`GSSB`) or modified (`GSSB`, `ULS`) in a node application then this is also visible to all the other node applications in this `UTM` cluster application after the transaction has ended.  
To permit the use of `GSSB` and `ULS` globally in the cluster there is a special lock management function for which an optional deadlock detection mechanism has also been implemented.
- The dialog services of users generated with `RESTART=YES` are valid globally in the cluster.  
This means that users who were generated with `RESTART=YES` can usually continue open transactions at another node application.
- `KDCUPD`  
The `KDCUPD` tool now has two modes for `UTM` cluster applications:
  - A mode in which the user data that is local to the node is transferred for each node application
  - A mode in which the user data that is valid globally in the cluster (`ULS`, `GSSB`, service data, passwords, locales) is transferred

- Cluster-internal communication

Cluster-internal communication has been modified in order to improve performance.

### **Additional UTM cluster files**

Additional files are required for the new functionality available in UTM cluster applications:

- The cluster GSSB file and cluster ULS file are created at generation time in order to permit the global administration of the GSSB and ULS areas throughout the cluster.
- To permit lock management globally throughout the cluster, there is one separate lock management file. This is created as soon as the UTM cluster application is started with the newly generated UTM cluster files.
- A cluster page pool for the storage of the user data applicable throughout the cluster is created at generation time. This user data includes, for example, the contents of GSSB and ULS and the data of a user's dialog service that has been generated with `RESTART=YES`.

Note: There is still also a local page pool for each node application. This is used to store the user data that is local to the node.

### **Generation**

The following statements have been modified in order to make it possible to generate UTM cluster applications:

- **CLUSTER**
  - New operands `ABORT-BOUND-SERVICE`, `DEADLOCK-PREVENTION`, `PGPOOL` and `PGPOOLFS` which make it possible to control service restarts, the locking behavior for global memory areas and the properties of the cluster page pool.
  - The permitted range of values for the `FILE-LOCK-RETRY` and `FILE-LOCK-TIMER-SEC` operands has been changed.
  - The operand `LISTENER-ID` is also now available for Unix systems and Windows systems.
  - The `USER-RESTART` and `GLOBAL-UTM-DATA` operands are no longer supported.
  - The `LISTENER-PORT` operand is no longer supported under BS2000.
- **CLUSTER-NODE**
  - The length of the `HOSTNAME` operand is now limited to a maximum of 8 characters.
  - `catid_B` is no longer supported in BS2000/OSD.

- MAX  
SINGLE is the only permitted specification for the KDCFILE operand.
- LTERM and USER  
The default value is RESTART=YES exactly as in the case of standalone applications.
- OPTION  
If GEN=CLUSTER is set then the UTM cluster files that are set up on generation are always regenerated. They must not exist prior to this.

### Enhancements to KDCADMI for UTM cluster applications

The following new and modified object types, operation codes and data structures are now available for the administration of UTM cluster applications:

- New object type KC\_CLUSTER\_CURR\_PAR for the operation codes KC\_GET\_OBJECT and KC\_MODIFY\_OBJECT with data structure *kc\_cluster\_curr\_par\_str* in order to display the current values of UTM cluster applications, for example the cluster page pool allocation.
- The operation code KC\_LOCK\_MGMT now has the new sub-opcodes KC\_ABORT\_BOUND\_SERVICE, KC\_ABORT\_ALL\_BOUND\_SERVICES and KC\_ABORT\_PTC\_SERVICE.  
  
These sub-opcodes are used to mark one or more (user) services bound to an abnormally terminated node application for abnormal termination. I.e. the services are terminated abnormally immediately on the start-up of the node application to which they are bound.  
  
This enables users to sign on at another node application.
- Using the KC\_GET\_OBJECT with the object type KC\_USER or KC\_USER\_DYN2 now also shows whether the user
  - has a node-bound service
  - or a node-bound service with a transaction in the state PTC.
- New sub-opcode KC\_READ\_NO\_GSSBFILE to specify whether or not the cluster GSSB file should be accessed when reading (further) GSSB objects with KC\_GET\_OBJECT. Using this sub-opcode for follow-up calls at KDCADMI considerably improves performance when a large number of GSSBs are present.
- Data structure *kc\_curr\_par\_str* (current values of the user parameters)  
*kc\_curr\_par\_str* has been extended in order to display statistics fields relating to lock conflicts.

- Data structure *kc\_cluster\_par\_str* (properties of a UTM cluster application)
  - *kc\_cluster\_par\_str* has been extended to make it possible to display the new settings of UTM cluster applications (e.g. sign-on behavior in the case of open, node-bound services, cluster page pool) and control the locking behavior for global storage areas.
  - The structure no longer contains the fields *global\_utm\_data* and *user\_restart*.
- Data structure *kc\_cluster\_node\_str*

The *hostname* field has been shortened to a length of 64 characters.
- Data structure *kc\_lock\_mgmt\_str*

The name *kc\_lock\_mgmt\_str* given to the structure used to release locks in V6.0 has been changed to *kc\_lock\_mgmt\_str* in V6.1.

## Messages

The messages K168, K172, K173, K177 and K187 present in UTM V6.0 have been replaced by the new message K190.

## 1.3.2 New client functions

Some of the changes apply to all clients, i.e. they apply both to the clients of standalone UTM applications and those of UTM cluster applications. Other changes apply only to the clients of UTM cluster applications.

### New and modified functions for all UPIC clients

- WARN or GRACE shutdown in a UTM application

UPIC client applications can react specifically to WARN or GRACE shutdowns. If a connection has been established to a UTM application and a WARN or GRACE shutdown is then triggered in the application, this is passed on to the client via the UPIC protocol. The client program can query the shutdown status and shutdown time using the new functions *Extract\_Shutdown\_State* and *Extract\_Shutdown\_Time*.
- Elimination of the OCX interface

The OCX interface is no longer provided in UPIC. Consequently, the section "ActiveX Control UpicB.ocx" is no longer included.
- Linux x86 64-bit platform

UPIC client applications can run in either a 32-bit or 64-bit environment on Linux x86 systems.



### **New functions for UPIC clients of UTM cluster applications**

This functionality is only available if upicfile cluster entries (prefix CD) are used.

- **WARN or GRACE shutdown in a UTM cluster application**

If, after connecting to a node application, the UPIC client detects that the node application is already in a WARN or GRACE shutdown state, then it disconnects again and establishes a connection to another node application.
- **Changing node application while a service is open**

If a user sign-on is rejected at a node application because a service that is bound to another node application is open for this user, then UPIC attempts to sign on this user at this (other) node application.

### **1.3.3 New and modified functions for openUTM WinAdmin**

Not only has WinAdmin been converted to Java technology, it also possesses a range of new and modified functions. Some of these functions are available for all applications (standalone UTM applications and UTM cluster applications) whereas others apply only to UTM cluster applications.

#### **Changeover to Java technology**

WinAdmin has been converted to Java technology:

- **Menu bar**

The menu bar has been modified and simplified. The *Application*, *Cluster*, *Edit*, *Position*, *Queue* and *System* menus are no longer present. The functionality of these menus has been included in the context menus.
- **List windows and text windows**

The display has been simplified. At the same time, the functionality has been greatly extended. The toolbar present in earlier versions is no longer present.
- **Tooltips in object lists and property dialogs**

Tooltips are displayed in UTM object lists and the property dialogs of UTM objects. These provide more detailed information on the cell contents and object properties. This is particularly beneficial in the case of UTM cluster applications since it means that the values of the individual node applications are available in lists and dialogs that apply globally throughout the cluster.
- **Definition of partial object lists**
- **The online Help system supplied with WinAdmin runs as a JavaHelp system.**

- Performance

Performance has been considerably improved through the use of parallel UPIC connections and asynchronous communications. Among other things, WinAdmin therefore now reacts faster to input entered in parallel to communications activities.

### **New functions operative in all UTM applications**

- Display of all open services in a running UTM application that have a transaction with the state PTC.
- Display of the generated database systems
- Function for rolling back transactions with the state PTC
- Display of the applications that are present in a collection

### **New functions operative in UTM cluster applications**

- Display of the new parameters for a UTM cluster application
- Display of the current values for a UTM cluster application
- New functions at the node application in order to mark all services bound to an abnormally terminated node application for abnormal termination so that the users can sign on at another node application.
- New functions for the **User** object in order to mark the services associated with users for abnormal termination:
  - Services bound to a node application that has terminated abnormally
  - Services with a transaction with the state PTC that are bound to a node application that has terminated abnormally

This allows the user to sign on at another node application.

- Activation/deactivation of deadlock detection in UTM cluster applications

### **Elimination of the Generate function**

The Generate function is no longer available in WinAdmin. As a result, the entire associated **Generate** subtree is no longer displayed and all the related commands are no longer present in the context menus.

## 1.4 Notational conventions

- B** This symbol is used in the left-hand margin to indicate BS2000/OSD-specific elements of a description.
- B**
- X** This symbol is used in the left-hand margin to indicate Unix system specific elements of a description.
- X**
- W** This symbol is used in the left-hand margin to indicate Windows specific elements of a description.
- W**
- B/X** This symbol is used in the left-hand margin to indicate parts of the description that are only relevant for openUTM in BS2000/OSD and Unix systems.
- B/X**
- B/W** This symbol is used in the left-hand margin to indicate parts of the description that are only relevant for openUTM in BS2000/OSD and Windows systems.
- B/W**
- X/W** This symbol is used in the left-hand margin to indicate parts of the description that are only relevant for openUTM in Unix systems and Windows systems.
- X/W**



Indicates references to comprehensive, detailed information on the relevant topic.



Indicates notes that are of particular importance.



Indicates warnings.



---

## 2 Structure and use of UTM programs

In this chapter you will be presented with an initial overview of programming UTM applications. You will learn what services and program units are and how you can apply the openUTM concepts to implement the business logic of your applications.

A UTM application provides your users with certain services: it processes service requests (jobs) which are sent by terminal users, client programs or other applications.

A **service** serves to process a job from a UTM application. It consists of one or more transactions and one or more program runs. openUTM differentiates between dialog services and asynchronous services. An openUTM service generally corresponds to a business transaction of the application logic.

When you design an application you program the business transactions of the application logic in the form of **program units**, also called **service routines**. The program units run as subprograms and are under the control of the main routine of a UTM application; the main routine is a component of the openUTM system code.

You specify the task of the service you wish to provide the users of your application with in the business logic implemented in the program units. The program units can be programmed in one of the common programming languages (C, C++, COBOL and others).

You can access UTM system functions and external resource managers such as databases from within the program units. The program units utilize the UTM system functions using integrated **UTM calls**, e.g. for transaction management or to send messages to a communication partner.

You can use various interfaces for these UTM calls: In addition to the KDCS interface described in this manual you can also use the X/Open interfaces CPI-C, XATMI and TX (see the openUTM manual "Creating Applications with X/Open Interfaces").

## Characteristics of the KDCS interface

The KDCS interface (compatible data communication system) has been defined and standardized (DIN 66 265) as a manufacturer-independent interface for transaction-oriented applications. openUTM supports the full extent of this standard and offers significant extensions, e.g. for distributed processing. For an overview of these extensions refer to the table on [page 204ff.](#)

KDCS possesses the following function characteristics:

- extensive range of function calls for universal use (e.g. also for pseudo conversations, message queuing or direct communication with terminals)
- KDCS specific storage area for simple and safe programming
- event functions for event control

KDCS is available for the C, C++ and COBOL programming languages; in BS2000/OSD, it is also available for Assembler, Fortran, PL/I and Pascal-XT.

B

## UTM application program - UTM application

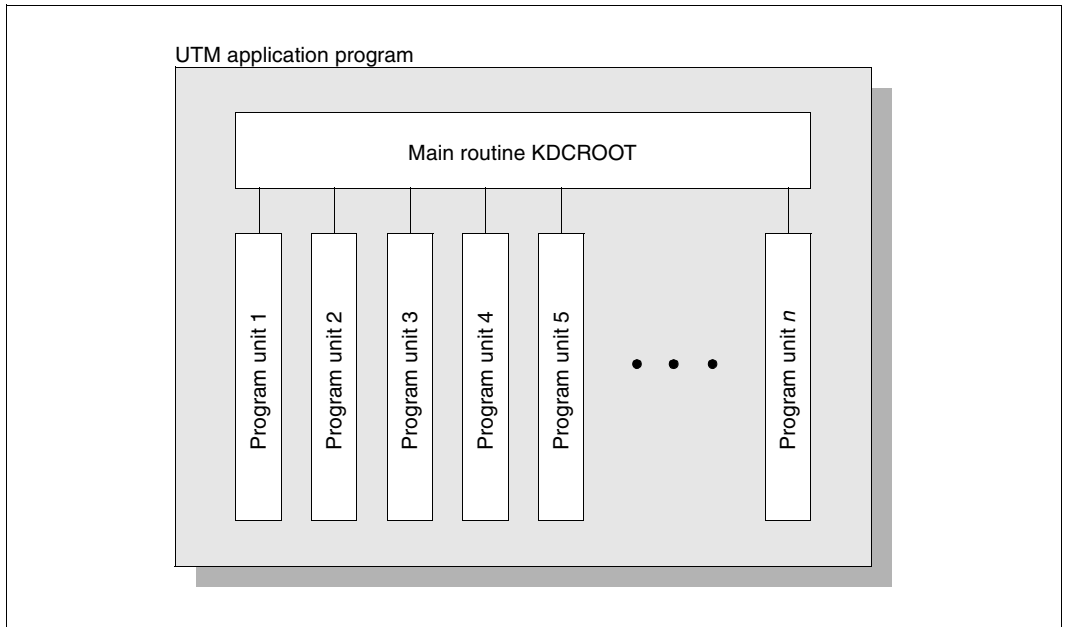
A UTM application program consists of the UTM main routine KDCROOT and the UTM program units.

The main routine KDCROOT controls the flow of the application as part of the UTM system code. It is created when the application is generated (see the openUTM manual "Generating Applications").

In order to run the UTM program units under openUTM management, you have to link the compiled service routines, together with other modules (assignment tables, messages, used libraries, etc.) and the main routine KDCROOT to the **UTM application program** (see the openUTM manual "Using openUTM Applications under BS2000/OSD" and the openUTM manual "Using openUTM Applications under Unix Systems and Windows Systems").

The linking can be done statically (i.e. before the application is started) or dynamically (i.e. when the application is started or during its operation).

At **UTM application** startup the UTM application is started in a number of processes which is specified by you. From a technical point of view, the UTM application is therefore a processing group which constitutes a logical server unit at runtime.



Main routine KDCROOT and multiple program units

The program units and the main routine KDCROOT interact via **KDCS calls**. In a program unit, the KDCS calls are used to inform the main routine KDCROOT which function openUTM is to perform. You use the KDCS parameter area to specify the necessary entries and pass its address as the first parameter with each KDCS call.

Predefined language-specific data structures enable you to structure the KDCS parameter area, in COBOL these are located in the KCPAC COPY element and in C/C++ in the *kmac.h* include file. For a language-independent description of the value to be entered in this area for the individual KDCS calls refer to [chapter "KDCS calls" on page 203ff](#). For language-specific particularities refer to the chapters ["Additional information for C/C++" on page 485](#) and ["Additional information for COBOL" on page 543](#).

The sections below explain the way in which you can structure an application program.



As used in this chapter, the terms "program" and "program unit" refer to the execution of this program unit, not to the program text. For example, the phrase "sequence of calls in a program unit" refers to the sequence in which these calls are run, not the sequence in which they appear in the source program. This is also referred to as the **program unit run**.

## 2.1 The openUTM service concept

### Starting a service

One or more transaction codes are assigned to each program unit, either during application generation or using dynamic configuration.

The transaction code of the first program unit of a service has a special function because it is used to start this service. This transaction code is also called the service transaction code or service TAC for short. openUTM sets up a specific context (storage area, etc.) for each service started.

You can enter the service TAC in a large number of ways, e.g. by entering it at the terminal, selecting a menu item in an alphanumerical format, mouse clicking on a GUI client or using a Web browser.

Together with the transaction code you may transfer a message to openUTM containing necessary data for the required processing.

### Dialog step and processing step

In the simplest case, a service consists of a single processing step, i.e. no interactions are required to process the service request: the result is output following the input of the service TAC, e.g. "All finished".

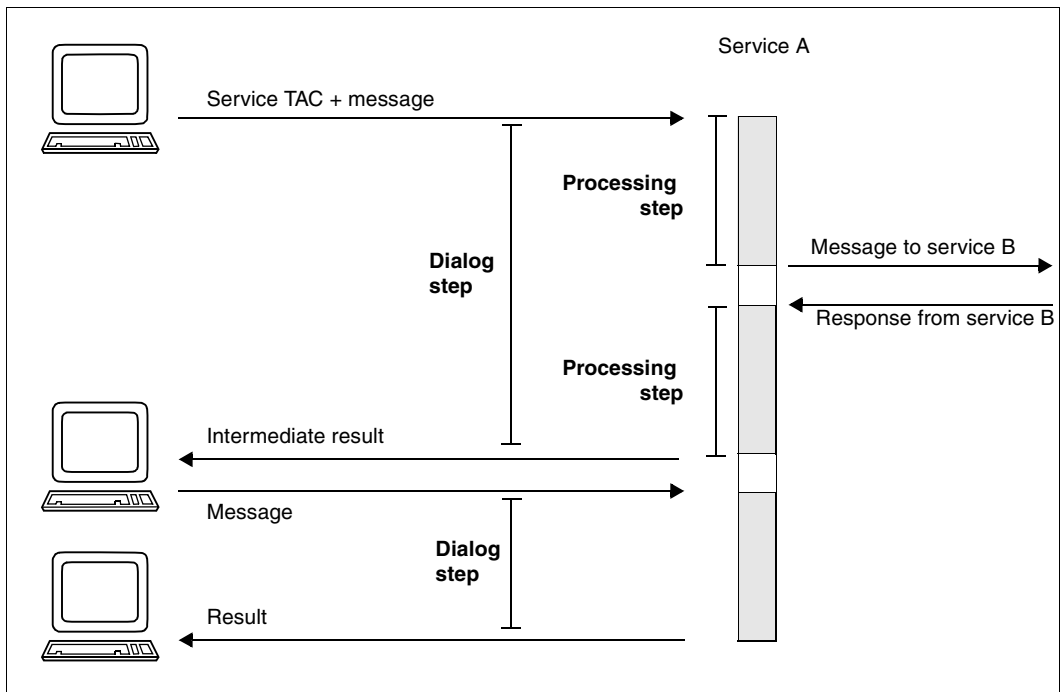
However, in many cases this structure is inadequate: you may have to request additional data, display intermediate results, or take account of individual selection options and branches in the service sequence. A service therefore often consists of multiple dialog steps.

A **dialog step** starts with a dialog message which one communication partner sends to the UTM application, and terminates with a dialog message which the service sends to the same communication partner as a response. Between these two points in time, the data is processed and there is no communication with this communication partner.

In distributed processing, a service does not only communicate with the user who started the service, but also with one or more partner services. A service started by a user may therefore not send the next message to the user but to another service application. In this case, since the message is not a response, it is called a processing step rather than a dialog step: A **processing step** starts when a dialog message is received and ends when the next dialog message is sent. This can be a response to the same partner (in which case the processing step corresponds to a dialog step) or a message to a third party.

A service may therefore be divided into multiple dialog steps, and a dialog step – in distributed processing – into multiple processing steps.





Dialog steps and processing steps

### Switching between inputs and responses

openUTM requires a "strict dialog" to structure dialog services: i.e. each message must be followed by an answer. After sending a message to a service, you must first wait for a response before you can send another message to this partner.

This sequence - together with the modular service structure for processing steps - enables openUTM to maximize process utilization (see following two sections).

### Modular processing steps in service structure

If a service consists of multiple dialog or processing steps, the service does not usually consist of only **one** service routine. Instead, it consists of a sequence of separate service routines called **program units**. Normally a program unit is equivalent to a processing step: i.e. a program unit reads a message and issues a message when it terminates. Subsequently the process is automatically released and is available for other jobs. The next program unit does not start until the next message is received from the communication partner.

For example, the service does not occupy a process while a user reads the output and prepares the next input at a terminal. As soon as the terminal user has finished the input, another process may, under certain circumstances, continue the dialog without notifying the user or the program unit.

Hence openUTM optimizes process utilization and this has a positive effect on system performance. openUTM uses this dialog concept (also called "**pseudo-conversational**") for dialogs with terminal users as well as for program-program communication.

In addition, the basic design for the use of modular processing steps simplifies the design of applications and results in clearly structured programs. However, it is flexible enough not to limit the programmer and makes a number of variations available to openUTM.

For examples of multi-step services and more information about connections between program units refer to [section "Structuring services" on page 40ff.](#)

### **Asynchronous services**

Using openUTM you can define services which run in the dialog with the user as well as services which can be started even when disconnected from the user. The message queuing functionality integrated in openUTM enables you, for example, to disconnect particularly large and non-time critical jobs - such as long statistical calculations or sorting tasks - from online dialogs without discontinuing transaction logging. You can use the message queuing functionality not just to perform processing jobs; it also lets you output messages, e.g. for print jobs, messages to a terminal or messages to service-controlled queues ([page 61](#)).

The message queuing concept and its scope of application are introduced in the openUTM manual "Concepts und Functions", for further information see [section "Message Queuing \(asynchronous processing\)" on page 50ff](#) of the present manual.

## 2.2 Structure of a program unit

Before we explain the structure of more complex services we will describe the structure of a single program unit in this section. First the program framework applicable for all types of program units is explained followed by the structure of a dialog program unit.

The structure of an asynchronous program unit is explained in [section “Structure of an asynchronous service” on page 53](#).

### 2.2.1 Program framework

The interaction between the program units and openUTM is implemented using **KDCS calls**. With these calls you inform openUTM which functions are to be executed. You pass the address of the **KDCS parameter area** in every KDCS call, and in some calls you pass the address of a message area.

The call parameters are passed to openUTM in the KDCS parameter area. You are provided with predefined language-specific data structures for the KDCS parameter area, in COBOL these are located in the KCPAC COPY element and in C/C++ in the *kcpa.h* include file. For a language-independent description of the value to be entered in this area for the individual KDCS calls refer to [chapter “KDCS calls” on page 203](#).

openUTM passes return information after every KDCS call (except for PEND) in the **KB return area**. The evaluation of the return codes tells you if the execution was successful or unsuccessful and can be used to take the appropriate control measures in the program (see also [section “Programming error routines” on page 89](#)).

You will also find the current information on users, services, program units and communication partners after every KDCS call in the **KB header**.

The KB header and the KB return area are part of the **communication area (KB)**, see [page 80](#). openUTM provides a program unit with the address of the KB in a call parameter when the program unit is called.

You are provided with predefined language-specific data structures - for COBOL in the COPY element KCKBC and for C/C++ in the *kcca.h* include file - for the structure of the KB header and KB return area.

The first UTM call in a program unit run is the INIT call. This call initiates the interoperation of a program unit and openUTM. Other program code may precede the INIT call. After the INIT call, openUTM provides current, runtime-specific information in the KB header and in the message area.

The final call in a program unit run is the PEND call. This call terminates the program unit; control is not returned to the program unit once this call has been issued.

The different variants of the PEND call are used to control execution of a UTM service. You have the following options:

PEND PR	ensures that the processing step continues in another program unit without message exchange with the partner.
PEND PA	same as PENDING PR.
PEND PS	specially for the sign-on service, similar to PENDING PR.
PEND KP	terminates the processing step but not the transaction.
PEND SP	terminates the transaction but not the processing step.
PEND RE	simultaneously terminates the processing step and the transaction.
PEND FI	terminates dialog step, transaction and service.
PEND FC	terminates transaction and service and continues the dialog step in another service.
PEND RS	aborts the processing step and resets the transaction to the last synchronization point.
PEND ER	aborts the processing step, resets the transaction, terminates the service and generates a UTM dump. In BS2000 the application program is loaded dynamically; in Unix systems and Windows systems it is restarted.
PEND FR	same effect as PENDING ER but without dynamic loading or restarting of the application program.

Note that many actions, such as the sending of an output message to the communication partner of a UTM service, are not executed by openUTM until the PENDING call is issued, i.e. they are not executed during the program unit run. The source of this behavior is the transaction-oriented manner in which openUTM functions; you can decide if the action executed in this transaction by the program unit is to be committed or rolled back up until the end of the transaction.

If a serious error occurs in a UTM transaction, then openUTM independently resets the entire transaction to the most recent synchronization point and terminates the service (see also the openUTM manual "Concepts und Functions").

## 2.2.2 Structure of a dialog program unit

openUTM requires a "strict dialog" for dialog program units, i.e. each message input must be followed by a message output specifying the result or an error message.

Following **INIT**, the program unit can use **MGET** to read the dialog message, which can be received from a terminal, a client program or from another program.

This message can be:

- a complete message or
- a message segment
- an empty message if, for example, only a TAC has been specified
- a reset message from a program unit which has been terminated using PEND RS
- a return code if, for example, the user pressed a function key. In this case, the message must be read with another MGET.
- in the case of distributed processing, status information from, for example, a job receiver which terminated following an error
- in the case of distributed processing using OSI TP, a handshake request (sent by the partner using MPUT HM) or a negative handshake confirmation (sent using MPUT EM)

After this input has been processed, you have to use **MPUT** to answer your partner's query (end of dialog step). In the case of distributed processing the message can also be addressed to a job-receiving service (no end of dialog step, only end of processing step).

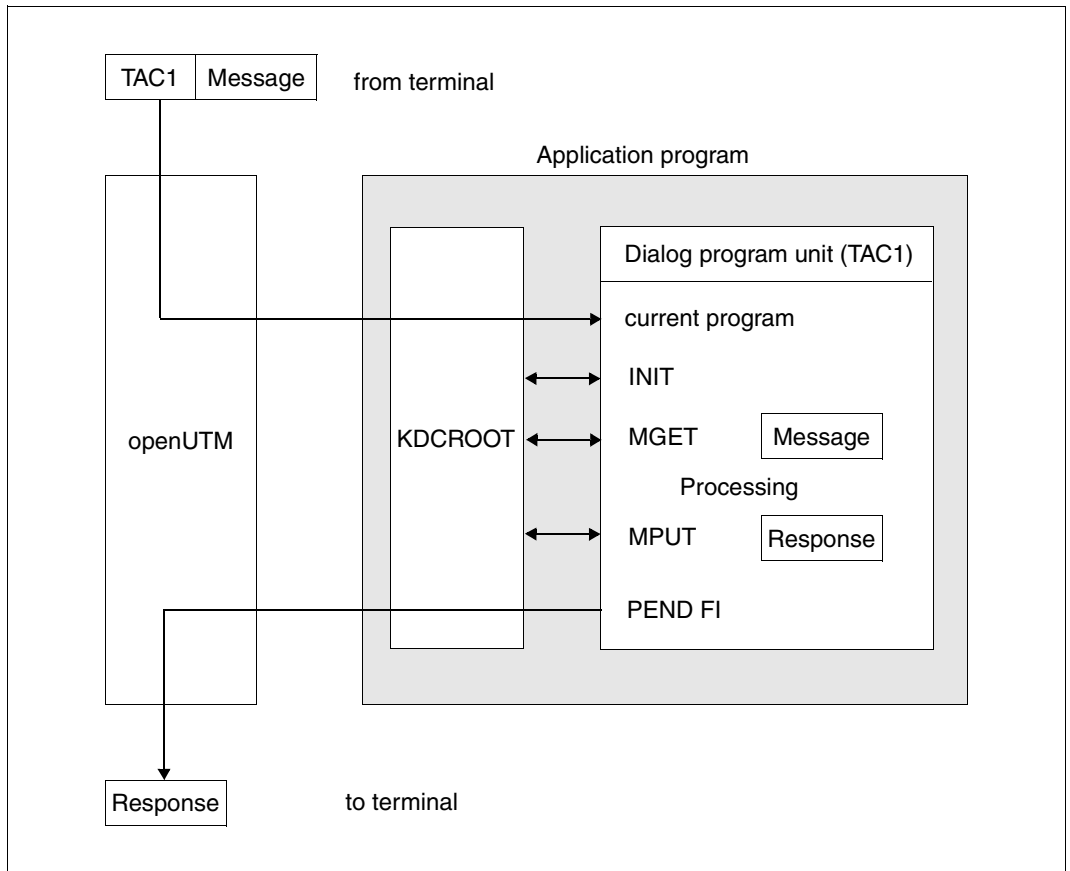
The final UTM call in your program unit must be a **PEND**, as described in [section "Program framework" on page 35](#).

If the PENDING call terminates the processing step, openUTM outputs the message to the terminal, the client program or another program after PENDING processing.



As a general principle, you have to issue an MPUT call before a PENDING call which terminates a processing step. Exceptions to this rule will be indicated explicitly.

The diagram below shows the basic structure of a dialog program unit:



Structure of a dialog program

The transaction code "TAC1" is input by the terminal user. TAC1 was assigned to the program unit at generation time (KDCDEF statement TAC, operand PROGRAM=current program name).

### 2.2.3 Reentrant capability of program units

openUTM always uses a transaction code to call a program unit. However, the program unit is not loaded each time it is called, instead a program copy in virtual memory processes unrelated jobs in sequence. In practice this means that there may be a switch to another process following a PEND, with the result that the subsequent program unit runs in another process and may therefore be confronted by another data environment.

A UTM program unit must therefore be serially reusable, i.e. it must be a reentrant program:

- program-specific data must be set to its initial status at the beginning of the program unit run
- a program-specific data field may only be read if it has already been written in the same program unit run.

openUTM simplifies the programming of reentrant programs for you. It provides a special program unit-specific storage area (SPAB see [page 78](#)) which is managed by openUTM. If you use this storage area for all variable data, openUTM automatically ensures that your programs are reentrant.

What this means for COBOL program units is that variables in the WORKING STORAGE SECTION must be installed again when calling the program unit, assuming they have been defined in an earlier program unit run. For C/C++ this applies to the variables with the storage class attribute *static* or *extern* and to those with an external connection (global module variables without a storage class attribute). For variables whose scope applies to more than one process (i.e. which lie in shared memory) at least write access to such variables must be serialized. However, the variables mentioned above can be used without restrictions for read-only access.

For C and C++, variables with the storage class attribute *auto* or *register* can be used without problems. They must, however, be defined before reading.

## 2.3 Structuring services

A service may consist of one or more program units. The structure of a service consisting of only one program unit and processing a single processing step (single-step service) is illustrated in the diagram on [page 38](#):

A single program unit processes the required task entirely in one step and subsequently terminates it using PEND FI.

For complex tasks which require multiple steps, you can structure a service or transaction into multiple parts. To do this, you have the following options:

- multi-step services
- multiple program units in one processing step
- multiple processing steps in one program unit
- subprogram calls issued by program units
- chained services
- stacked services

You may also use a combination of these options.

You have considerable freedom when designing your UTM program units: the only obligatory rules are that you use INIT and PEND to establish the program framework and MPUT for the processing steps.

### 2.3.1 Multi-step services

Dialogs consisting of multiple processing steps constitute a frequent type of transaction processing. This technique is designed to simplify the work of application users. They should be able to formulate jobs interactively step by step and thus be able to evaluate the interim results at each processing step.

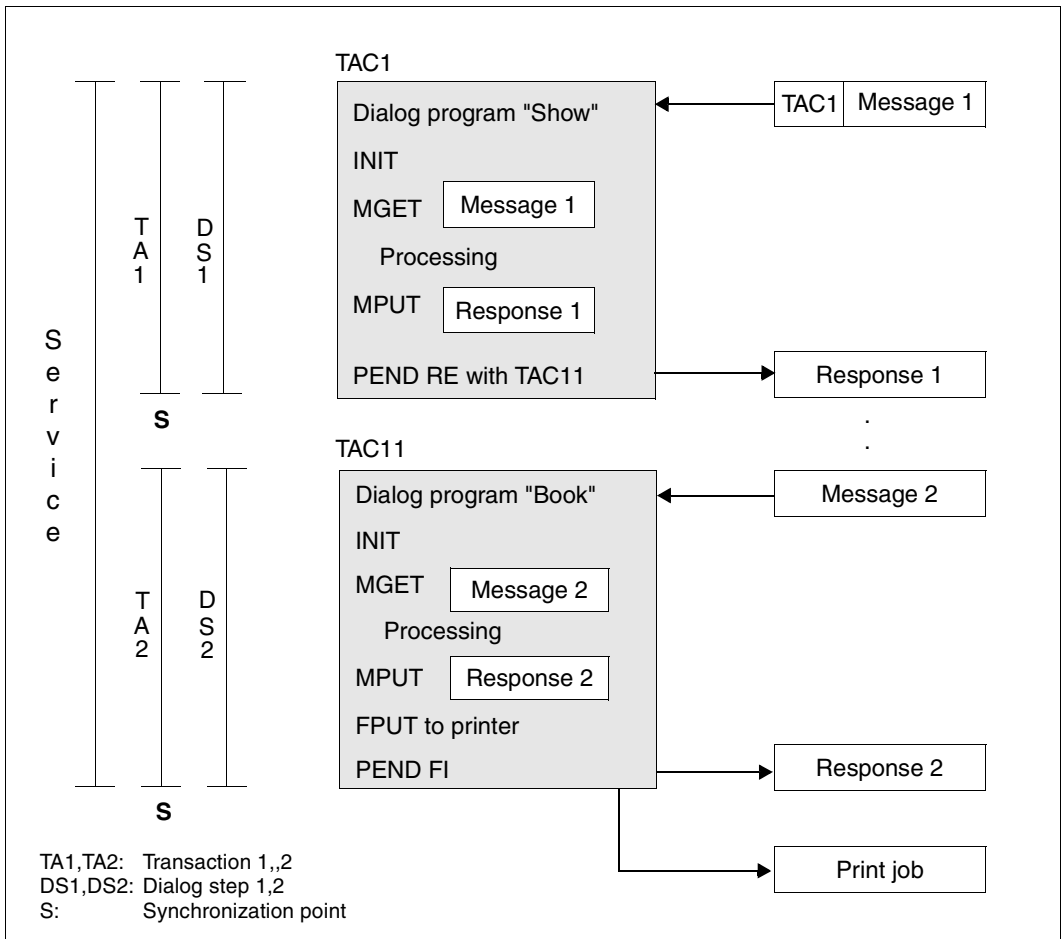
The reservation of train tickets, for example, can be programmed as a service in two steps:

1st step: Query whether seats are still available.

2nd step: Reserve the seat and confirm reservation.

By chaining the programs that implement the individual processing steps, you ensure that the entire service is processed in the right order. You chain program units via entries in the PEND call. Here you select the RE operation modifier in the KCOM field if you also want to set a synchronization point at the end of the processing step and specify the transaction code of the follow-up program in the KCRN field. If you only want to terminate the processing step without setting a synchronization point, you have to select the KP operation modifier instead of RE in the PEND call. The follow-up program is started when openUTM receives the next input message.





Multi-step service

The service presented in the diagram consists of two program units which perform one dialog step each. However, there is no limit to the number of program units that you can combine in a a service. For all program units of a service, openUTM provides service-specific storage areas which can be used by the program units to transfer information (see [page 74ff](#)). These storage areas are included in transaction logging.

The second program unit contains an FPUT call in addition to the MPUT call. This call does not create a dialog message. It creates an asynchronous message, in this case an output message to the printer.

### Same processing step for differing actions

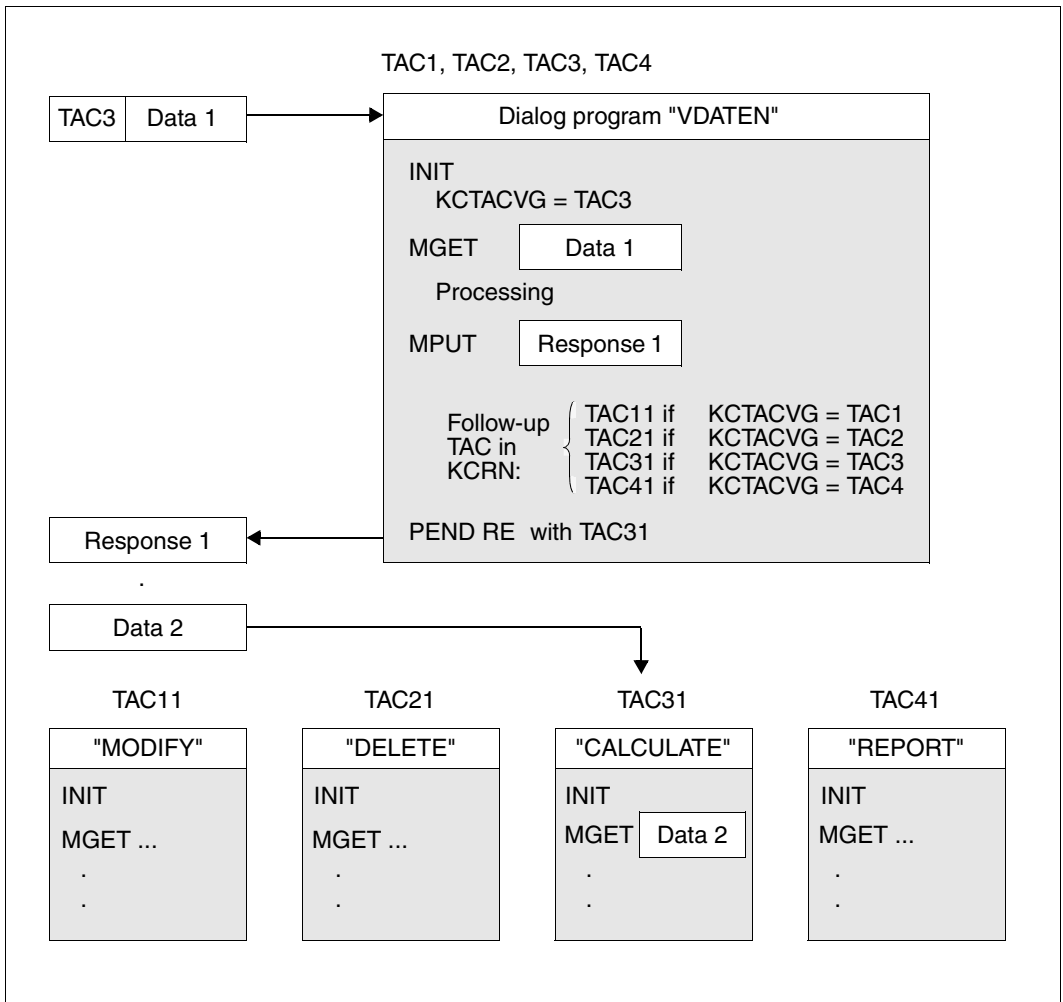
If two or more actions of the same type are performed in an application, it is advisable to handle processing steps that are identical for all actions in a single program unit. This is demonstrated using multiple step services in the example below:

The data of an insurance policy is to be displayed at the outset of different actions, so that one of the following steps can be performed:

- modify data
- delete data
- calculate premium
- report a claim

The first step is identical for all four actions, while the subsequent steps are all different (see diagram below).

First, all four actions are processed in the same program unit. Each action has its own transaction code, i.e. four different transaction codes are assigned to the first program unit. When the program unit is started, openUTM displays the transaction code used for the service start in the KCTACVG/ kccv\_tac field of the KB header. Depending on the transaction code the program unit determines which program unit is started as follow-up program unit, i.e. which follow-up TAC is entered in the KCRN field when calling PEND.



Program unit to which multiple transaction codes are assigned

### 2.3.2 Multiple program units in one processing step

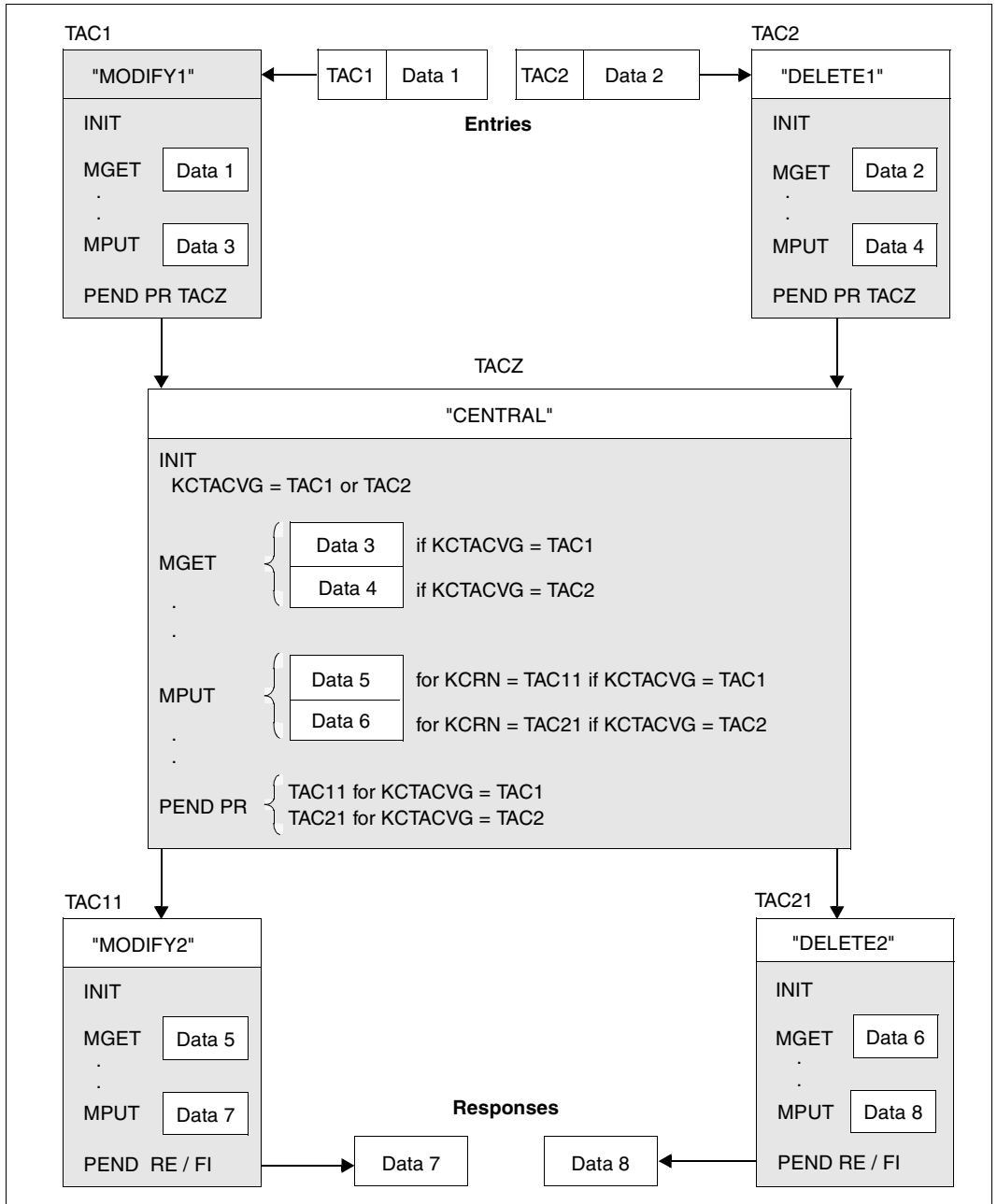
If different processing steps contain identical subtasks, it is advisable to split the individual processing steps into several parts. You write a separate program unit for each subtask and this can be used jointly by all processing steps. This is an exception to the basic rule for a service structure "one processing step = one program unit": here, **one** processing step is performed by **multiple** program units. This is achieved by using the PEND PR/PA/SP calls which terminate the program unit, but not the processing step. If you use PEND SP, a synchronization point is set. If you use PEND PA/PR, the transaction remains open.

In the following diagram, the two processes are single-step services, although each of them consists of three program unit runs:

- The service to which the TAC1 transaction code is assigned changes the data of an insurance contract.
- The service to which the TAC2 transaction code is assigned deletes the data of an insurance contract.

What is common to both services is that they have to change the initial data record. In order to create a separate program unit to perform this common task the processing step is split into individual parts.

After the INIT call the KCTACVG field contains the TAC which was used to start the service. The program unit "CENTRAL" then determines which program unit has to be used to continue the transaction. You have to specify the TAC of this program unit in the KCRN field for both MPUT as well as PEND PR. The MPUT message is not sent to the communication partner, but to the follow-up program unit where it is read using MGET. Since the processing step is not terminated in the program unit "CENTRAL", an MPUT call is not mandatory in this program unit: you may also transfer the data using service-specific storage areas instead of MPUT (see also [section "KDCS storage areas in openUTM" on page 74ff](#)).



Multiple program units performing one processing step

### 2.3.3 Multiple processing steps in a single program unit

A UTM program unit runs in a process between INIT and PEND. Since the process is released at the end of the program unit, a follow-up program unit may run in a different process from the first. This means that the follow-up program unit cannot access the process-specific environment (context), such as resources and program-specific data areas of the first program unit. Usually this is not necessary, since data can either be transferred via an MPUT call issued to the follow-up program or via process-specific storage areas provided by openUTM.

However, openUTM also allows you to retain the process-specific context of a program unit for multiple processing steps. You use a PGWT (program wait) call with the operation modifiers KP, PR, CM and RB. This sets a wait point without terminating the program unit, i.e. the program context is preserved. These variants are used for different purposes.

- PGWT KP terminates the processing step and sends the MPUT message. The program unit waits until the next message arrives from the partner. PGWT KP corresponds to the PEND KP call in the previous program unit and the INIT call in the next program unit.
- PGWT PR waits for a message to a queue without terminating the processing step. PGWT PR corresponds to a PEND PA/PR call in the previous program unit and an INIT call in the next program unit.
- PGWT CM terminates the transaction without terminating the program unit. However, a synchronization point set with PGWT CM is not a restart point. The follow-up transaction can therefore be rolled back with PGWT RB only and not with PEND RS.

The following also applies:

- If an MPUT call was executed before the PGWT CM, the MPUT message is sent and the program unit waits until a response is received from the partner. This PGWT CM corresponds to a PEND RE call in the previous program unit and an INIT call in the next program unit.
- If no MPUT was executed before the PGWT CM call, the program unit is continued immediately. A PGWT CM without a preceding MPUT call corresponds to a PEND SP call in the previous program unit and an INIT call in the next program unit.
- PGWT RB rolls back a transaction.

The functions of two successive program units can therefore be performed by one program unit. The entire functional sequence then runs under one and the same process. The process-specific context is available until the end of the program unit run.

During that period, the program unit occupies a process exclusively. As a result, the UTM application usually requires more resources (processes).

The PGWT call permits the simple integration of program systems which expect a combined SEND/RECEIVE interface into openUTM.

Alongside the PENDING PR/PA/SP calls, the PGWT call represents another way of varying the modular processing step structure:

While the **PENDING PR/PA/SP** calls enable you to split **one** processing step over **multiple** program units, the **PGWT** calls make possible **multiple** processing steps in **one** program unit.

Since valuable resources are involved with the PGWT calls, these calls should be used only sparingly in an application; it should **only** be used where the other options of the KDCS interface are inadequate; the frequent use of PGWT calls can have a negative effect on the performance of a UTM application.

### 2.3.4 Subprogram calls from program units

In a program unit you can also issue subprogram calls, e.g. C/C++ functions or COBOL subprograms. These subprograms may contain subprogram calls themselves. Using subprogram calls you call programs via their program name (in C/C++: function name) and not via their transaction code.

Subprograms can also be written in a programming language other than that of the calling program. The openUTM manual "Using openUTM Applications" describes what you need to take into account and which compilers and runtime systems are required. You will find more information on subprogram calls in the [section "C/C++ program units as subroutines" on page 485](#) and [section "COBOL program units as subroutines" on page 543](#).

The program run must either return to the program unit or be terminated in a subprogram using the PENDING call.

### 2.3.5 Chaining services

Services are usually terminated by the **PENDING FI** (finish) call. If you call PENDING FI a dialog message is sent to the terminal, a client program or another program. The end of service is also the end of the processing step.

However, you can use the **PENDING FC** (finish and continue) call to chain another service and continue the processing step within that service. This is useful, for example, if a dialog message is not to be output to the client or if the use of TACs is to be hidden from the user.

Like PENDING FI, PENDING FC terminates both the transaction and the service and releases service-specific storage areas (LSSBs, KB). Data is transferred to the chained service using an MPUT call; you cannot use service-specific storage areas to do this.

The differences in programming to PEND FI are as follows:

- With PEND FC you have to enter the TAC of the chained service in the KCRN field.
- MPUT is not necessary before PEND FC. However, if an MPUT call is issued then the follow-up TAC must also be entered in KCRN (as in PEND PA/PR).

If the first transaction of the chained service is reset, then openUTM restarts at the synchronization point in PEND FC and restarts the chained service.

### 2.3.6 Stacking services

A terminal user can stack a service, i.e. he or she can interrupt an already started service, insert another service and, when this terminates, continue the interrupted service. A service can only be stacked if it is located at a synchronization point, i.e. directly after a PEND RE. There are two ways of doing this:

- by pressing a function key generated with SFUNC ...,STACK=...
- by using the event exit INPUT

#### Continuing a stacked service

A stacked service, also known as a predecessor, is reactivated as soon as the inserted service is terminated by PEND FI. The terminated service generates an output message. Since openUTM only permits service stacking at a synchronization point, both the last output message and the service-specific areas (KB, LSSB) of the predecessor are still available. You use the (last) MPUT call in the inserted service to determine how the output messages of the two services are to be processed. You have three options:

- MPUT NE in the inserted service outputs the message of this service on the screen together with message K096. The terminal user presses Enter to receive the last output message of the predecessor.
- MPUT PM with KCLM = 0 immediately outputs the last output message of the predecessor (PM stands for "predecessor message").
- MPUT PM with KCLM > 0 (permitted only in format mode) overwrites the output message of the predecessor with the message of the inserted service to the length defined in KCLM. openUTM then outputs this revised predecessor message. The format specified in MPUT PM should be a partial format of the predecessor format.

MPUT PM is only permitted in an inserted service if the program unit is terminated by PEND FI, i.e. only in the last processing step of the service.



**Service stacks**

Service batches are formed by stacking services. You can retrieve information about the current service stack in the following ways:

- the KCHSTA and KCDSTA fields of the KB header show the size of the stack and how this has changed since the last program unit run
- the INFO call INFO PC (predecessor service) provides information about the direct predecessor in the batch.

PEND FI in an inserted service always returns you to the immediate predecessor; a stack is deleted when the last inserted service has terminated.

## 2.4 Message Queuing (asynchronous processing)

Message Queuing (MQ) is a form of communication in which messages are exchanged via intermediate queues (store and forward) instead of being directly exchanged. Because there is no synchronization between the sending and receipt of messages, message to message queues are also called **asynchronous messages**. Communication takes place by means of **asynchronous jobs**. An asynchronous job consists of the asynchronous message, the recipient of the message and possibly also the desired time of execution.

Message queuing is only supported locally in the node in the UTM cluster applications. This means that asynchronous messages can only run and be read, displayed or administered in the node application in which they were created.

The one exception is that you can transfer asynchronous jobs from a terminated node application into a running node application using the online import facility.

openUTM provides you with two types of message queues:

- **UTM-controlled queues:**

In the case of UTM-controlled queues, the interposed queuing mechanism is made available in its entirety by openUTM. In other words, in addition to pure queuing functionality, openUTM also takes on the subsequent processing of the message (e.g. output to a communication partner or startup of a service).

- **Service-controlled queues:**

In the case of service-controlled queues, a service is responsible for the further processing of the message. In other words, openUTM provides only the queuing functionality. The communication partner for whom the message is intended must read the message from the queue independently (using the KDCS call DGET). If there is no message in the queue, a service can also wait for the arrival of a message.



For general information about the message queuing concept and how to use it, refer to the *openUTM manual "Concepts und Functions"*.

## 2.4.1 Messages to UTM-controlled queues

In the case of UTM-controlled queues, openUTM monitors the subsequent processing of messages written by a program unit to UTM-controlled queues. In other words, when a message is sent to a UTM-controlled queue, it is specified how the message is to be processed subsequently. The jobs associated with this message are therefore classified according to the recipient:

- output jobs
- background jobs

in the case of background jobs we distinguish between:

- **local** background jobs  
i.e. background jobs, which request services from their own application.
- background jobs which request **remote** services  
There is more information on this in the section [“UTM-controlled queues in distributed processing”](#) on page 190ff.

### 2.4.1.1 Output jobs

Output jobs are asynchronous jobs which output a message, for example a document, to a printer or a terminal. However, the output target may also be another application connected via a transport system interface.

Output jobs consist of the target specification together with the asynchronous message to be output.

Output jobs are initiated by corresponding MQ calls from a program unit of the UTM application.

### 2.4.1.2 Background jobs

Background jobs are asynchronous jobs which are addressed to an asynchronous service of its own or a remote application. Background jobs are particularly suitable for time consuming or non-time critical processing, in which the result has no direct effect on the current dialog.

Background jobs consist of the transaction code (TAC) of the program unit used to start the background job (service TAC) and possibly a message for the program unit. The type of transaction code determines whether the job is to be processed as an asynchronous job or as a dialog message.

Background jobs can be created as follows:

- by an input from a terminal
- by an MQ call from a service of the UTM application
- by a message from another application that communicates with the UTM application via the LU6.1, LU6.2 or OSI TP protocol
- by an input from another application connected via the transport system interface
- by a UTM message when the message is assigned the message destination MSGTAC (i.e. event-driven, see [page 473](#))

Background jobs can be restarted after abnormal termination of a service (redelivery), see [page 60](#) or sent to the dead letter queue.

### 2.4.1.3 MQ calls of the KDCS interface

openUTM provides calls for UTM-controlled queues that are powerful in terms of the functions they offer but nevertheless easy to program. The "free" element in the call names reflects that message queuing is a type of communication which is disconnected from the sender and which is not dependent on the availability of the receiver.

- **FPUT (Free message PUT)**  
You use FPUT calls to send asynchronous messages. The target may be the output device (output job), an asynchronous service (background job) or an application (see [page 190](#)).  
An asynchronous message may also consist of multiple message segments. In this case you have to use a separate FPUT call for each message segment.
- **DPUT (Delayed free message PUT)**  
The DPUT call can also be used to send an asynchronous message or a message segment to an output device, an asynchronous service or another application (see [page 190](#)). However, compared to the FPUT call, the DPUT call also allows you to use time control and confirmation jobs.

- **FGET (Free message GET)**  
You use the FGET call to read asynchronous message or message segments within an asynchronous service.
- **MCOM (Message COMplex)**  
You use the MCOM call to assign confirmation jobs to asynchronous jobs.
- **DADM (Delayed free message ADMINistration)**  
You can use the DADM call to request summary information about the entire contents of a queue or information about its individual elements. Additionally you can also control the processing sequence with DADM: you can advance jobs, cancel individual jobs or delete the entire queue.

For the precise format and further information about these calls refer to the [chapter “KDCS calls” on page 203](#).

#### 2.4.1.4 Structure of an asynchronous service

An asynchronous service starts with an asynchronous program unit. The program unit is assigned a transaction code TYPE=A (asynchronous) at generation. An asynchronous program unit does not only differ from a dialog program unit in its transaction code type, it is also differently structured.

##### Structure of an asynchronous program unit

Asynchronous programs do not have to read an input message or create an output message. In the first program unit and first processing step you can issue an FGET call following an INIT. This FGET receives the asynchronous message from the partner. This may be:

- a complete message
- a message segment or
- an empty message in cases where the job consists of one TAC only (e.g. created via a function key at the terminal).

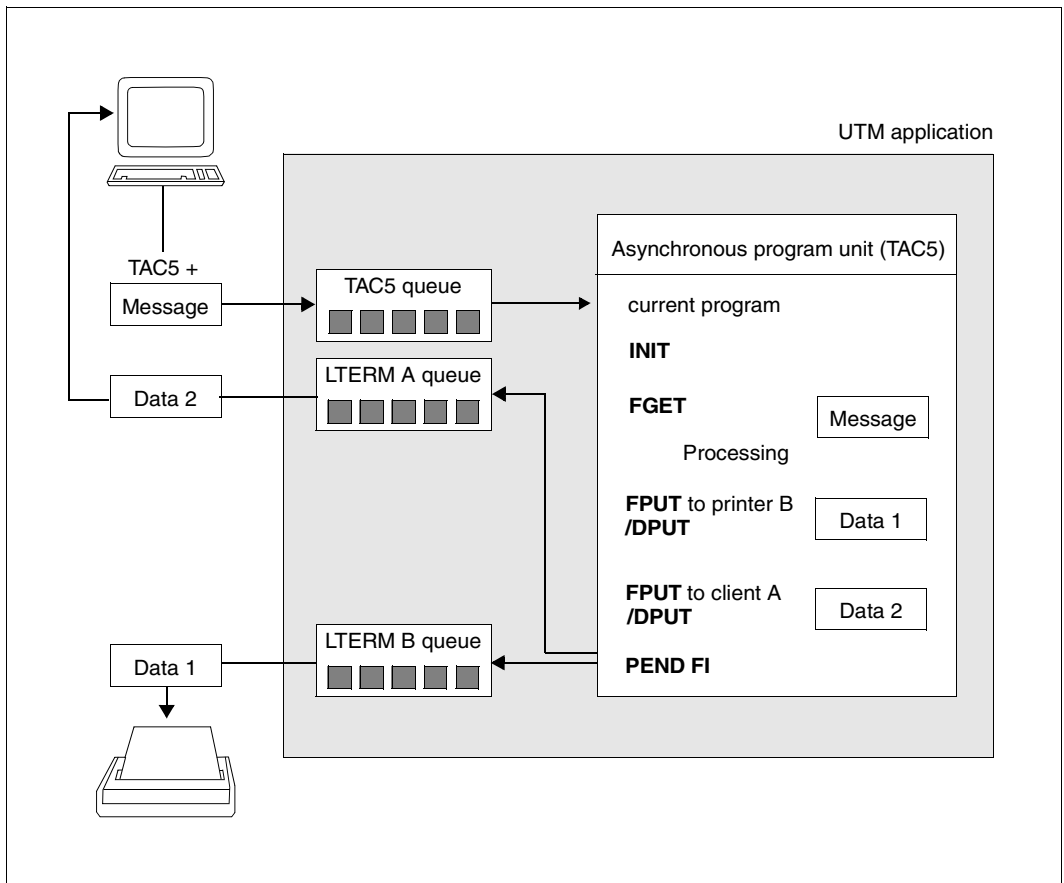
These may be followed by any number of KDCS calls, with the exception of MGET. However, you can use an MGET call in a follow-up program or follow-up processing step.

You cannot use MPUT to send a response to the communication partner from which you received the asynchronous message. Since the service runs disconnected from the partner, the partner may no longer be connected to the application at the time of processing. However, you can send a message to the partner with FPUT or DPUT. This message will be inserted in the message queue assigned to the partner. An MPUT call is only permitted if the message is addressed to a follow-up program or a job receiver service.

The last UTM call in your program unit must be PENDING, as described in [section “Program framework” on page 35ff](#). An asynchronous program is normally terminated with a PENDING FI call. This also terminates the service. openUTM transfers any unsent messages to the partner or partners.

Other PENDING variants are possible, such as PENDING PA/PR/SP for program unit chaining, and PENDING KP and RE for distributed processing. You can also use the PGWT KP call in an asynchronous program unit if you want to terminate the processing step without terminating the program unit or the transaction in distributed processing.

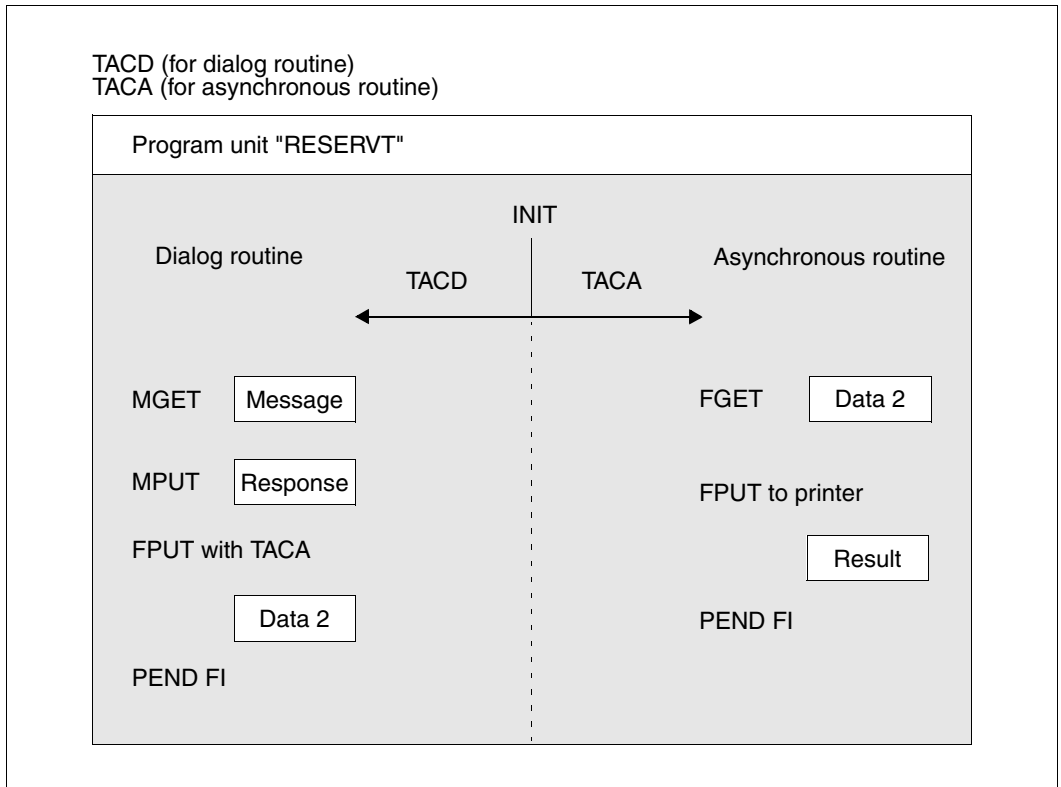
An asynchronous service may be split into multiple program unit runs, and also into multiple processing steps in the case of distributed processing (see [page 56](#) or [page 57](#)).



Structure of an asynchronous program unit

### Combined dialog and asynchronous program

You can assign multiple transaction codes to a program unit. It is also possible to assign a dialog TAC and an asynchronous TAC to the same program unit. This means that you can start the program in dialog mode or asynchronously. openUTM indicates the way the program unit was started in the KCPRIND field of the KB header: "A" for asynchronous, "D" for dialog. The program unit can evaluate this field and branches accordingly.



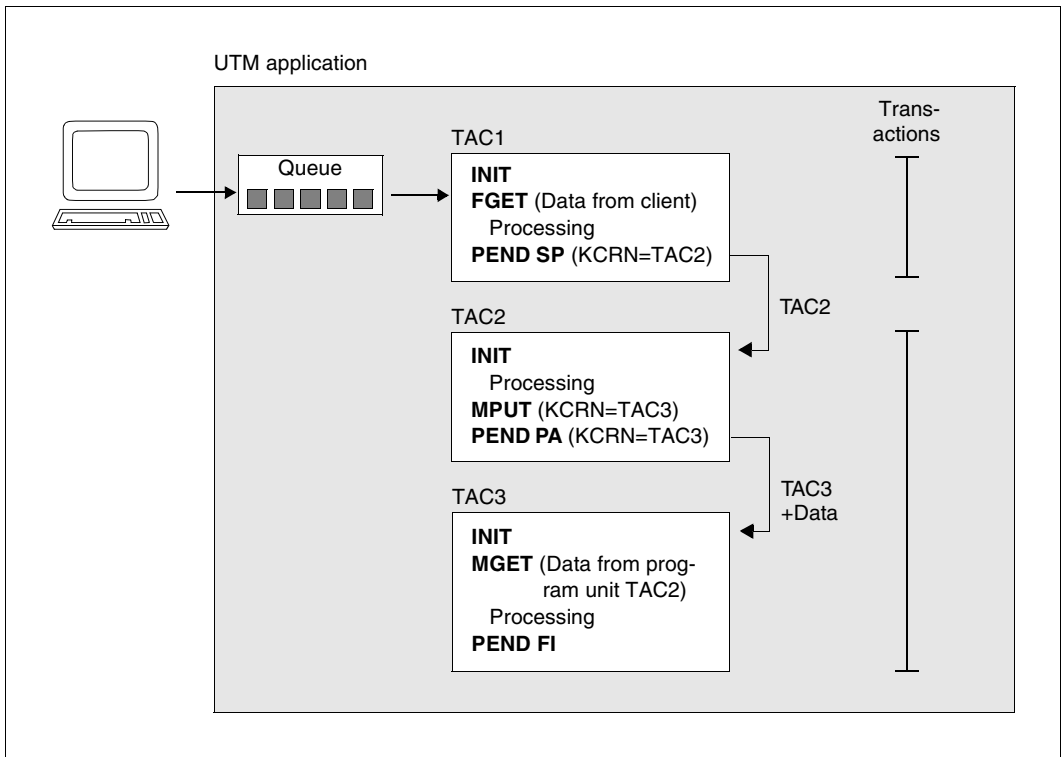
Dialog and asynchronous processing in a program

If the program in the diagram is started with TACD, FPUT is executed after PEND. This means that the same program unit is called again to process an independent second service. The processing of an asynchronous service outside the dialog program may lead to better response times in the dialog with the client.

### Asynchronous service with multiple program units

An asynchronous service may be structured into multiple program units and transactions. The last program unit is terminated with PENDING FI. In the preceding program units the PENDING variants SP or PA/PA are possible.

PENDING variants such as PENDING KP or PENDING RE which terminate a processing step are only possible within an asynchronous service if distributed processing is used (see diagram on next page).



Structure of an asynchronous service split into three program units

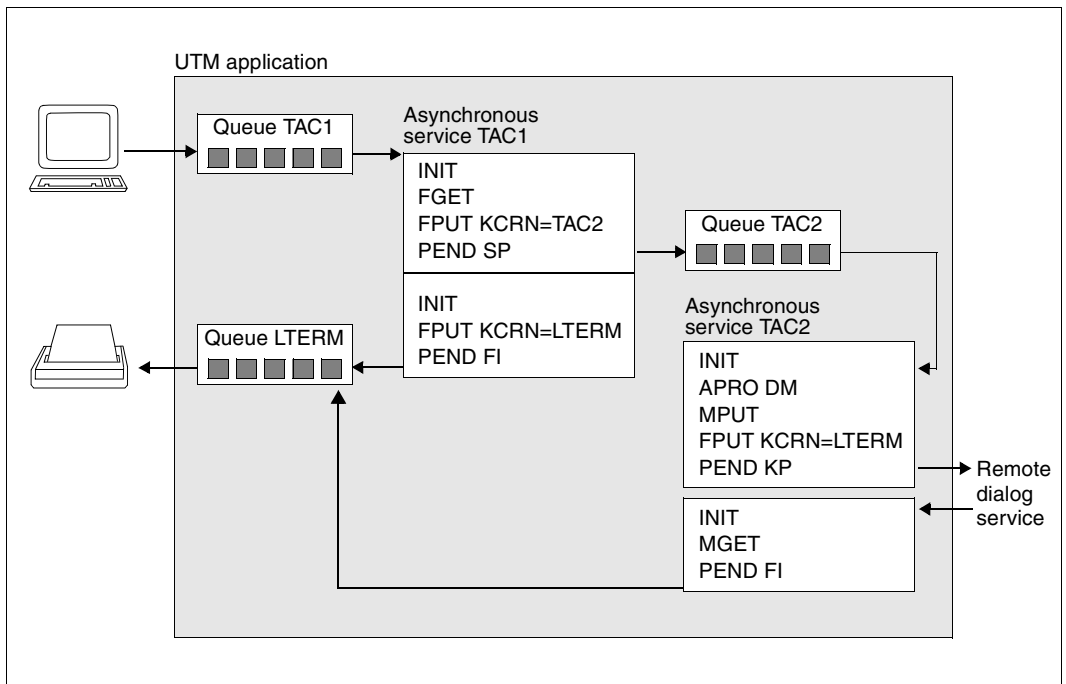
In the depicted example a background job is issued to an asynchronous job of the same application from a terminal. To do this, you have to enter the transaction code of this service and, if necessary, a message at the terminal. openUTM automatically places the job in the relevant queue and starts the asynchronous service disconnected from the job submitter as soon as the necessary resources are available. The first program unit reads the data with FGET and terminates with PENDING SP. A synchronization point is set and the follow-up program specified in KCRN is started.



The TAC1 program has not transferred an MPUT to the TAC2 program. However information can be forwarded in the service-specific storage areas. The TAC2 program selects an MPUT call for the transfer of information to TAC3 and terminates with PEND PA. This means that no synchronization point is set. In the TAC3 program the service is terminated with PEND FI.

### Asynchronous services which also issue jobs

An asynchronous service may in its turn create asynchronous jobs. These may be output jobs or further background jobs. In distributed processing, you can also issue dialog jobs to partner applications from an asynchronous service, i.e. the asynchronous service communicates with remote dialog services.



Asynchronous services which also issue jobs

In the first program unit of the TAC 1 asynchronous service, a job is issued to the TAC2 asynchronous service. To do this, you have to specify the TAC2 transaction code in the KCRN field when you call FPUT. Since the program unit terminates at a synchronization point, the job is already inserted in the relevant queue at the end of the program unit.

An asynchronous job is also issued in the TAC2 service of the first program unit. This is an output job to the printer. You have to specify the LTERM name of the printer in the KCRN field when you call FPUT. Since this program unit does not terminate at a synchronization point the job is inserted into the LTERM queue at the end of the service (next synchronization point).

The TAC 2 service is divided into two processing steps. The first program unit uses MPUT to send a message to a remote dialog service. The second program unit is started when the response is received from the remote service. This is read using MGET.

### Job complexes

Together with the asynchronous job ("basic job") you can describe up to two more **confirmation jobs** which are associated with the positive or negative result of job processing. These confirmation jobs are processed after the basic job is processed. The job submitter can use the confirmation jobs to react to a positive or negative job result. A confirmation job which is not used, e.g. a negative confirmation job for a positive result, is deleted. The entity formed by a basic job and its associated confirmation jobs is called a **job complex**.

The following table shows some of the possible events which may occur during asynchronous processing and the effects of these events on the confirmation jobs.

The beginning and the end of a job complex are defined by means of separate KDCS calls, the **MCOM BC** (begin of complex) call and the **MCOM EC** (end of complex) call. With the MCOM BC call you define the complex identifier (complex ID), the destination of the asynchronous job and the TACs of the asynchronous programs which are to process the positive or negative confirmation. All the jobs generated within the complex are described using DPUT calls. You must specify the complex ID as the target.

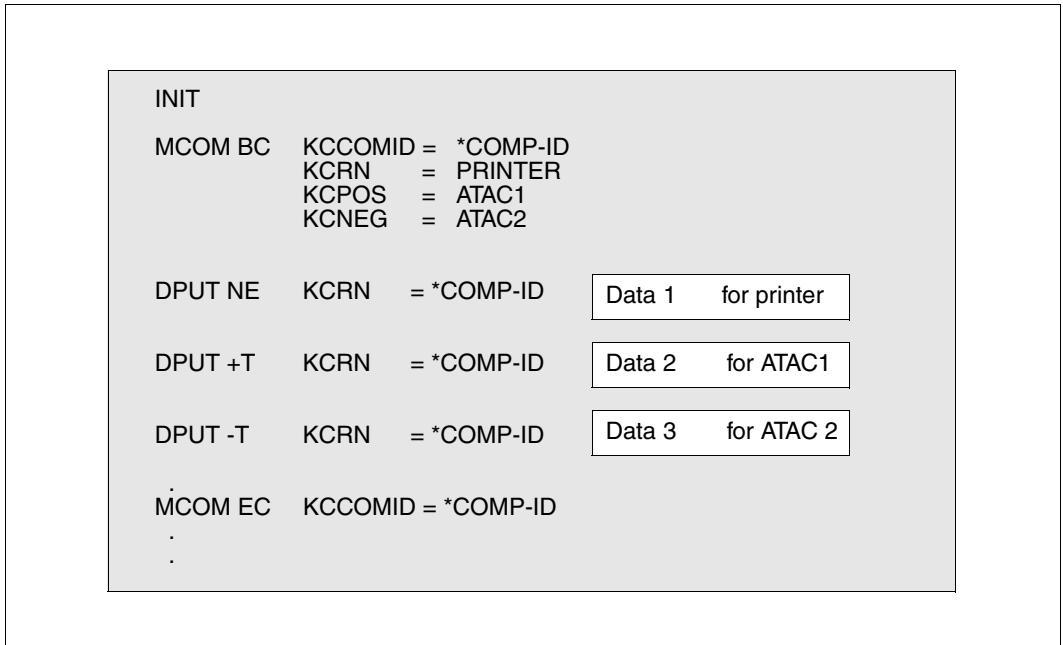
A job complex can be defined both in dialog and in asynchronous programs.

If the basic job of an job complex is addressed to a remote asynchronous service, then the service identifier must be assigned (via APRO AM) before the beginning of the job complex. With MCOM BC you define the service ID in KCRN; the confirmation jobs have to be processed by the local application.

B  
B

Event	Effect on confirmation job
PEND FI in the local asynchronous service	Start of positive confirmation job and deletion of negative confirmation job
For background job, successful transfer to asynchronous service	
Positive print confirmation from printer or print administration	
Processing a message of a TAC queue and termination of the transaction	
PEND ER/FR in local asynchronous service without redelivery	Start of negative confirmation job and deletion of positive confirmation job
Error while preparing the output message by VTSU-B in BS2000	
Error while formatting the output message	
Processing a message of a TAC queue and rollback of the transaction without redelivery	
For background job to asynchronous service, rejection of the job	
Deletion of a job by administration	
Delete the output job when establishing or clearing down the connection of a RESTART=NO client.	
PEND ER/FR in the local asynchronous service with redelivery	Without effect, since basic job is still present
Negative print confirmation from printer or timeout waiting for a confirmation	
Sequence of jobs changed by administration	
Repetition of a print job	Confirmation jobs are deleted; logged in SYSLOG
Processing a message of a TAC queue and rollback of the transaction with redelivery	
Deletion of a job with confirmation jobs by print administration	Confirmation jobs are also not accepted (see KDCUPD log)
KDCUPD does not accept a job	

This diagram illustrates a job complex, taking a print job as an example.



Job complex for a print job

### 2.4.1.5 Redelivery with background jobs

If an asynchronous service is terminated abnormally by PENDING/FR or by a system PENDING before a transaction was completed with Commit, the service is restarted and the FGET message is redelivered, providing this functionality has not been deactivated at generation.

Whether redelivery is possible and how often redelivery is attempted can be defined at generation (REDELIVERY operand in the MAX statement).

When the FGET call is executed, the number of redeliveries is output in the KB return area.

### 2.4.1.6 Saving incorrectly processed messages in the dead letter queue

At generation time, it is possible to define whether, as an alternative to redelivery or as the last fallback stage after the maximum permitted number of redelivery attempts, messages relating to asynchronous transaction codes with CALL=BOTH/FIRST and TAC queues are to be stored in the global dead letter queue. The dead letter queue therefore collects letters which could not be processed.

In the event of persistent errors, the dead letter queue makes it possible to prevent message loss without entering an endless loop.

## 2.4.2 Messages to service-controlled queues

In the case of service-controlled queues, the subsequent processing of messages sent by a program unit to a service-controlled queue is dealt with by the services of the application. openUTM merely ensures that the messages in the queues are saved. The application programs must themselves read the messages saved in these queues; if there are no messages in a queue, an application program unit can also wait for a message to arrive in a queue.

openUTM distinguishes between USER queues, TAC queues and temporary queues. The properties of these queues are described in sections [“USER queues” on page 61](#) to [“Temporary queues” on page 63](#). The associated MQ calls are listed in [section “MQ calls of the KDCS interface” on page 64](#). For more information on the lifetime of queues and messages in queues, refer to [section “Lifetime of queues and queue messages” on page 64](#).

Service-controlled queues offer new communication opportunities in a large number of cases; it is possible, for example, to use message queues to implement the following scenarios:

- communication between independent services in an application
- “pseudo dialogs” with remote transport system or socket applications
- parallel processing of database accesses in (read) transactions
- sending of messages to UTM users (mailbox functionality)
- sending of asynchronous messages to UPIC clients
- sending of messages to queues in other applications (remote queues: see [section “Service-controlled queues in distributed processing” on page 193](#))
- outputting of UTM messages at the UTM administration workstation WinAdmin
- implementation of user-controlled processing of asynchronous messages
- serialization of program units (running in the dialog and asynchronously)
- output of the data of other TLS blocks to the data station in the dialog
- as global storage areas of unlimited size

The queues can be administered with the DADM call.

### 2.4.2.1 USER queues

USER queues are permanent, service-controlled message queues. A USER queue is available to every generated UTM user at all times. USER queues can be accessed by any service by means of a program call, provided it knows the name of the user.

By means of this queue, messages can be sent to the user at the terminal, for example, or to a UPIC user. The USER queue is used as a mailbox in this case. The queue can also be used for communication between the dialog service of the user and asynchronous services that run under the same user ID.

In this way, you can, for example, distribute multiprocessor-capable, parallel database information across a number of asynchronous services and “collect” the results in the dialog service.

At generation, read and write protection can be assigned for the USER queue in the KDCDEF statement USER (Q-READ-ACL and Q-WRITE-ACL parameters). You will find more information on the generation of data access control in the openUTM manual “Generating Applications“ under the KDCDEF statement USER. Regardless of the data access control in existence, a user can always read messages from and send messages to his or her own user queue.

Time-controlled messages cannot be sent to USER queues.

By means of the KDCS call INIT PU, a program unit can query how many messages there are for the user ID under which it is running.

### 2.4.2.2 TAC queues

TAC queues are permanent, service-controlled message queues. Each TAC queue has a fixed name that is generated with the KDCDEF statement TAC ... TYPE=Q. TAC queues can be accessed by any service by means of a program call, provided it knows the name of the queue.

Remote message queues, for example, can be implemented with the help of TAC queues. These remote queues are addressed in the local application by means of the LTAC name (see also [page 193](#)).

TAC queues can also be used in job complexes. Both the basic job and the confirmation jobs can be directed at TAC queues.

Messages can also be sent to TAC queues with time control.

At generation, read and write protection can be assigned for the TAC queue in the KDCDEF statement (READ-ACL and WRITE-ACL parameters). By means of dynamic administration (STATUS attribute), TAC queues can be locked completely for read and/or write access.

For more information on the generation of TAC queues, refer to the section on the KDCDEF statement TAC in the openUTM manual “Generating Applications“.

### Dead Letter Queue

The dead letter queue plays a special role here. The **dead letter queue** is a TAC queue with the fixed name KDCDLETQ. It is always available to save queued messages sent to transaction codes or TAC queues but which could not be processed. During generation, the saving of queued messages in the dead letter queue can be activated or deactivated for each message destination individually using the TAC statement's DEAD-LETTER-Q parameter.

To be able to process the messages in the dead letter queue, for example after eliminating an error, you must either reassign them to their original destination or assign them to a new destination. You can use DADM MV to move an individual message and DADM MA to move all the messages in the dead letter queue to a specified or to the original queues.

The K134 message makes it possible to monitor the message volume arriving in the dead letter queue (see openUTM manual "Generating Applications", DEAD-LETTER-Q-ALARM operand in the MAX statement).

### 2.4.2.3 Temporary queues

Temporary queues can be created dynamically in the program unit run by means of the KDCS call QCRE and deleted by means of the QREL call. No administration authorization is necessary for these calls.

Temporary queues are particularly suitable for communication between services: a service creates a temporary queue and passes the name of this queue with the asynchronous services it creates. The service subsequently reads the messages that these asynchronous services send to the temporary queue or waits for these messages. The temporary queue is then deleted again. When a new temporary queue is created, messages can be written to this queue in the same transaction. However, these messages cannot be read and administered until after the transaction is successfully completed.

The name of the temporary queue can either be freely selected or created automatically by openUTM. The advantage of having names created automatically by openUTM is that the names assigned are always new. Only after 100 million calls is the same name assigned again.

A temporary queue can be deleted at any time by means of the QREL call. On the successful completion of the transaction, all the messages in the queue are deleted and the name and table space of the queue are released. All services waiting for messages of the deleted queue are continued.

Time-controlled messages cannot be sent to temporary queues.

The maximum number of temporary queues that can be created is specified at the generation of the application (QUEUE statement, NUMBER operand: see also the openUTM manual "Generating Applications"). In the QUEUE statement it is also possible to limit the number of messages stored in the queue using the QLEV (queue level) operand; this allows you to limit the utilization of the page pool. However, you can also specify the queue level dynamically in the QCRE call. If you do this, anything specified at generation is overwritten.

A program unit can obtain the names of all the temporary queues and their properties by means of the KDCADMI call KC\_GET\_OBJECT (object type: KC\_QUEUE).

#### 2.4.2.4 MQ calls of the KDCS interface

The KDCS interface provides the following calls for service-controlled queues:

- **DGET (Delayed free message GET)**  
The DGET call is used to read messages or message segments from USER, TAC or temporary queues and to wait for a message of a service-controlled queue. Read with subsequent deletion ("processing") and read without deletion ("browse") are possible.
- **FPUT (Free message PUT)**  
FPUT calls are used to send messages to TAC queues. A message can consist of several message segments. A separate FPUT call is required for each message segment.
- **DPUT (Delayed free message PUT)**  
The DPUT call sends a message or message segment to a USER, TAC or temporary queue. Time control and basic and confirmation jobs are only possible for TAC queues.
- **QCRE (Queue CREATE)**  
The QCRE call is used to create a temporary queue.
- **QREL ((Queue RE)lease)**  
The QREL call is used to delete a temporary queue.
- **DADM (Delayed free message ADMinistration)**  
You can use DADM to obtain an overview of the entire contents of a queue or of specific elements in it. In addition, DADM also allows you to control the order in which messages are processed: you can bring messages forward or delete individual messages or the contents of the entire queue.

You will find the exact format of these calls and further information on them in [chapter "KDCS calls" on page 203](#).

#### 2.4.2.5 Lifetime of queues and queue messages

In UTM-S, all queues and all messages in these queues are preserved after the end of the application. In other words, the messages in the queues are stored safely (failsafe) and are still available after a restart. In the event of regeneration, the messages can be transferred using the KDCUPD utility.

In UTM-F, the USER and TAC queues are preserved after the end of the application, but the temporary queues are lost. The messages stored in all three types of queue are lost.



**Redelivery of queue messages**

If queue messages were processed and if the transaction is then rolled back, the messages are placed back in the queue. They can be read with DGET. The maximum number of redeliveries can be set at generation (REDELIVERY operand in the MAX statement). Redelivery can also be deactivated using this operand.

When the DGET call is executed, the number of redeliveries is output in the KB return area.

Alternatively, such messages can also be saved in the dead letter queue.

**2.4.2.6 Deleting USER and TAC queues by means of programmed administration**

USER and TAC queues can be deleted by means of programmed administration.

A USER queue is deleted when the associated user ID is deleted. The deletion can take effect either immediately or at the next generation. However, the immediate deletion of USER queues is only possible for standalone applications.

The dead letter queue cannot be deleted.

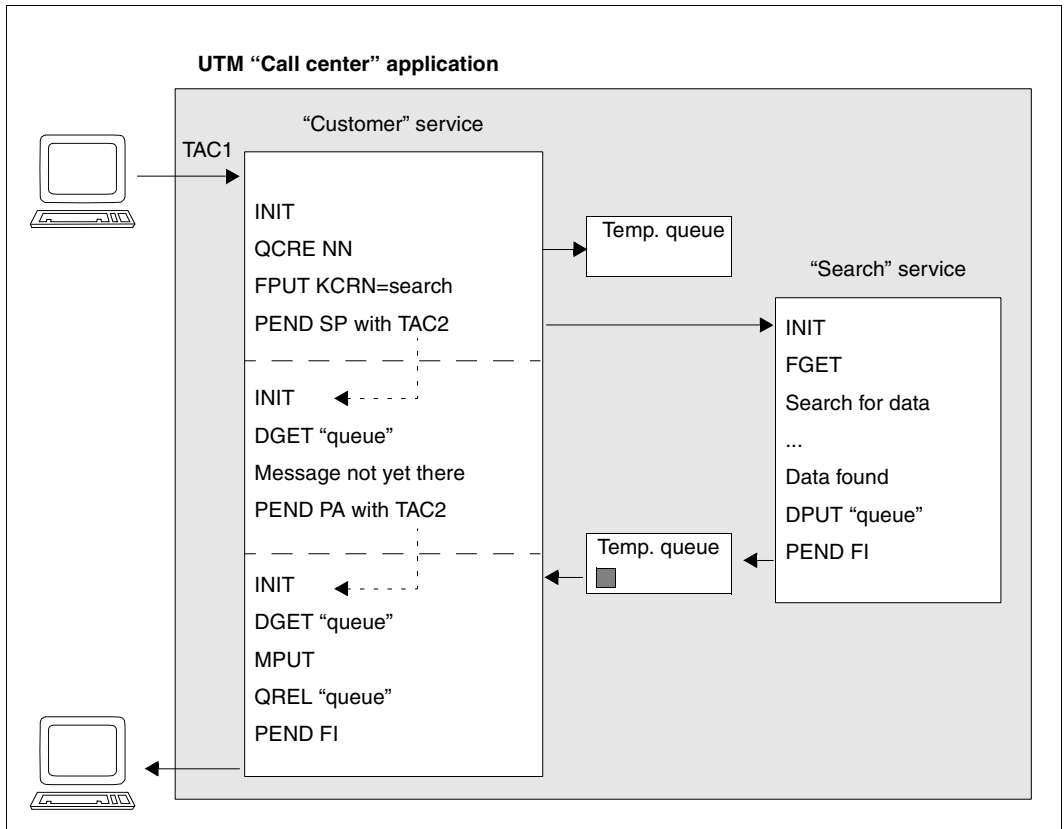
The deletion of TAC queues does not take effect until the next generation.

2.4.2.7 Examples

The following sections describe typical application examples and show how you can use service-controlled queues and create the associated program units.

**Example 1: Communication between two services in an application**

The diagram shows the evaluation of a background job with the help of a temporary queue. The main service “Customer” in a call center searches for the required data by means of a background job and reads the results from a temporary queue or waits for the corresponding message if it is not yet available at the time of the DGET.



Communication between two independent services with the help of a temporary queue

A temporary queue is created by means of QCRE NN. The name of the queue is returned in KCRQN. The background job “Search” is then created with FPUT, and the transaction is terminated immediately with PEND SP. The name of the queue is passed with FPUT.

The “Search” service reads the message with the name of the temporary queue and searches for the required data. Meanwhile, the main service, “Customer”, queries the result of background processing with DGET. Since the result is not yet available in this case, the current program unit of the main service terminates; openUTM waits for the arrival of the message. The function is implemented in such a way that there is no program unit active and no process linked for the main service in the wait time.

When the “Search” auxiliary service has obtained the required data, it creates a message for the temporary queue and terminates. The DPUT job is thus executed.

The main service is continued when the message arrives. It reads the result from the temporary queue and sends it to the client. It then deletes the temporary queue and terminates (the deletion does not take effect until the successful completion of the transaction).

To control the sequences executed in the program units, an auxiliary field called `WaitForMsg` must be created in the KB communication area so that it can be accessed by all the service’s program units. If the return code 08Z is obtained repeatedly, superfluous waiting is avoided by means of a RSET call.

#### Program unit TAC1:

```
...
QCRE NN
FPUT with KCRN=search
kb.WaitForMsg = 0
PEND SP with TAC2
```

#### Program unit TAC2 for evaluating the result:

```
...
DGET queue
if ( KCRCCC = "08Z" )
then if kb.WaitForMsg = 0
      then kb.WaitForMsg = 1
           PEND PA,<tac2>
      else RSET
else processing
QREL queue
...
PEND FI
```

**Example 2: “Pseudo dialogs” with remote transport system applications**

Another application example is obtained by carrying out a simple modification of example 1: a dialog or asynchronous service wants read access to a database in another application.

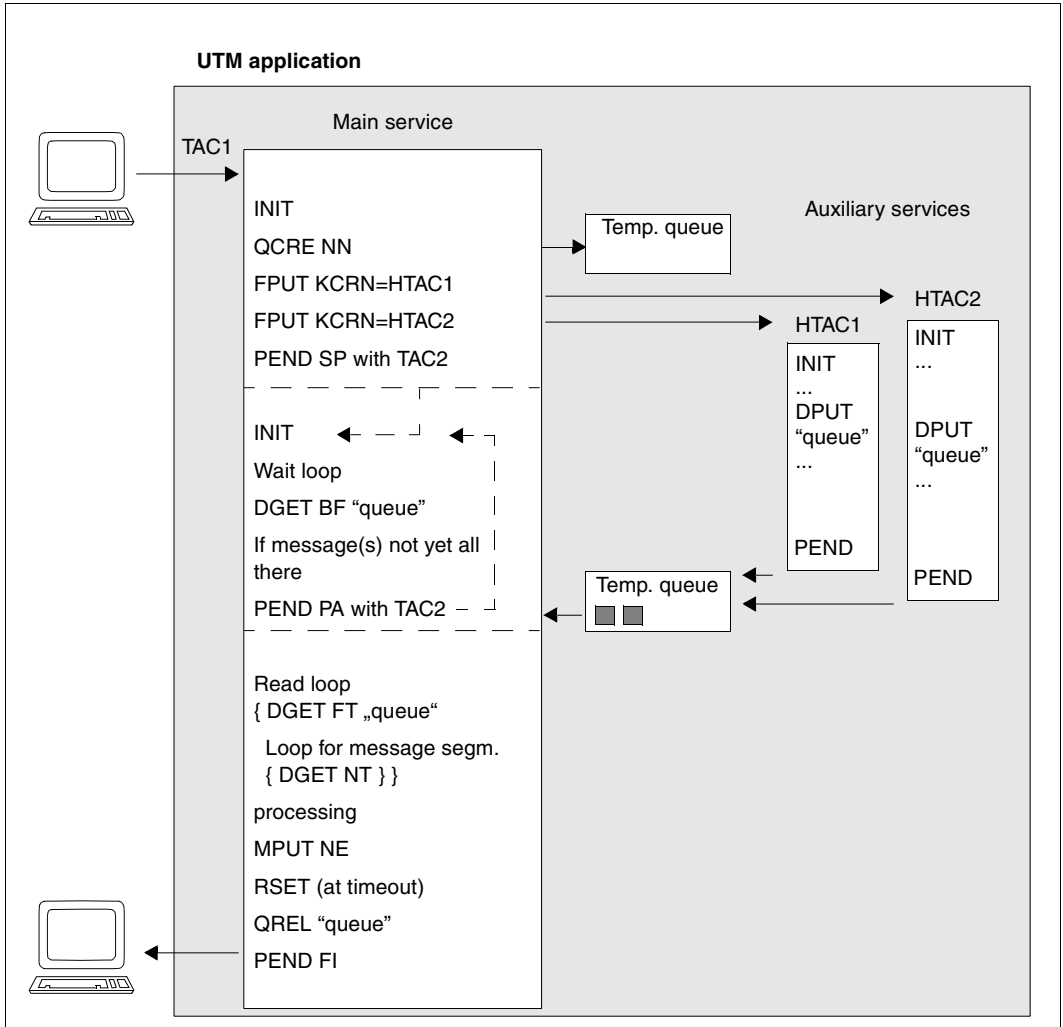
A communication partner of the type APPLI or socket replaces the auxiliary service of example 1. The FPUT call in the main service addresses the LTERM name of the remote application instead of an asynchronous TAC; apart from this change, the calls and processes involved in the two services remain unchanged. If the remote application is a UTM application, the TAC to be called in the remote application must be at the beginning of the FPUT message. The remote application must begin its reply message with the name of the queue in which the main service expects the reply.

openUTM determines the destination of the messages from the name of the queue alone. The name of the temporary queue must therefore not be the same as a TAC or LTERM name of the local application. Name clashes are avoided because the name of the temporary queue is assigned by openUTM (QCRE NN), and neither TAC nor LTERM names begin with a digit.

**Example 3: Communication with more than one service**

If a service wants to communicate with more than one auxiliary service simultaneously, what happens is similar to what happens in communication with only one auxiliary service. When the replies are read, however, it is important that all the replies are read; it may therefore be necessary to wait for all of these. You must ensure that each message is waited for no more than once.

To control the processes in the program units, auxiliary fields are created in the KB communication area in this example so that all the program units of the service can access them. As it is possible that not all the replies can be read in a program unit run, the browse function is used to implement a wait loop. The replies are not processed until the last message has arrived.



Communication of a main service with two auxiliary services by means of temporary queues

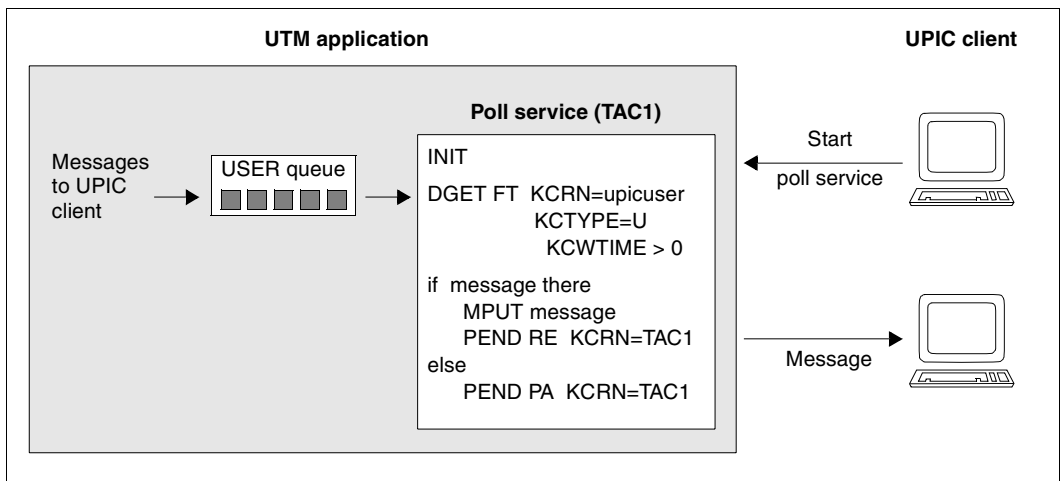
Auxiliary fields must be used for this communication, and loops must be programmed. The various steps are explained in the following table:

TAC1 program	Explanation
INIT	
QCRE NN	Create temporary queue. The name of the queue is returned in KCRQN.
FPUT KCRN=HTAC1 FPUT KCRN=HTAC2	Create messages for the auxiliary services. The name of the queue is passed in the message.
kb.NrMsgs = 2 kb.WaitForMsg = 0 kb.i = 0 kb.kcgtm = spaces kb.kcdpid = spaces	Initialize fields in the KB program area. These fields control the sequence of execution in the follow-up programs.
PEND SP KCRN=TAC2	Terminate transaction. The asynchronous jobs are started.
TAC2 program	Explanation
INIT	
Timeout = 0	Initialize timeout field.
for ( kb.i < kb.NrMsgs AND Timeout = 0 DGET BF with length 0 kb.kcgtm kb.kcdpid	In a loop an attempt is made using DGET BF with waiting to wait for all replies without reading or deleting data; in other words, all messages are retained. Make sure that each message is not waited for more than once.
if ( KCRCCC = "08Z" ) then if kb.WaitForMsg = 1 then Timeout = 1 else kb.WaitForMsg = 1 PEND PA KCRN=TAC2 else kb.i = kb.i + 1 kb.WaitForMsg = 0 kb.gtm = KCRGTM kb.kcdpid = KCRDPID	There must be a wait. The program run is then terminated with PEND PA; the current TAC is specified as the follow-up TAC.  The counter is incremented when a reply arrives. The creation time and DPUT ID of the message are buffered.
if Timeout = 0 then for ( i < kb.NrMsgs ) DGET FT for ( KCRCCC != "10Z" ) DGET NT processing MPUT NE	All replies are read in full in two nested loops. Processing of the replies then begins. The result is sent to the client using MPUT.
else RSET MPUT	At timeout, superfluous waiting is avoided by means of an RSET.
QREL "queue" PEND FI	Before the service terminates, the temporary queue is deleted again.

#### Example 4: Sending asynchronous messages to UPIC clients

Asynchronous messages must not be directed at UPIC LTERMs; with the help of service-controlled queues, however, asynchronous events or messages can also be delivered to a UPIC client. To do this, you either direct the asynchronous messages to the USER queue of the UPIC client or you generate a TAC queue additionally for each UPIC client. The UPIC client starts another dialog service in parallel with its normal dialog services. This dialog service assumes the task of waiting at this queue until a message arrives, reading it out and forwarding it to the UPIC client.

The following diagram shows how this is implemented using USER queues:



Sending of asynchronous messages to UPIC clients using USER queues

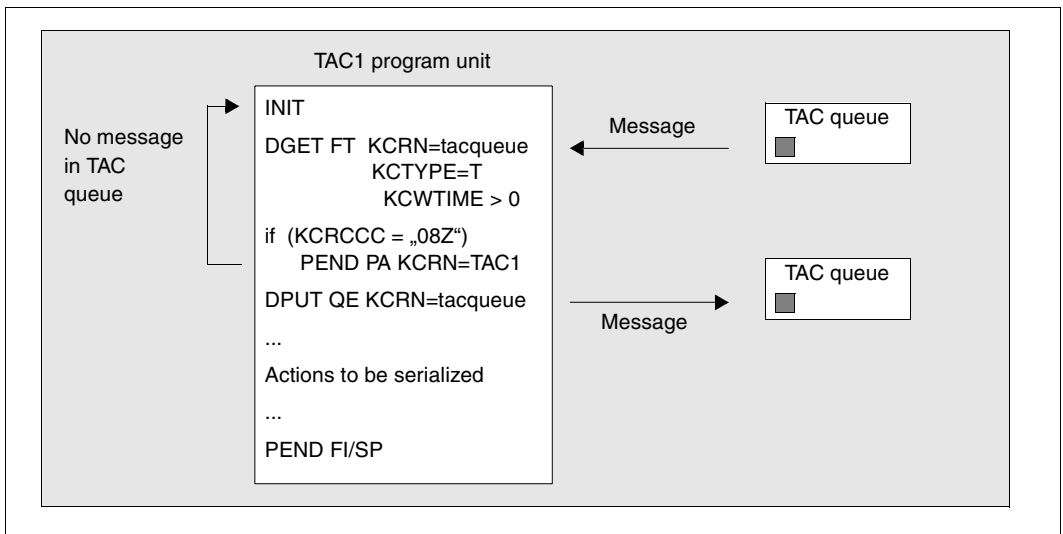
**Example 5: Serialization of program units**

If a program (segment) that is executed in both dialog and asynchronous services is to be serialized, you can use a TAC queue for serialization. TAC queues have an advantage over GSSBs (global secondary storage areas) in that waiting takes place outside the program unit context and the process is thus free for other things.

In the following example, a segment of a program unit that can run both in the dialog and asynchronously is secured against parallel processing by means of a TAC queue.

Generation:

```
TAC tacqueue,TYPE=0
```



Serialization of program units using TAC queues

A message is sent to the TAC queue by means of DPUT for the next service to execute the critical program segment. Only on successful completion of the transaction does the DPUT call take effect.

The mechanism described here must be initialized so that the critical program unit segment can be executed. This can be implemented at application startup, for example, by means of an MSGTAC program that responds to a start message (K050, K051) and creates a message for the TAC queue.



### Example 6: Output of other TLS blocks in the dialog service

Accesses to TLS blocks of a different data station are only permitted in asynchronous services. The output of data from other TLS blocks from a dialog service to the data station is possible if example 1 is slightly modified:

The dialog service passes to the auxiliary service the name of the TLS block and of the LTERM. The auxiliary service reads the required data with a GTDA call and sends it to the temporary queue.

The dialog service reads the TLS data from the temporary queue and outputs it at the data station.

### Example 7: Service-controlled queues as global storage areas

Only serial access to application-global storage areas (GSSBs, see [page 83](#)) is possible, i.e. areas of this kind are locked during read/write. If, instead of a GSSB, a temporary queue is used with the "browse" function, several services can then access the storage area in parallel.

Also, fewer restrictions apply to temporary queues than to GSSBs. The storage area of a temporary queue is not limited (GSSB: 32,767 bytes) and more queues than GSSBs are permitted (500,000 as opposed to 32,767).

Using a temporary queue it is possible, for example, to implement an **online auction**:

- DPUT QT/QE puts the descriptions of the objects into a temporary queue one after the other. A fixed name is given to the queue. Later additions can be made to the queue.
- DGET BF/BN allows those interested to read these descriptions; this can also be done in parallel by several services.
- DGET PF/PN processes a description and therefore removes it from the queue, e.g. once an object has been sold in the auction.
- The temporary queue is deleted with QREL at the end of the auction.

The number of messages in a temporary queue can be limited at generation time (KDCDEF QUEUE statement, QLEV operand). If only the latest messages are of interest, wrap mode can be selected (QUEUE statement, QMODE=WRAP-AROUND operand). Once the maximum number of messages has been reached, each new message causes the oldest message to be deleted.

## 2.5 KDCS storage areas in openUTM

Under openUTM, program units can use various storage areas for the reading and writing of user data. These storage areas ensure a clear distinction possible between program and data areas and ensure that programs are reentrant. Additionally, these areas also enable the transaction-logged, high-performance exchange of information between programs and guarantee the efficient usage of working memory. Some memory areas are specially designed for statistical purposes and for logging.

In addition to the storage areas shown here, openUTM also offers you the chance to use service-controlled queues for communication between program units (see [section “Messages to service-controlled queues” on page 61](#)).

The following storage areas are available:

- standard primary working area (SPAB)
- communication area (KB)
- local secondary storage area (LSSB)
- global secondary storage area (GSSB)
- terminal-specific long-term storage (TLS)
- user-specific long-term storage (ULS)
- user log file (for writing only)
- areas (AREA)

The number of GSSBs, LSSBs, TLS blocks and ULS blocks available in a UTM application is specified during generation; names are also assigned for the TLS and ULS blocks at this time; the names of the GSSBs and LSSBs are specified when programming the application.

### Storage location for user data

In the case of standalone applications, the user data in the KB, GSSBs, LSSBs, TLS and ULS blocks is stored in the KDCFILE.

In a UTM cluster application, each node application possesses a separate KDCFILE. The fact that some data is valid at local node level whereas other data is valid globally throughout the cluster results in a number of peculiarities compared to standalone applications:

- Data that is local to the node is stored only in the KDCFILE file of the relevant node application. Data that is local to the node includes, for example, TLS, asynchronous messages, background jobs and data relating to other node-bound services. The KDCFILES of the individual node applications are not identical at runtime.
- Data that applies globally in the cluster is stored in UTM cluster files such as the cluster page pool or the cluster user file, see the openUTM manual “Concepts und Functions”. This includes user data such as GSSB, ULS, the service data of users and passwords.

## Storage concept

The storage concept supports the following functions:

- reentrant capability of program units (SPAB)
- application data can be accessed by two or more services running in parallel (GSSB, TLS, AREA)
- protection of user-specific data against access by foreign login names (ULS)
- transaction-oriented storage of logging data (user log file)
- independence of program units running in parallel (KB, LSSB).

openUTM provides two types of storage area:

- working memory areas (KB, SPAB)  
These are storage areas which are available for the program units in the working memory. You can address them directly.
- secondary storage areas (GSSB, LSSB, TLS, ULS, log files)  
openUTM sets up these storage areas in background storage areas and uses special KDCS calls to write to and read from them.

Depending on their function these areas are assigned to:

- a program unit (SPAB)
- a service (KB and LSSB)
- an LTERM, LPAP or OSI-LPAP partner (TLS)
- a user ID, USER/ LSES or ASSOCIATION (ULS)
- the application (log file, AREAs, GSSB).

If a service is processed in two or more processing steps or in two or more program units, the data has to be stored and passed on. For this you use:

- the communication area (KB) and the
- local secondary storage area (LSSB).

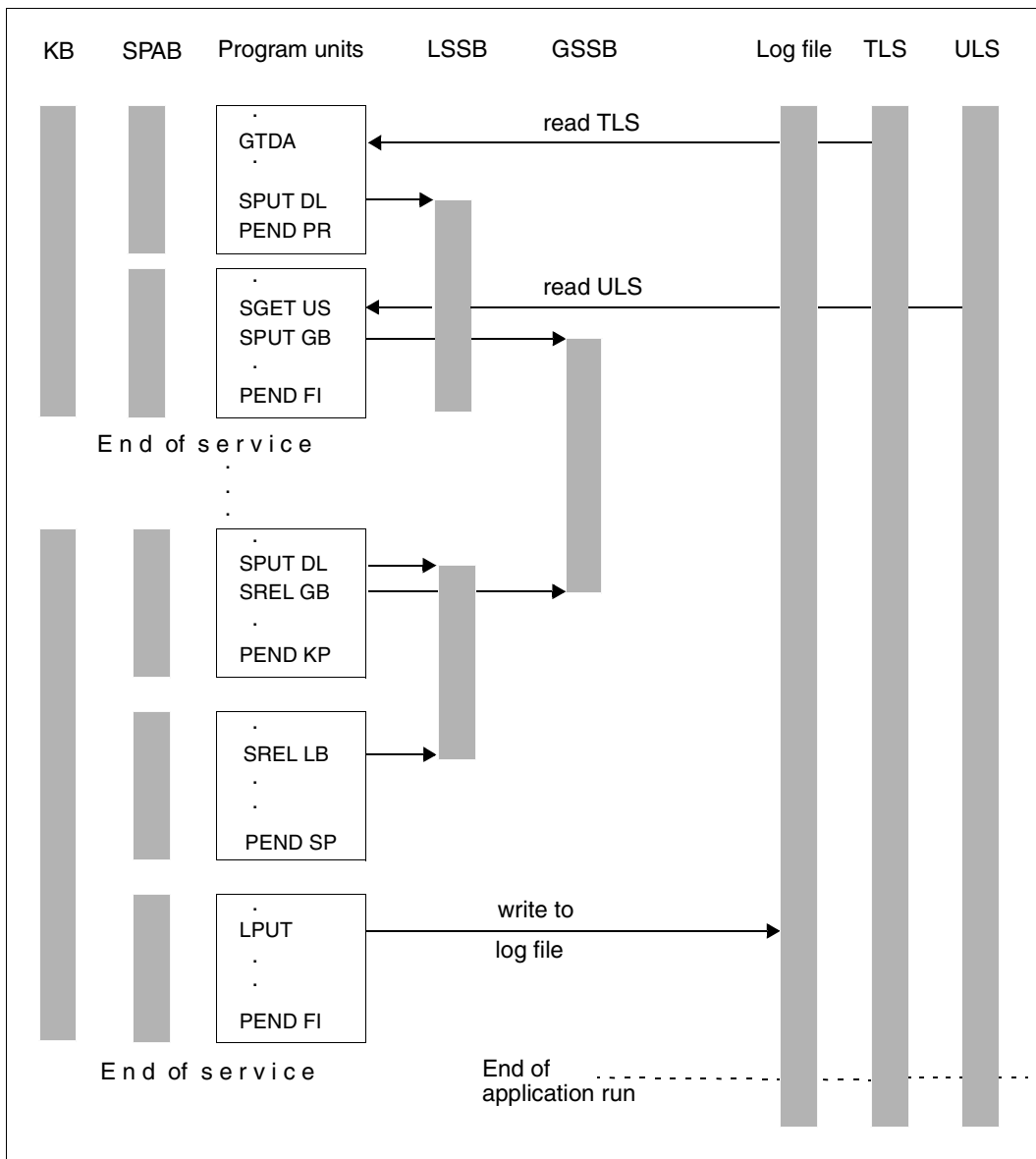
You should use the KB for data required in each processing step.

LSSBs are to be used either when the data is not required in each program unit run of a service or when there is more than 32 Kbytes of data, which means that it will not fit in the KB.

### Function and lifetime of the storage areas

The following table provides a summary of the lifetimes and functions of the KDCS storage areas. The diagram on the next page also shows the KDCS calls you can use for storage processing and illustrates the assignment of the individual storage areas.

Abbreviation	Area type	Lifetime	Function
KB	Communication area	Start of service to end of service	Accesses current information provided by openUTM; exchanges data between program units of a service
SPAB	Standard primary working area	Start of program unit to end of program unit	Transfers parameters for KDCS calls; message buffers
AREA	Additional storage area, declared for each generation	Duration of application	Accepts global data of an application; preferably used for read access only.
LSSB	Local secondary storage area	From first write call to explicit release or deletion of user ID	Exchanges data between the program units of a service
GSSB	Global secondary storage area	From first write call to explicit release or end of service	Exchanges data across services
ULS	User-specific long-term storage	Generation until generation modification	E.g. statistics which are specific to particular users (USER, LSES, ASSOCIATION)
TLS	Terminal-specific long-term storage	Generation until generation modification	Statistics which are specific to particular connection points (LTERMs, LPAPs, OSI LPAPs)
USLOG	User log file	Individually defined	Logging



Assignment of the KDCS storage areas and KCDS calls for storage processing

## 2.5.1 Standard primary working area (SPAB)

By default openUTM assigns a SPAB to each program unit, the maximum length of which is defined when the application is generated (see the openUTM manual "Generating Applications"). It is available to the program unit from the program start until the PEND call.

No data can be stored or transferred using this area beyond the end of the program unit run. Since the SPAB serves only as a program unit-specific working area, it is **not** included in the transaction and is not reset by a RSET call.

The SPAB may contain the following:

- the KDCS parameter area for call execution.

In the parameter area the program transfer the data which is necessary for a KDCS call. The entries in the fields of the KDCS parameter depend on the call in question (see [chapter "KDCS calls" on page 203ff](#)). Language-specific data structures are available to structure the parameter area: for COBOL in the KCPAC COPY element and for C/C++ in the *kcmac.h* include file.

- the message area (NB) for provision of the input/output data.

Messages read using MGET, DGET, FGET, SGET or GTDA are entered in the message area. Here, openUTM also presents information which you can request with any of the INIT, DADM, INFO and PADM call variants. When performing output, you also use the message area to make available the data transferred with MPUT, FPUT, DPUT, SPUT, PTDA or LPUT.

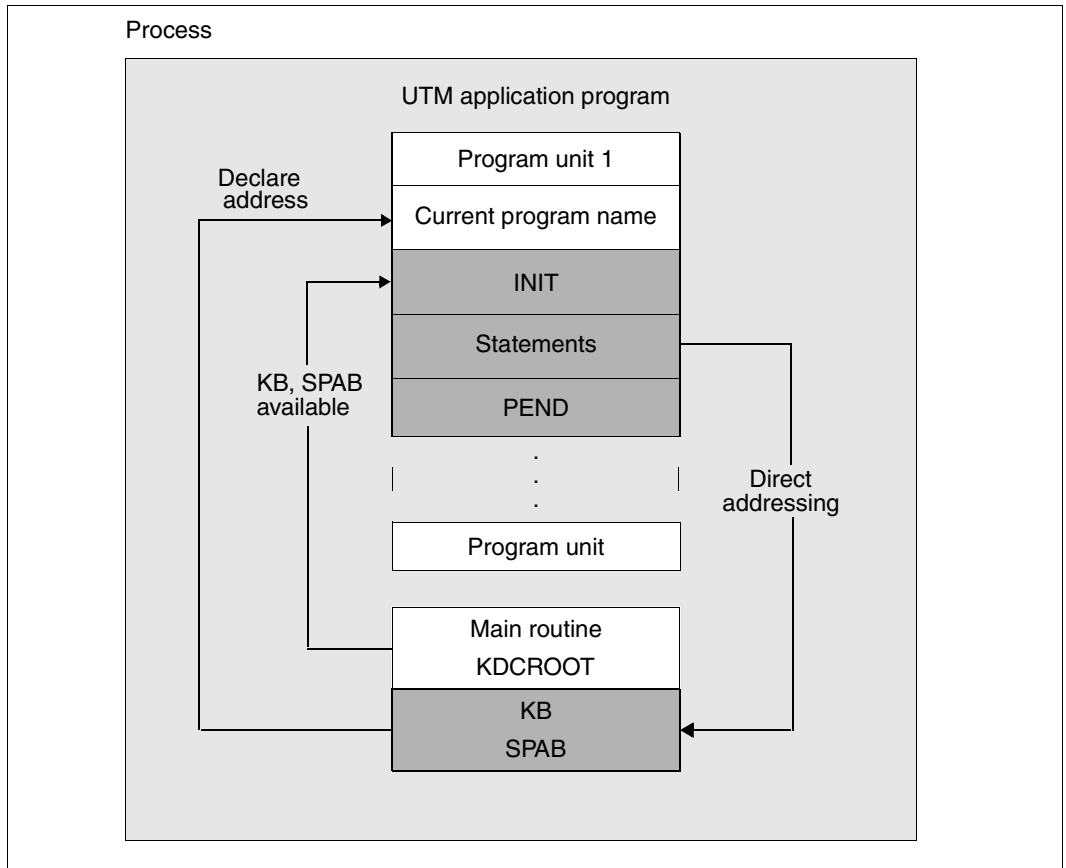
For each call you have to specify the address of the message area.

- further areas with variable data specific for the program run.

### Freely selectable fill characters

When you generate a UTM application you can specify a fill character of your choice (in the CLRCH operand of the KDCDEF statement MAX). openUTM then overwrites the SPAB with this character at the end of each processing step. After the start of the application program the area is filled with the specified character in the generated length.

This function is important for program testing because it facilitates the discovery of certain errors in single-process mode. Additionally, you can use this function for data protection.



Use of primary storage areas (SPAB, KB)

## 2.5.2 Communication area (KB)

openUTM creates the KB when a new service is started. The KB is retained until the service is terminated. The contents of the KB are transferred to the program currently being executed. The size of the KB can be adapted to the data to be transferred, i.e. it can be increased or decreased for individual processing steps.

There are two areas of fixed lengths at the beginning of each KB. These are used for communication between openUTM and the program:

- the KB header
- the KB return area.

Language-specific data structures are available for structuring these areas: for COBOL in the KCKBC COPY element, and for C/C++ in the *kcca.h* include file.

In the **KB header** you can find current information about the service, program unit and communication partner after calling INIT, and after each subsequent call.

In the **KB return area**, openUTM transfers its return data to the program after each call (except PEND). The evaluation, in particular of the return code, provides information about whether or not a call has been executed successfully and can be used for specific control measures in the program (see also [section “Programming error routines” on page 89](#)).

Additionally, you can define an area of variable length in the KB, the **KB program area**, whose structure you can determine freely. While the KB header and the KB return area are always present, you can select whether or not you want to use a KB program area. It can be used to transfer and store service-specific data of any type.

The maximum length of this area is specified when the application is generated (KB operand in the MAX statement). In the program, you use INIT to specify the length which the program currently expects. This length may not be longer than the value you generated with the KB operand. When the next synchronization point is reached the KB is saved in the length specified by INIT. Therefore only enter the absolutely necessary minimum length for the KB when calling INIT. In this way you avoid unnecessary information being saved.

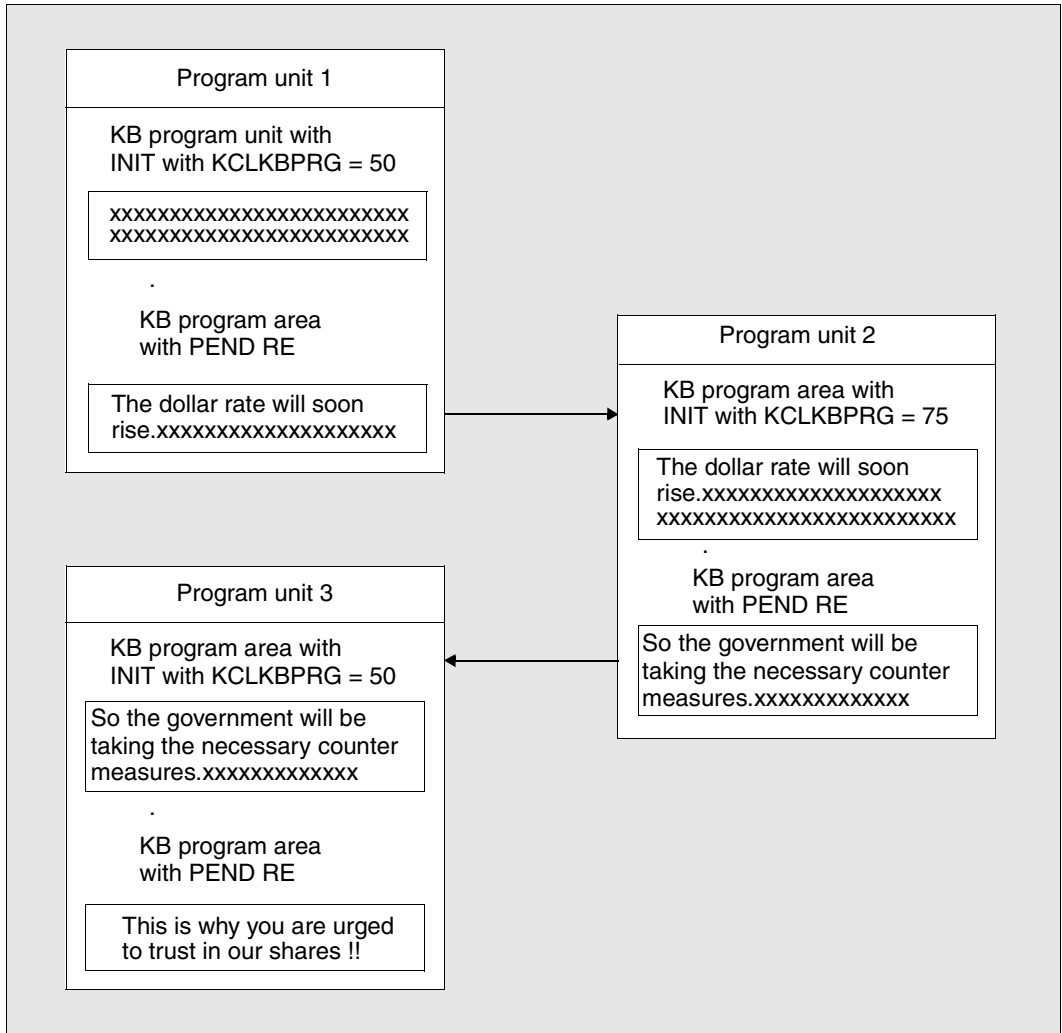
If multiple programs with KB program areas of differing length process the same service, length conflicts may occur during data transfer between programs. A program always receives data in the length specified in the KCLKBPRGE field by the predecessor program when INIT was called. If the receiving program provides longer data fields, then the remaining area is undefined. If the receiving program defines a shorter KB, it nevertheless receives the data in its full length. The truncation only takes effect when the data is passed on again (see diagram next page).



**Freely selectable fill characters**

When you generate a UTM application you can specify a fill character of your choice (in the CLRCH operand of the KDCDEF statement MAX). openUTM then overwrites the SPAB with this character at the end of the service. After the start of the application program the area is filled with the specified character in the generated length.

This function is important for program testing because it facilitates the discovery of certain errors in single-process mode; additionally, you can use this function for data protection.



KB program area for data transfer

### 2.5.3 Local secondary storage area (LSSB)

LSSBs are service-specific background storage areas which are used to forward data between program units within a service. Their contents are stored externally in the KDCFILE or, in the case of UTM cluster applications, in the cluster page pool, see [“Storage location for user data” on page 74](#). While openUTM automatically makes the communication area available for each program unit and then saves it, the LSSB is only accessed when necessary. It is therefore particularly useful for data which is accessed for reading only or if multiple program units are present which do not require the data between the point when the data is written and the time it is read.

For example, you can use LSSBs in the following cases:

- creation of a "browse function" for dialog outputs
- "temporary" data storage

You can use the SPUT call to write to a LSSB. To do this, enter the name of the area you want to write to. This name is only valid for the local service; if two services have the same name, then the names designate two different LSSBs. The LSSB is set up on the first write access within a service. Here the programmer defines the length of the LSSB. An LSSB can have a maximum length of 32767 bytes.

You use the SGET call to read from the area. If the storage is not required after the SGET call, it can be simultaneously released. You can also use the SREL call to release an LSSB. As soon as the associated services has terminated, any unreleased LSSBs are released.

The maximum number of LSSBs that you can create within a service is determined when the application is generated (MAX statement, LSSBS operand, see also the openUTM manual "Generating Applications").

## 2.5.4 Global secondary storage area (GSSB)

GSSBs are background storage areas just like LSSBs. The GSSB differs from the LSSB in its function, lifetime and application. The GSSB permits the transfer of data from one service to another. Data transfer is also possible after the end of an application in UTM-S, i.e. data remains available to the application after restart.

In UTM cluster applications, GSSBs are supported throughout the cluster. I.e. they are stored in the cluster page pool and are therefore available for all users in all node applications. This means that all the node applications have the same view of the data in the storage area, see also [“Storage location for user data” on page 74](#).

You can, for example, use a GSSP to transfer data between dialog and asynchronous services if you do not use service-controlled queues (see [page 61](#)).

It is open to all services. You must therefore define an unambiguous name in all the program units of an application. A GSSB is reserved for a service from an SPUT, SGET or SREL call until the end of the transaction. This prevents multiple accesses. A call from another service to the reserved GSSB waits until this is free again. openUTM locks the GSSB from the time of access to the end of a transaction (see also the section “Action with locked storage areas” on [page 88](#)).

You define the maximum wait period at generation time (KDCDEF statement MAX, operand RESWAIT). The UNLK call explicitly unlocks a GSSB, i.e. releases it prematurely for a waiting service, provided that the GSSB has only been read.

You call SPUT to create a GSSB (and its name).

The contents of a GSSB are maintained until it is released in a program unit by means of the SREL call. Its life is not limited. It is thus again made available to a UTM application after an interruption of operation provided that, in the case of a standalone application, the same KDCFILE or, in the case of a UTM cluster application, the same cluster page pool is used as before the interruption or if the user data has been transferred to a new KDCFILE or the new cluster page pool using the KDCUPD tool.

In a program unit, you can use the KDCADMI call KC\_GET\_OBJECT (object type: KC\_GSSB) to ascertain the names of all currently existing GSSBs.

You define the maximum number of GSSBs which can be generated when you generate the application (MAX statement, operand GSSBS=, see also the openUTM manual “Generating Applications”).

## 2.5.5 Terminal-specific long-term storage area (TLS)

The TLS is assigned to a connection point (LTERM, LPAP or OSI-LPAP partner) and is used to store information that is to be available independently of the life of the services and the operating time of the application. This long-term storage may consist of multiple blocks whose names you define in the TLS statements. A separate TLS statement is used to define each TLS block name. You can specify multiple TLS statements. The defined block names are the same for all LTERM/LPAP/OSI-LPAP partners. A particular block can be identified by its block name and the names of its LTERM/LPAP/OSI-LPAP partners (see the openUTM manual "Generating Applications").

TLSs are only supported locally in the nodes in UTM cluster applications, i.e. each node application has its own TLSs.

For example, you can use a TLS

- to generate statistics for an LTERM, LPAP or OSI-LPAP partner or
- to log access violations, for a example for an MSGTAC exit, see ["Example of a MSGTAC program unit" on page 475](#).

A program unit of a dialog service can only access blocks belonging to its "own" TLS, i.e. only the TLS of the LTERM, LPAP or OSI-LPAP partner via which the service was started.

A program unit run of an asynchronous service can read blocks from all the LTERM, LPAP or OSI-LPAP partners of the UTM application.

You use GTDA to read and PTDA to write a TLS block. openUTM locks a TLS block from the time of access to the end of the transaction. You can specify the maximum wait period at generation (KDCDEF statement MAX, Operand RESWAIT). If a TLS has only been read, you can use the UNLK call to *explicitly unlock the* TLS (see also the section "Action with locked storage areas" on [page 88](#)).

The TLS is used to store information that has to be available independently of the life of the services and the operating time of the application. It is thus again made available to the UTM application after an interruption of operation provided that the same KDCFILE as before is used or if the KDCUPD tool is used to transfer the user data to a new KDCFILE.

## 2.5.6 User-specific long-term storage area (ULS)

You can assign a user-specific long-term storage area (ULS) to the user IDs (USER/LSES/ASSOCIATION) when you configure the UTM application. This long-term storage may consist of multiple blocks whose names you define in the ULS statements. A separate ULS statement is used to define each ULS block name. The defined block names are the same for all user IDs. A particular block can be identified by its block name and the name of the associated user ID (see the openUTM manual "Generating Applications").

In UTM cluster applications, user-specific, long-term storage is supported throughout the cluster. I.e. the data is stored in the cluster page pool and is therefore available for all users in all node applications. This means that all the node applications have the same view of the data in the storage area, see also ["Storage location for user data" on page 74](#).

The ULS blocks defined for the application are also set up for LU6.1 sessions and OSI TP associations.

You can, for example, use the ULS to record statistics about user IDs.

Without administrator privileges, a program unit can only access the ULS blocks of the user ID under which it was started. To access ULS blocks of other user IDs, the program unit must have administrator privileges.

You use the SGET call to read and the SPUT call to write a ULS block. openUTM locks a ULS block from the time of access to the end of a transaction. You can specify the maximum wait period at generation (KDCDEF statement MAX, Operand RESWAIT). If a ULS block has only been read, you can use the UNLK call to unlock the block explicitly (see also [section "Action with locked storage areas \(TLS, ULS and GSSB\)" on page 88](#)).

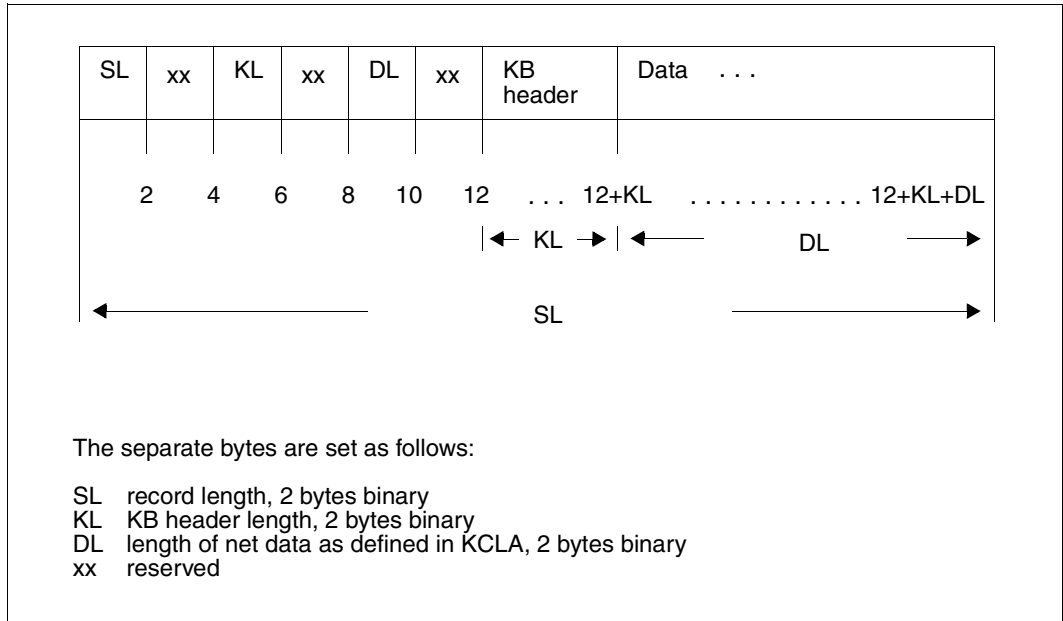
The ULS is used for storing information that has to be available independently of the life of the services and the operating time of the application. It is thus again made available to a UTM application after an interruption of operation, provided that, in the case of a standalone application, the same KDCFILE or, in the case of a UTM cluster application, the same cluster page pool is used as before the interruption or if the user data has been transferred to a new KDCFILE or the new cluster page pool using the KDCUPD tool.

### 2.5.7 User log file

The user log file (USLOG file) is used for recording user-specific data.

User log files are only supported locally in a node in a UTM cluster application, i.e. every node application writes to its own USLOG file.

It has the following record structure:



Record structure of the user log file

Each data record is generated with an LPUT call. openUTM additionally enters the contents of the KB header in the prefix of the data record. These contents are the same as at the time of the INIT call. You simply have to provide the data area containing the data which you want to log.

For more details on the user log file see the corresponding openUTM manual “Using openUTM Applications”.

## 2.5.8 Other areas

A program unit can use up to 99 other areas, which you define with the KDCDEF statement AREA (see the openUTM manual "Generating Applications"). AREAs are shared storage areas available to the UTM application. The structure of these areas is not predefined by openUTM and you can define them as you wish.

The addresses of such storage areas are passed along with SPAB and KB as additional parameters at the start of the program. These areas are **not** subject to the transaction concept: openUTM does not save these areas, nor does it reset them with an RSET or lock them. This means that the application program alone is responsible for managing these storage areas.

For information about how you define such an area in your programming language, refer to [page 489](#) for C/C++ and [page 546](#) for COBOL.



When program units which use AREAs are transferred from one application to another, the use of AREAs may cause problems, for example because of differences in parameters. Alternatives to the use of AREAs are described on [page 491](#) for the C/C++ programming language and [page 548](#) for COBOL.

## 2.5.9 Action with locked storage areas (TLS, ULS and GSSB)

If multiple UTM transactions attempt to access GSSB, TLS or ULS storage areas simultaneously, openUTM synchronizes these accesses.

Each access (for reading, writing or deleting) to a TLS, ULS or GSSB causes openUTM to lock this area for other transactions:

- until the next synchronization point (PGWT CM, PEND RE, FI, SP or FC) or
- until the next reset operation (PGWT RB, RSET, PEND RS or PEND ER/FR) or
- until release with the UNLK call if the area has only been read.

If a transaction attempts to access the locked area, openUTM places it in a queue until

- the lock has been released
- or the maximum wait period has elapsed.
- or the transaction responsible for the locking of the area uses PEND KP or PEND PA/PR with TAC class change or wait for DGET message or with PGWT KP/PR to set itself to a wait state of undefined length.

The process remains blocked and cannot take on other tasks during the wait period.

This maximum wait period is defined by the operand RESWAIT in the KDCDEF statement MAX (default: 60 seconds, see also the openUTM manual "Generating Applications"). If a transaction waits longer than the time defined by RESWAIT (*time1*), openUTM rejects the access attempt with the return codes KCRCCC = 40Z and KCRCDC = K810.

UTM rejects such an access attempt **immediately**

- if the area has been locked by another transaction which has just used PEND KP or PEND PA/PR with TAC class change or wait for DGET message or PGWT KP/PR to set itself to a wait state of undefined length (KCRCCC = 40Z and KCRCDC = K810), or
- if waiting would lead to a deadlock in GSSB, TLS and ULS areas (KCRCCC = 40Z and KCRCDC = K820). For this to be possible, deadlock detection must be explicitly enabled in cluster applications (see also KDCDEF statement CLUSTER DEADLOCK-PREVENTION in the openUTM manual "Generating Applications").

If a transaction accesses a GSSB, ULS or TLS for reading only, it is advisable to release this area again as quickly as possible - in particular before database calls or PEND PA/PR with TAC class change or wait for DGET message or before PGWT KP/PR - in order to shorten the waiting times of other transactions.

No such locks can occur during access to LSSBs, since it is not possible to access LSSBs across services.



## 2.6 Programming error routines

openUTM returns the following return codes in the return area of the communication area after KDCS calls: in the KCRCCC field, after each call a KDCS return code in accordance with DIN 66 265 (compatible return code) and possibly a UTM-specific return code in the KCRCDC field (KCRCDC). These return codes inform you whether openUTM was able to execute the KDCS call or why openUTM could not execute it.

For each KDCS call there is a description of the error codes that may occur with that call. For a list of error codes refer to the openUTM manual “Messages, Debugging and Diagnostics”.

If control is returned to the program unit after a call, it therefore has to check:

- whether the call was executed without errors; this is the case if the program unit recognizes that the KDCS return code has the value "000" (in the KCRCCC field).
- which errors have occurred on the basis of the specified error code.

Additional information that may be relevant for the measures which are to be implemented include the actual transferred length (KCRLM field) and the employed format character (KCRMF/kcrfn field) etc.

In the event of serious errors, the program unit does not get control back. In such cases, openUTM resets the transaction, terminates the service and creates a PEND ER dump; the error code that led to the abortion of the service can be obtained from the UTM-DIAGAREA of the PEND ER dump. You will find more information on the analysis of a PEND ER dump in the openUTM manual “Messages, Debugging and Diagnostics”.



Control is not returned to the program unit after a PEND call. The return area cannot therefore be subsequently evaluated in the program unit. If errors occur during PEND processing, UTM responds as described above.

The program unit can use RSET, PGWT RB or PEND RS/ER/FR to reset the transaction: a PEND ER causes a dump to be requested.

It might also happen that openUTM returns an error message because it has received an error message from one of the components involved (e.g. the formatting system).

In the case of such errors openUTM executes an internal error handling routine, thus relieving the program unit of this task. Where possible, openUTM corrects the error, otherwise the service is aborted by PEND ER and the appropriate UTM error code is set.

For examples of error routines, refer to the sample programs, for C/C++ [page 510ff](#), for COBOL [page 563ff](#).

## 2.7 Message segments

The KDCS interface enables you to process message segments. In this way, you can combine message components in order to forward them to the communication partner as a complete message.

The use of message segments has the following advantages:

- message areas only have to be as large as the largest message segment
- message segments can be processed separately
- formats can be made up of partial formats.

You use the MPUT NT, FPUT NT, DPUT NT or DPUT QT call to send message segments to openUTM.

Message segments may be addressed to a terminal, a client program, a queue, a printer, a follow-up program unit, a local asynchronous service or, with distributed processing, a remote partner.

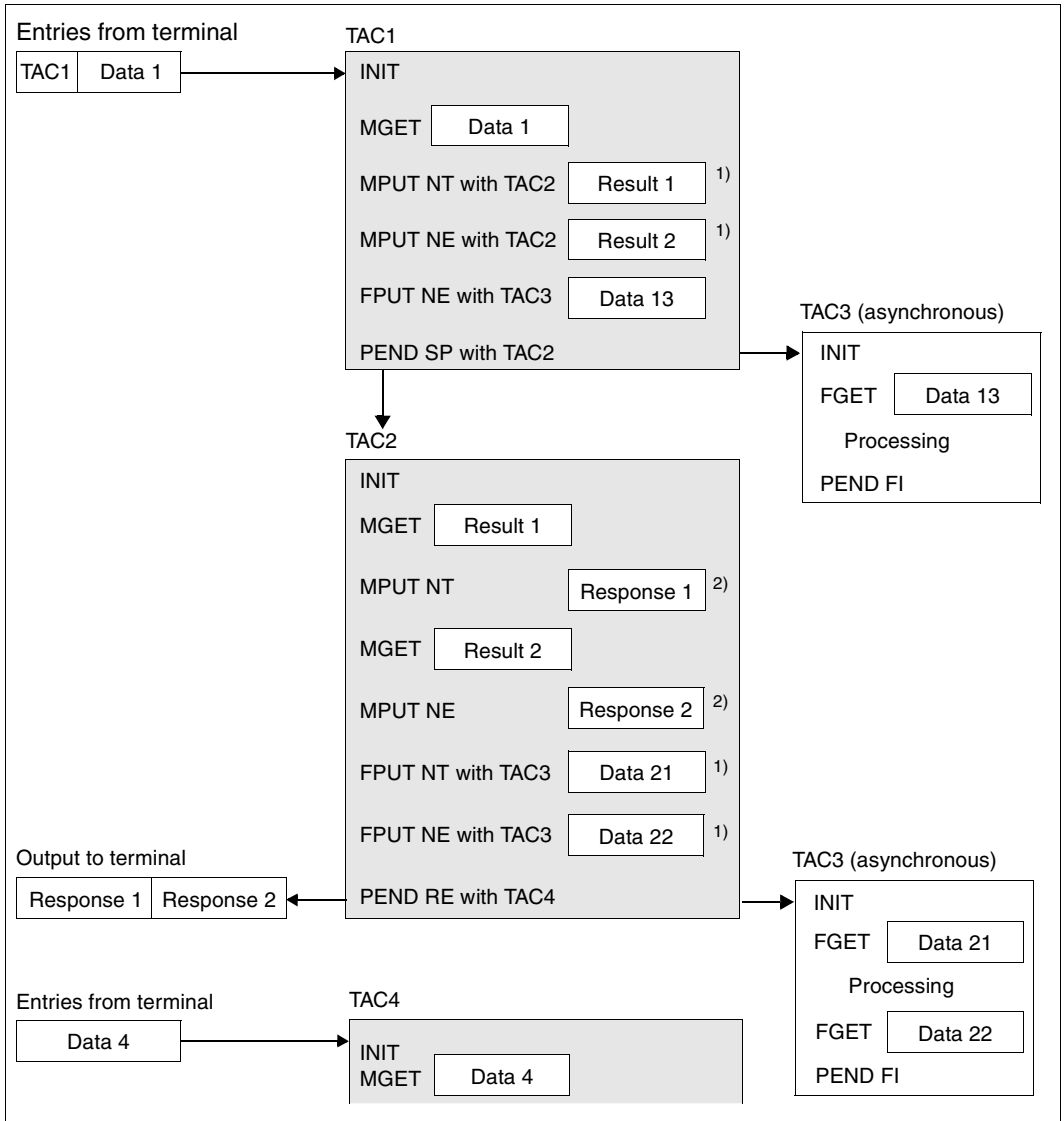
In all cases where multiple message segments are addressed to a program unit, you must read each of these message segments with a separate DGET, MGET or FGET.

If you use formats, you can send formatted message segments to terminals and printers. In the case of dialog messages to terminals, you must use partial formats (see [page 113](#)). In the case of formatted asynchronous messages to terminals, each message segment is treated as a separate message. In the case of formatted asynchronous messages to printers, the message segments are combined to form a logical unit.

Messages with format identifiers can also be sent to UPIC clients (starting with openUTM-Client V4.0) as well as to LU6.1 partners. Formats are not used to prepare the messages for these partners, but more precisely to describe the structure of the user data. openUTM does not call a formatting routine when sending formatted messages to these partners, rather the format name is sent to the UPIC client or LU6.1 partner instead.

### **Note on the diagram below**

If, in the first program unit (TAC1), FPUT is written as FPUT NT, it will again produce an asynchronous message of its own since openUTM terminated the message with PEND PR.



Message segments

- 1) If message segments are sent from one program unit to another, each message segment must be read with its own MGET NT or FGET.
- 2) Message segments to a terminal are combined by openUTM into a single message.

## 2.8 Communication partners of a UTM application

A UTM application can communicate with a variety of partners. Communication partners of an openUTM application are for example:

- Terminals

The KDCS interface enables you to connect character-oriented terminals directly. You can use these terminals in line mode or you can use screen forms (masks).

- Client programs

These can be, for example, UPIC clients, but they may also be clients which use the transport system interface directly.

- Transport system partners

Transport system partners are applications that use the transport system interface directly to communicate with the UTM application, e.g. sockets. These can also be other UTM applications.

- Printers

You can create output jobs for printers from openUTM services (see [section “Programming error routines” on page 89](#)). Print jobs are also part of the transaction concept.

- OSI TP, LU6.2 or LU6.1 partners

The partner may be another openUTM or OpenCPIC application, or it may be an application that uses a transaction system from a different manufacturer.

- Message queues

In UTM services, messages can be written to message queues, from where they can be retrieved by the recipient at any time. These queues are also included in the transaction concept.

You use the same KDCS calls for communication with all these partners:

MPUT	to send a dialog message to the partner
FPUT/DPUT	to send an asynchronous message to the partner or a message queue
MGET	to receive a dialog message from the partner
DGET	to read an asynchronous message from a message queue
FGET	to receive an asynchronous message from the partner

The following calls can also be used for communication within a UTM application:

- You can use MPUT to send a message to a follow-up program unit which then can use MGET to read this message. However, you can also use service-specific storage areas to exchange information between program units of a service (KB and LSSB, [page 74](#)).
- You can use FPUT/DPUT to send asynchronous messages to local asynchronous services or message queues. The messages can then be read with FGET/DGET.

## 2.9 Output to printers

You can output to printers in two different ways:

- hardcopy to a central or local printer
- output jobs to printers (print jobs), also called spooling

**W** Transaction-oriented output to printers is not supported under Windows systems.

### 2.9.1 Hardcopy mode with openUTM

**B** You can activate automatic hardcopy mode using the screen function KCREPR in the KCDF  
**B** field of the KDCS parameter area. If you use #formats control is performed via global  
**B** attributes.

**B** In BS2000/OSD, you can also activate the hardcopy function using edit profiles (specifica-  
**B** tions in KCMF/kcfn). To do this, you have to set the operand HCOPY=Y in the KDCDEF  
**B** statement EDIT when you define the edit profile (see the openUTM manual "Generating  
**B** Applications").

**B** In hardcopy mode you have to use MPUT, FPUT or DPUT to send the message to the  
**B** terminal. This initiates the printout, i.e. if you use MPUT, blanks are entered in KCRN and  
**B** if you use FPUT/DPUT, the LTERM name of the terminal is used.

**B** The terminal user can also press the key **LA1** to print the current screen contents to the  
**B** selected, assigned printer.

## 2.9.2 Print jobs

You use the FPUT or DPUT message queuing calls to create print jobs (see [section “Messages to UTM-controlled queues” on page 51](#) and [chapter “KDCS calls” on page 203](#)). Here you have to specify the LTERM name of the printer in the KCRN field. openUTM enters the job in the corresponding queue.

You can

- use a single FPUT or DPUT call (NE in KCOM field) to generate a print job or
- construct print jobs using multiple FPUT or DPUT calls (for message segments): NT in KCOM field and NE for the last FPUT or DPUT call.

openUTM handles the message segments of a print job as one entity when determining print sequences or for error handling. With printer pools, openUTM sends all the message segments of a job to **one** printer.

You can either print out in line mode.

**B** In BS2000 you can print out in format mode.

**B** In BS2000/OSD it is also possible with RSO printers to pass a parameter list to RSO using FPUT RP or DPUT RP. RSO changes the parameter settings depending on the particular printer type.

### Administering message queues and printer control

You use the KDCS call DADM (see [page 224ff](#)) to administer the UTM-controlled message queues which contain the print jobs and are assigned to the LTERM partner.

With the DADM (delayed free message administration) call, you can:

- read information about jobs of a message queue into the message area
- change the processing sequence of jobs of a message queue
- delete individual jobs or an entire message queue
- move defective messages from the dead letter queue

You can use the KDCS call PADM for printer control.

With PADM (printer administration) you can:

- activate or deactivate the confirmation mode for a printer control LTERM
- confirm or repeat a print out
- modify the assignment of the printer to an LTERM partner
- change the printer status, i.e. lock and release the printer, establish or cancel connection to a printer
- read information about a printer into the message area
- read information about print outs to be confirmed

For detailed information about the administration of message queues, the execution of print outs and printer control options refer to the openUTM manual "Administering Applications".

### Avoiding bottlenecks during high volume printing

The message queues are stored in the Pagepool of the KDCFILE. To avoid bottleneck situations when printing high volumes and to avoid a Pagepool overflow, you can make the following provisions in the program unit or at generation.

- In the program unit:
  - Evaluate the return code 40Z/K701 after FPUT/DPUT calls to printers or the "K041 warning level # for PAGEPOOL exceeded" message in an MSGTAC program unit and, for example:
    - establish a connection to the printer for which there are many messages available so that the messages can be printed and then deleted by openUTM
    - or delete FPUT via the DADM administration, or block TACs that send FPUTs to the printer.
- For each generation:
  - use QLEV= (LTERM statement) to establish the maximum number of messages which openUTM can temporarily store in a message queue. Print jobs are not taken into account until end of transaction. openUTM rejects further messages to this printer with 40Z.
  - use QAMSG= (LTERM statement) to determine whether messages to a printer which is not connected to the application are to be buffered.
  - estimate the necessary, additional space for printer messages in Pagepool and enter a sufficiently high value in the PGPOOL operand of the MAX statement, for recommendations see the openUTM manual "Generating Applications".



The output of UTM message K022 at a printer always results in a separate print job.



### 2.9.3 Output in line mode

If you want to output in line mode, the first byte in the KCMF/kcfn field of the KDCS parameter area must be a blank.

In line mode, the message may contain all logical control characters, e.g. for form feed or line feed. You can define these control characters yourself in the program unit and in this way structure the text for the print-out yourself.

B In BS2000/OSD you can also use edit profiles to present/prepare the messages (see also  
B [section “Controlling the output in line mode \(BS2000/OSD\)” on page 118](#)). You enter the  
B name of the edit profile in KCMF/kcfn starting at the second byte.



A message terminated by FPUT/DPUT NE is always printed as a separate message. In the case of cut sheet printers, the program must allow users to change sheets themselves for output in line mode (form feed) since the device cannot recognize the end of sheet itself.

### 2.9.4 Outputs in format mode

B In format mode, openUTM interoperates with a formatting system. Refer to the manual for  
B your formatting system for information about the supported printer functions.

B In the KCMF/kcfn field of the KDCS parameter area, you enter for the FPUT/DPUT call:

B  $\left. \begin{array}{l} * \\ + \\ \# \end{array} \right\} \text{format\_name}$   
B  
B

B where:

B \* Formatting without the option of changing the attribute

B + Formatting with the option of changing the attributes of individual format fields

B # Formatting with the option of both changing the characteristics of individual format  
B fields and changing the global characteristics of the format fields

B In BS2000/OSD the formatting system must support the printer models as they are known  
B in the VTSU-B or as they were generated for RSO (for PTYPE=\*RSO).

- B If you use the FORMAT event exit for printouts, you have to consider the following:
- B – you do not have to set the restart area
- B – you have to request the print out confirmation in the message header
- B – you have to enter the confirmation bytes transferred by openUTM in the message header
- B – you ascertain the formatting specifications, such as format name, device type, etc. from the KB header and the parameter area of the FORMAT event service.
- B For the structure of the message header, refer to your printer manual.

### B **Form feed in BS2000**

- B In BS2000/OSD, whenever FPUT/DPUT is called, whether or not a form feed is performed depends on the contents of the KCDF field in the KDCS parameter area (exception: when #Formats are used):
- B KCREPL            Form feed before printing, i.e. the format is printed starting at the basic form setting specified at the printer.
- B other             No form feed, i.e. the format is printed starting at the next line (also for the first FPUT/DPUT).
- B This allows you to concatenate formats in order to print large forms, independent of any length limitations. However, the physical message segment must be shorter than:
  - B – the BCAM letter length (see TRANSDATA manual "Generating a Data Communication System")
  - B – the TRMSGLTH specification (MAX statement for KDCDEF)
  - B – the device-specific length that is supplied by VTSU-B when a connection is established.
- B The length is only limited by TRMSGLTH for RSO printers (PTYPE=\*RSO).

## 2.10 Support for ID card readers

B UTM supports ID card readers at data display terminals.

B You can use the ID card reader in two ways:

- B – access protection through ID card control at sign-on
- B – data input via ID card

B You cannot use the ID card reader both ways simultaneously: after signing on with the ID card reader, the terminal user cannot then use the ID card to enter data.

### 2.10.1 Signing on to the application via ID card reader

B You can configure user IDs for a UTM application in such a way that access to the application via a user ID is only possible with a special ID card (magnetic strip card): This can be done either via the KDCDEF statement USER Operand CARD at generation or through dynamic configuration (KC\_CREATE\_OBJECT, object type KC\_USER).

B If a user signs on via a user ID for which an ID card check is configured, the user is requested to insert the ID card in the reader. openUTM checks the ID card information. If this information agrees with the information generated in the configuration for this user ID, the user is allowed to work with this application  
B If an ID card is in the reader during dialog input, openUTM enters the identifier "A" in the KCAUSWEIS/kccard field of the KB header.

B The user must leave the ID card in the reader until he or she has signed off from the application with KDCCOFF, otherwise openUTM disconnects the connection to the terminal.

B You can call the ID card information in the program units, for example using the KDCS call INFO with the "CD" (CARD) operation modifier.

## 2.10.2 Data input via ID card

B If you do not use the card reader for sign-on check, you can enter data via the card reader  
B in the following way in BS2000:

B If you use the screen function KCCARD or an edit profile with SPECIN=I to output a dialog  
B message (MPUT), the keyboard is locked and the user is requested to insert the magnetic  
B strip card into the ID card reader (this should be made clear in the text for the dialog  
B message).

B The follow-up program unit can then use MGET to read the data of the magnetic strip card  
B like a normal screen input.

B If no magnetic strip card is available to the user, he or she can unlock the keyboard by  
B pressing either the `[K14]` key (or `[ESC]` together with `[.]`). In this case, when MGET is called,  
B the follow-up program receives the return code which you assigned to the `[K14]` key with the  
B KDCDEF statement SFUNC during generation.

B If you remove the magnetic strip card from the card reader, openUTM repeats the last dialog  
B output (internal KDCDISP).

B *Checking availability*

B – You can check whether the ID card is available in the KCAUSWEIS/kccard field in the  
B KB header. In this case openUTM sets the character "A" if the ID card was inserted at  
B the last input.

B – You can check the availability of an IID card reader by means of an INFO CK call: the  
B return code of the INFO CK call tells you whether or not there is an ID card reader at  
B the terminal. It makes sense to call INFO CK *before* an MPUT call with KCCARD. In the  
B case of an MPUT with KCCARD to a terminal which does not have a card reader, UTM  
B terminates the service abnormally.

---

## 3 Interaction with databases

openUTM supports coordinated interaction with database systems. openUTM uses **two-phase-commits** to synchronize the UTM transactions and the transactions of the database system (see the openUTM manual “Concepts und Functions”).

A UTM application can interact in coordination with multiple database systems. This means that a UTM transaction can contain calls to **different** database systems. KDCS calls to terminate a dialog step (PEND KP or PEND PA/PR) within a transaction usually result in a change of task or process. In OSI TP applications with Commit, this also applies to PEND SP, RE, FI, PGWT CM calls in the job-submitting service. If it is necessary to work together with one or more database systems in this type of environment then the task change must be supported by each of these.

Furthermore, it is possible in distributed processing to process data from a number of different database systems on different computers in a single transaction.

### Resetting transactions

If an errors occurs, openUTM resets all the databases of an transaction to a common synchronization point. There is no need for the programmer to coordinate openUTM and the database systems.

The database system can also reset DB transactions itself, e.g. to release long-term locks. In this case the transactions are again synchronized and the program unit is informed by the corresponding DB system return code.

### Internal interface between openUTM and database systems

openUTM employs a uniform and neutral interface to control the interaction with database systems. In this way openUTM is independent of any implementation-specific features of the various database systems.

Under Unix systems and under Windows systems, this is the XA interface standardized by X/Open, in BS2000/OSD the XA interface is offered and the functionally equivalent IUTMDB interface.

## Supported database systems

- B** openUTM under BS2000/OSD supports coordination with the following database systems:
- B** – UDS/SQL
  - B** – SESAM/SQL
  - B** – Oracle
  - B** – LEASY (the LEASY file system behaves like a database system with openUTM). Because it is not possible to use LEASY from multiple computers, LEASY cannot be used in UTM cluster applications.
- X/W** Under Unix systems and Windows systems openUTM supports coordination with the following database systems:
- X/W** – Oracle
  - X/W** – INFORMIX

## Connecting openUTM with database systems

The database systems with which a UTM application is to coordinate and interact are specified in the KDCDEF statement during generation of the UTM application: under Unix systems and Windows systems you use the RMXA statement, for BS2000/OSD the DATABASE statement (see the openUTM manual "Generating Applications").

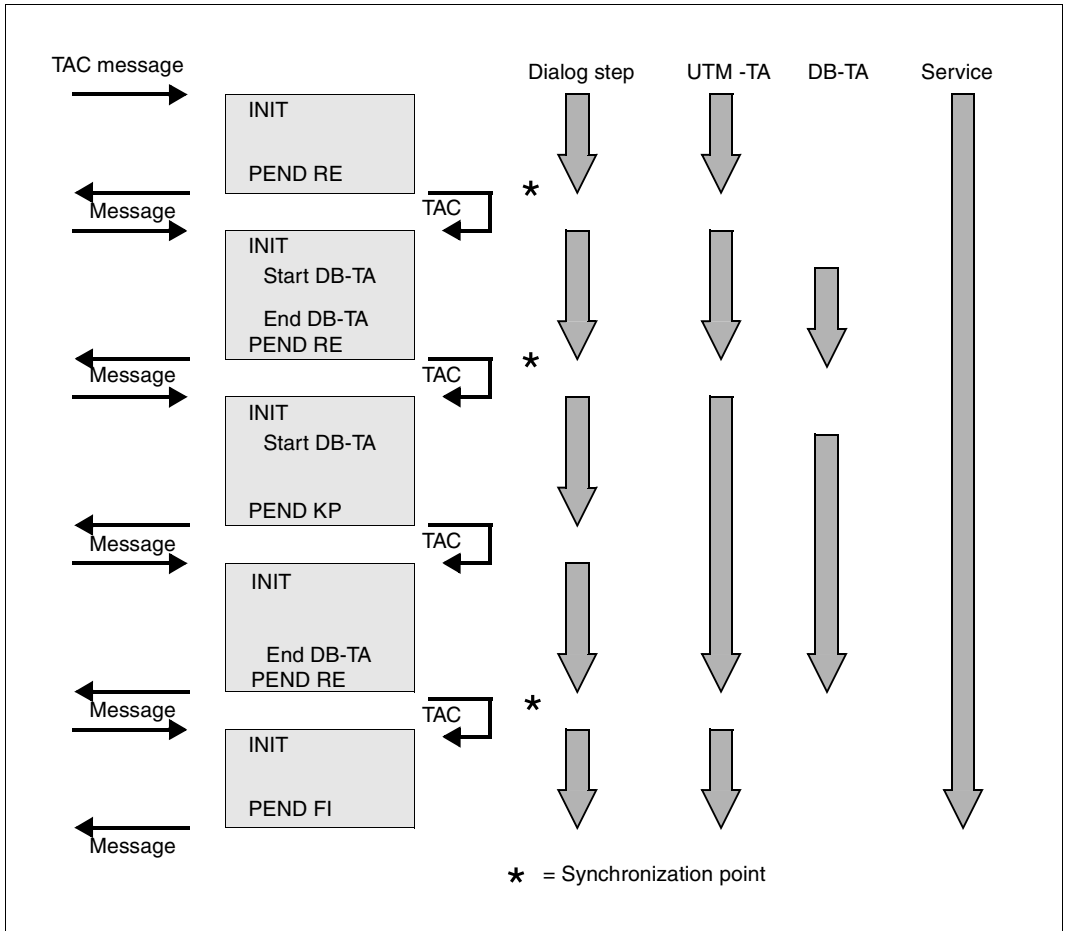
- B** *Note for BS2000/OSD*
- B** Some database systems provide various call interfaces for the application program. These may be implemented as CALL interfaces or language elements in the programming language (e.g. in COBOL). You use the KDCDEF statement DATABASE to specify the interfaces which the program units of a UTM application use for communication.
- X/W** *Note for Unix systems and Windows systems and for XA-capable databases under BS2000/OSD*
- X/W** No interface is specified for the RMXA statement for the application program. The interface depends on the resource manager used.

### *Note on using XA*

As a rule, there is a static and dynamic XA switch. A database can provide one or both variants. If the database provides a dynamic XA switch, you should use this. This minimizes the resources occupied in the database system.

### 3.1 UTM transaction and DB transaction

A service is constructed of one or more UTM transactions. The structure is determined by the program units of the service. A program unit opens a DB transaction when required, i.e. if it is necessary to read or modify the database.



UTM transactions and DB transactions in a service



With regard to database transactions the PGWT CM call behaves in the same way as PEND SP or PEND RE.

### **Multi-step transactions**

A multi-step transaction consists of several UTM dialog steps within a transaction. It is often impossible to modify the data records of a dialog in a single processing step. You need a first processing step to display the data and a second to enter the modifications. By locking the data records for both processing steps you can extend the lock to a multi-step transaction.

The time between the processing steps is monitored by a timer which you specify with the KDCDEF generation tool at generation. After the timer has expired the transaction is reset and the locked data released.

The database systems also manage the locking period themselves and reset the DB transactions to release the locks. The service is informed at the next DB call.

When using multi-step transactions you should note that each reset may result in the repetition of multiple processing steps, depending on how recent the last synchronization point is.



## 3.2 Programming ESQL program units

A UTM-ESQL program unit is structured like a normal UTM program unit.

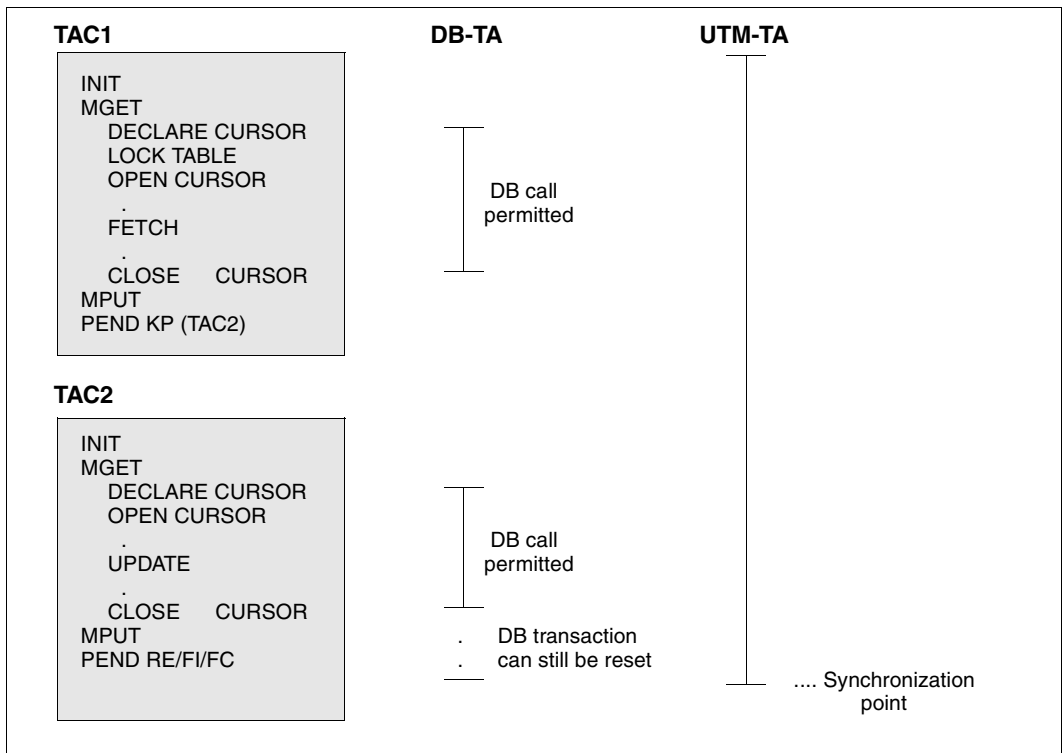
To communicate with databases you can use all the facilities of the ESQL interface.

However, for transaction logging you may only use KDCS calls.

Do not use ESQL calls such as `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` as this may result in errors. For further details refer to the manual for your database system.

### Note

Currently, Oracle and INFORMIX cannot migrate open cursors when connected via XA, i.e. you have to open them explicitly at the start of every program unit and close them explicitly at the end of every program unit. If you require this position in a follow-up program unit, you must program your application to save its value (e.g. in the KB or LSSB) and then set it in the follow-up program unit.



Example for use of `CLOSE CURSOR`

### 3.3 Error processing with connected databases

If errors occur in the interaction with database systems, openUTM processes the errors - the programmer does not need to take any special precautions.

The database systems informs openUTM of whether a call has been executed successfully. If an error has occurred openUTM checks the severity of the error and reacts accordingly: openUTM resets the transaction to the last synchronization point and issues messages explaining the causes of the error. If a serious error occurs as a result of interaction with the database openUTM generates the message K071. This message contains database status information and a specific return code, see the openUTM manual “Messages, Debugging and Diagnostics”.

openUTM uses XA messages (K201 - K232) to provide additional information about the status of the connection, irregular rollbacks, commits and errors when using XA calls (see also the openUTM manual “Messages, Debugging and Diagnostics”).

A special area for further diagnosis is available in the UTM dump, the UTM-DIAGAREA. openUTM enters trace information in this area. Alongside the KDCS calls, this area also logs all the calls to the database system. These calls can be seen after the DBCL string (see the openUTM manual “Messages, Debugging and Diagnostics”).

#### **Behavior after a system crash**

After an operating system crash and abortion of the UTM application, the database should be started before the UTM application at restart. If the UTM application was generated for the UTM-S variant (default setting), then openUTM executes a joint recovery phase with the database system/database systems at the start of the application.

---

## 4 Using formats

A format (also called a mask) is simply a form displayed at a terminal or output to a printer. Like any other form, a format consists of fields which you can fill in (input fields) and texts belonging to the form (text fields). This means that formats are forms which are stored in a computer and output at a terminal or printer when necessary (see also [section “Output to printers” on page 94](#)).

A format also contains information on how a field is represented on the screen (e.g. flashing), what you can enter in a field (e.g. numeric values only), or where the cursor should be found in the format display.

X/W  
X/W

A formatting system is not supported in Unix systems and Windows systems, although format names can still be sent to UPIC clients and LU6.1 partners.

### 4.1 Format names for message exchange with UPIC clients

In order to simplify programming, openUTM enters the name of the format to be used in the KCRMF/kcrfn field when INIT is called.

If a format consists of multiple partial formats you have to read each partial format with a separate MGET. Here the preceding MGET provides the format identifier in KCRMF/kcrfn. This format identifier must be entered in KCMF/kcfn in the following MGET call.

#### Exchanging messages with UPIC clients

Program units that are written for the formatted dialog with terminals can also be used for the dialog with UPIC clients without any changes. In this case, however, a formatting system is not called. Net messages or message segments and format names are exchanged between openUTM and the UPIC client. openUTM functions in the same manner when reading the message as when reading segment formats, i.e. if an incorrect format identifier was specified for MGET, then openUTM sets the return code to 03Z, returns the correct format identifier in KCRMF/kcrfn and sets KCRLM to 0. Nothing is entered in the message area.

The UPIC client can then set up the screen interface based, for example, on the format name. The UPIC version used must be  $\geq 4.0$  in order for this exchange to function correctly.

It is also possible to position the cursor in a program using "KDCSCUR" when outputting a message to the UPIC client. The UPIC version used must be  $\geq 5.0$  in order for this exchange to function correctly.

A UPIC client can pass the value of a function key to a UTM application so that the return code generated for this function key is triggered when an MGET is executed in the program unit (see the SFUNC statement generation, RET parameter). The UPIC version used must be  $\geq 4.0$  in order for this exchange to function correctly.

See the manual "openUTM Client for the UPIC Carrier System" for more detailed information.

## 4.2 Use of formats in openUTM in BS2000/OSD

B In BS2000/OSD, you can also use line mode to affect the output on the screen. As this also  
B includes the optical output of messages these options are also illustrated in this chapter (in  
B [section "Controlling the output in line mode \(BS2000/OSD\)" on page 117](#)) - although, strictly  
B speaking, this is not a format function.

### B Interaction with formatting systems

B In BS2000/OSD, openUTM supports the format handler FHS (format handling system) and  
B the format generator IFG (interactive format generator).

B You can use the format generator to create formats simply and quickly within a guided  
B dialog. Addressing aids are automatically generated which you can use in the program units  
B to structure the message areas. The finished formats are stored in under in libraries.

B If you want to perform input/output in program units, you have to set the required format  
B identifier in the KCMF/kcfn field. Interacting with the format handling system, openUTM  
B then automatically formats the messages.

B The format identifier consists of:

- B – the prefix (\*, + or #) which determines the type of the format (see next section).  
B In BS2000/OSD, you can also use the "-" character.
- B – the format name (up to a maximum of 7 characters).

## B Format types

B openUTM distinguishes between \*formats, +formats and #formats:

B \*formats use these formats if you do not want to change the attributes of the format  
B fields (e.g. display attributes) in the program. If you use \*formats only the  
B data fields are transferred in inputs/outputs.

B +formats are formats in which you are allowed to change the attributes of individual  
B format fields in the program. For example, you can return an incorrect entry  
B back to the terminal in flashing mode. Each data field is prefixed with a  
B 2-byte attribute field which you can use to insert the required attribute  
B combination in the program. If you enter binary zero, the attribute combi-  
B nation which was defined when the format was generated applies.  
B openUTM provides you with all the permissible attribute combinations in  
B language-specific data structures, in the KCATC element in COBOL and in  
B the *kcat.c* include file in C/C++.

B #formats are formats in which you are allowed to change both the attributes of the  
B format fields and the global attributes of the format in the program. For  
B input/output, the attribute fields and data fields are divided into separate  
B blocks. For further information refer to the formatting system manual.

B In BS2000/OSD, you can also use -formats. -formats are not formatted by the formatting  
B system, but by the event exit FORMAT, see [page 461](#). If the format identifier starts with the  
B "-" character, openUTM branches to this user-defined formatting routine.

## B Positioning the cursor

B When formatting output with + and \*formats, it is possible to control the position of the  
B cursor in a program (function KDSCUR).

B When setting the cursor, you have to specify the address of either the attribute field or the  
B data field. Which is required depends on what is specified in the FHS start parameter ATTR  
B or NOATTR:

- B – ATTR (only permissible with +formats): the address of the attribute field is specified
- B – NOATTR (for \*formats and +formats): the address of the data field is specified

B The cursor can be placed at the beginning of the assigned data field by correspondingly  
B marking the attribute field. The position of the corresponding screen field is known from the  
B format description. If several attribute fields of an output message are marked at the same  
B time in the program run, then only the first entry is used.

B The cursor can be placed on a specific field by calling the KDSCUR() function. The field  
B at which the cursor is to be placed the next time something is output is specified in the  
B message area as an argument of the function.

B Setting the cursor in COBOL program units:

B `CALL "KDCSCUR" USING FNAME.`

B Setting the cursor in C/C++ program units:

B `KDCSCUR (field name);` (The result of the function is of type void)

B The following arrangement is valid regarding the KDCS attribute functions (+formats and \*formats):

B The cursor can be positioned at the beginning of a message field F(i) by calling the  
B "KDCSCUR" subroutine with the i-th attribute field AF(i) as a parameter. The subroutine  
B contains the cursor identifier as a constant and adds this constant to the attributes already  
B specified in the attribute field passed as a parameter.

B Example:

B `CALL "KDCSCUR" USING AF1.`

B The program unit run would like to place the cursor at the beginning of the message  
B field F1 in addition to the field attribute already defined.

### B Changing format between input and output

B If openUTM outputs a message with a particular format identifier, you have to specify the  
B same format name in the KCMF/kcfn field with MGET for the next input.

B Exception:

B If you do not use partial formats in BS2000/OSD, openUTM tolerates for MGET an incorrect  
B format identifier in KCMF/kcfn: the message is nevertheless formatted in accordance with  
B the last screen format, the 05Z return code is set and the format identifier of the last  
B employed format is displayed in KCRMF/kcrfn.

B If an incorrect format identifier is specified in the KCMF/kcfn field when reading partial  
B formats, openUTM sets return code 03Z, returns the correct format identifier in  
B KCRMF/kcrfn and sets KCRLM to 0. There is no entry in the message area.

## 4.2.1 Screen output functions in format mode

openUTM enables you to request specific screen functions together with the message output (in format mode only with +formats and \*formats). To do this, you enter one of the predefined values for your programming language in the KCDF (device function) field in the KDCS parameter area. For C/C++ these values are in the kcdf.h include file and for COBOL in the COPY element KCDFC.

With "#" formats, KCDF must be set to binary zero. The screen functions are controlled using global attributes.

You can use the following functions in +formats and \*formats:

**KCREPL** (replace)  
Clear and rewrite screen

**KCERAS** (erase)  
If KCLM = 0, all variable fields are erased, the format is retained. If KCLM > 0, new field contents are output in the same format, other variable fields are erased

**KCALARM** (alarm)  
An audible alarm sounds at output

**KCREPR** (reproduce)  
Screen output to printer

**KCRESTRT** (restart)  
screen restart after PEND RS

**KCNODF** (no device feature)  
No screen function, KCDF is set to binary zero

If you are working with message segments, only the entry for the first message segment applies. All subsequent message segments must have binary zero in the KCDF field.

You can combine two or more screen output functions, for example with the statement:

COBOL:  $KCDF = KCREPL + KCREPR + KCALARM$   
C/C++:  $kcdf = KCREPL | KCREPR | KCALARM$

For performance reasons, however, you should use the KCREPL function sparingly.

The effect of KCERAS and KCREPL depends on the selection of the FHS start parameters, see the "FHS User Guide".

For information on error processing when formatting errors occur, refer to the openUTM manual "Messages, Debugging and Diagnostics".

## 4.2.2 Starting services using basic formats

B If necessary, you can make formats available before service start in order to simplify the  
 B entry of the data required by the service.  
 B These formats are called basis formats. You can use basic formats in the following ways:

B ● Define start formats  
 B When configuring the UTM application you can define a specific start format for each  
 B LTERM partner. In applications with the SIGNON event exit (see [page 476](#)) the exit can  
 B read this start format if no user is logged on to this LTERM. In applications without the  
 B SIGNON event exit and without user IDs, this start format is output after the terminal  
 B connection is established through this LTERM partner. When configuring the UTM  
 B application, you can define a user-specific start format for each user ID. After the user  
 B has signed on to UTM under this user ID, UTM outputs this format on screen. In appli-  
 B cations without a SIGNON event exit, openUTM outputs this format after the user has  
 B signed on to openUTM under this user ID. In applications with a SIGNON event exit, the  
 B exit can read this start format if a user has signed on under this user ID. \*formats,  
 B +formats and #formats can be used as start formats. #formats can only be used as start  
 B formats, however, when the SIGNON event service is used.

B ● Format output at end of service  
 B At the end of a dialog service you can output a format from a program unit This format  
 B can be used by the terminal user at the start of the next service.

B ● Format request using KDCFOR  
 B The user at the terminal can use the KDCFOR user command to request a basic format.  
 B openUTM will then output the desired format. You cannot use KDCFOR for #formats.


B In order to start a service by entering a format you have to transfer the desired transaction  
 B code together with the message. You can do this in the following ways:

B – In the format, the first input field is an 8-character field into which the user enters the  
 B desired transaction code.  
 B Note that this field (in +formats including attributes) is not transferred to the message  
 B area. (Exceptions: #-formats, BADTACS event service or if the INPUT exit was used to  
 B make other specifications). If the transaction code field contains blanks, the string up to  
 B the first blank is interpreted as the transaction code. If you use addressing aids to  
 B structure the input area, you must take account of this truncation of the transaction  
 B code.

B – The format permits the transaction code to be entered at any other specified location.  
 B The input is evaluated using the input exit and the transaction code is extracted.



- B – The transaction code is assigned to a function key at generation, see KDCDEF  
B statement SFUNC in the openUTM manual "Generating Applications", and the user  
B presses this function key.
- B In BS2000 the function key must be an F-key because messages are not passed when  
B the K-keys are used (see also the KDCDEF statement SFUNC).
- B – The format contains one or more UTM control fields. In a control field, the transaction  
B code is either entered by the user or is already preset. The "UTM control field" attribute  
B is assigned when the format is created.

B  If you use basic formats to start the service, you must always check the  
B KCRMF/kcrfn field in the first program unit in order to determine whether the  
B intended format was used to start the service.

## 4.2.3 Using multiple partial formats

B A screen display may consist of multiple partial formats. A partial format normally occupies  
B a part of the screen only. You must enter a separate MPUT NT call for each partial format.  
B If the screen is to be read again, a separate MGET call is required for each partial format.

### 4.2.3.1 Output formatting with partial formats

- B If a new screen is constructed of two or more partial formats, you must also specify the  
B value KCREPL in the KCDF field of the first MPUT NT. In all subsequent MPUT calls KCDF  
B must contain a binary zero. Otherwise, openUTM terminates the conversation abnormally,  
B setting KCRCCC = 70Z and KCRCDC = K606.
- B You cannot mix format mode and line mode within a dialog message: If the format changes  
B from line mode to format mode or vice versa within a partial message, the service is termi-  
B nated with KCRCCC=75Z.
- B You may switch between \*formats and +formats.

**B Updating a screen**

B You can update a screen with one or more MPUT NT calls. In the first MPUT call you can specify any value except KCREPL in the KCDF field since KCREPL is used to erase the entire screen. In subsequent MPUT NT calls KCDF must be set to binary zero as for setting up a new screen.

B If the format is changed, openUTM only erases those old partial formats which are overlapped by new ones.

B You can use the screen output function KCERAS to erase the variable fields when outputting a partial format.

**B Outputting partial formats with FPUT/DPUT NT**

B You can use FPUT/DPUT NT to output individual partial formats. However, you cannot construct a screen with multiple FPUT/DPUT NT calls, since openUTM transfers each partial format sent using FPUT/DPUT NT as a separate message when outputting to the screen. You may switch between format mode and line mode.

B Before a formatted asynchronous message is output the screen is automatically cleared. It is therefore not possible to use FPUT/DPUT NT to update the displayed format.

B openUTM responds with an automatic screen restart to inputs from asynchronously output formats - with the exception of command entries (see [section "Screen restart" on page 119](#)).

B If you output to printers, all message segments sent with FPUT/DPUT NT are combined and output as one message, even when you change between format mode and line mode.

**4.2.3.2 Input formatting with partial formats**

B The terminal user can enter data in the partial formats, but is not required to enter data in every one.

B Since openUTM passes all variable fields, after the INIT call the KCRMF/kcrfn field contains the name of the first partial format containing variable fields. After the MGET call KCRMF/kcrfn contains the name of the next partial format with variable fields. After the last partial format with variable fields has been read KCMF = KCRMF (kcrfn=kcrfn). If you try to read another partial format, although the message has already been entirely read, openUTM responds with the return code 10Z.

B *Example*

B MPUT output with 3 partial formats: PARFOR1, PARFOR2, PARFOR3;  
B variable fields PARFOR1, PARFOR2

B	KDCS call:	Returned by openUTM:
B	INIT	KCRMF = PARFOR1
B	MGET KCMF =	KCRMF = PARFOR2
B	PARFOR1	KCRCCC = 000
B	KCLA = ...	KCRLM = ...
B		Data in message area
B	MGET KCMF =	KCRMF = PARFOR2
B	PARFOR3	KCRCCC = 03Z (as KCMF contains a value other than that returned
B	KCLA = ...	in KCRMF by the preceding MGET
B		KCRLM = 0
B	MGET KCMF = KCMF	KCRMF = KCMF = PARFOR2
B	= PARFOR2	KCRCCC = 000
B	KCLA = ...	KCRLM = ...
B		Data in message area
B	MGET KCMF = ...	KCRMF = KCMF = PARFOR2
B	KCLA = ...	KCRCCC = 10Z
B		KCRLM = 0

### B **Entering a format consisting of multiple partial formats to start a service**

B If you want to start a service by entering a format consisting of multiple partial formats you  
B can use the first variable field of the first partial format to specify the transaction code (for  
B other options, see [page 113ff](#)).

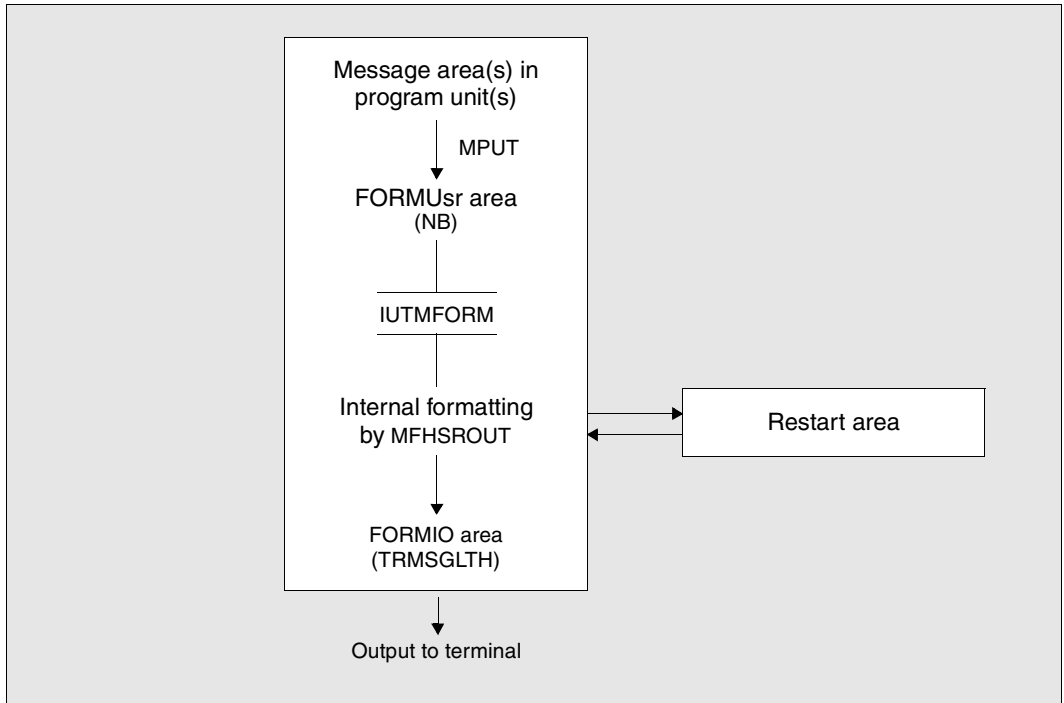
B openUTM then automatically separates the field with the transaction code from the  
B message: for \*formats the first 8 characters (transaction code) are removed in the first  
B partial format, for +formats the first 10 characters (attribute field and transaction code). The  
B other partial formats remain unchanged.

B An asynchronous service can also be started using a format consisting of multiple partial  
B formats which was output at the end of a dialog service. The asynchronous program must  
B be written in the same way as the dialog version: the INIT call provides the name of the first  
B partial formats with variable fields in KCRMF/kcrfn. Using FGET the program can retrieve  
B the input data from this partial format. In the case of an empty input message (transaction  
B code without data), the FGET call returns 10Z to the program unit.

#### 4.2.4 Message flow in openUTM (BS2000/OSD)

B The following diagram displays the message flow in the event of an MPUT call to a terminal:

B



B Message flow for MPUT to terminal

B The messages from the program units enter the FORMUSR area via UTM buffers at MPUT  
 B time. The size of this area, which must be able to contain the entire logical message, is  
 B defined in the NB parameter of the MAX statement (see the openUTM manual "Generating  
 B Applications"). This message is formatted by the FHS formatting system MFHSROUT and  
 B stored in the FORMIO area. The size of this area is defined in the TRMSGLTH operand of  
 B the MAX statement. The message is sent from this area to VTSU-B before it is sent to the  
 B terminal via BCAM.

B If partial formats are used, you have to make sure that the FORMUSER and FORMIO areas  
 B are always able to store the entire message.

**B** **Input formatting**

**B** With input formatting, the message stream runs in the opposite direction, using the same areas.  
**B**

#### 4.2.5 Controlling the output in line mode (BS2000/OSD)

**B** openUTM operates in line mode whenever the name specified in KCMF/kcfn starts with a blank. Under BS2000/OSD, openUTM also enables you to use the following options to control the screen output in line mode:  
**B**

**B** – use logical control codes to structure the output

**B** – also available in line mode:  
**B** screen functions KCALARM, KCREPR, KCRESTRT (see [page 111](#))

**B** – available in line mode only:  
**B** screen functions KCCARD and KCEXTEND (extended line mode)

**B** – use of edit profiles

**B** For a description of how to prepare messages for printing, refer to [section “Output to printers” on page 94](#).  
**B**

**B** **Using logical control codes for structuring**

**B** In line mode you can use logical control codes to structure the output. The VTSU (Virtual Terminal Support) terminal support facility then converts the control codes into the physical control codes necessary for the addressed device. You can use all logical control codes permitted by the TIAM access method, see the "TIAM User Guide". In some programming languages data structures are available which you can copy into the program (see the "TIAM User Guide").  
**B**  
**B**  
**B**  
**B**  
**B**

**B** **KCCARD screen function**

**B** You can use the KCCARD screen function to input data via magnetic strip cards:  
**B** If you use the KCCARD screen function to output a dialog message, the keyboard is locked and the terminal user is requested to insert a magnetic strip card into the ID card reader.  
**B**

**B KCEXTEND screen function**

**B** If you use this screen function the output fields are set to a particular, predefined value  
**B** which is equivalent to the macro WRTRD, ... EXTEND=YES. All output fields are displayed  
**B** in half video and protected by default, see the "TIAM User Guide" for further information.

**B Edit profiles**

**B** An edit profile is a set of attributes used for output in line mode. Using edit profiles you can,  
**B** for example, specify that characters entered at the terminal are not to be displayed  
**B** (password entry) or request lower case to upper case conversion. Alongside the KCCARD  
**B** screen function edit profiles are another way of requesting input from ID card readers. You  
**B** can also perform other screen functions such as, for example, KCREPR.

**B** You use the KDCDEF statement EDIT to specify the name and attributes of edit profiles at  
**B** generation (see the openUTM manual "Generating Applications"). The length of the name  
**B** can be up to 7 characters. You can use the MPUT/FPUT/DPUT calls to address an edit  
**B** profile in a program unit. To do this, enter a blank in the first byte of the KCMF/kcfn field and  
**B** the name of the edit profile in the remaining seven bytes.

**B** Edit profiles are handled like format names, i.e. the name of the edit profile of the last output  
**B** message is returned in the KCRMF/kcrfn field after INIT (as with format identifiers). If you  
**B** use MGET/FGET calls the name is entered in the KCMF/kcfn field.

**B** Note the following points when edit profiles:

- B** ● When using MPUT, FPUT or DPUT calls with edit profiles, no screen function may be  
**B** specified (KCDF must be set to binary zero) as otherwise openUTM responds to MPUT  
**B** with 70Z or to FPUT/DPUT with 40Z.
- B** ● As soon as edit profiles are generated for an application, you must use 8 blanks in  
**B** KCMF/kcfn to identify messages in line mode (without edit profile); it is not sufficient to  
**B** fill set the first byte with blanks, since openUTM interprets this as an incorrect format  
**B** identifier and sets the corresponding error code in KCRCCC.
- B** ● If you use MPUT NT, FPUT NT or DPUT NT to issue multiple message segments to a  
**B** terminal, the name of the edit profile must not change, otherwise openUTM responds  
**B** to MPUT with 75Z and to FPUT/DPUT with 45Z. When issuing partial messages to  
**B** printers edit profile names may change.
- B** ● If the screen was overwritten by the output of an asynchronous message, openUTM  
**B** performs an automatic KDCDISP when using edit profiles.
- B** ● For messages to UPIC clients the names of the edit profiles are treated like format  
**B** identifiers, i.e. they are also passed (therefore they also appear in the UTM program unit  
**B** in KCRMF/kcrfn), although they do not have any effect.

- B
B
  - For messages via distributed processing the names of the edit profiles are processed like format identifiers:
- B
B
  - In distributed processing via LU6.1 the names of the edit profiles are transferred (i.e. they appear in KCRMf/kcfn). However, they have no effect.
- B
B
B
  - In distributed processing using OSI TP, no edit profile must be specified in KCMF/kcfn, since openUTM uses the format identifier to transfer the abstract syntax.

### 4.3 Screen restart

openUTM automatically restarts the screen for terminals in the following cases:

- after a service has been interrupted, the user signs on to the application again, e.g.
  - after the application has been terminated normally without terminating the service
  - after the application has terminated abnormally
  - KDCOFF within a service
  - connection loss

You can deactivate this screen restart following repeated sign-on for particular user IDs or LTERM partners. To do this, you set the operand RESTART=NO in the USER or LTERM statements at generation.

- during a service, an asynchronous message was output to the screen and the user did not issue a KDCDISP command to continue the dialog, i.e. the user attempts to input the message into the asynchronously output format (only command entries are possible).
- within a service, after a follow-up transaction has been reset by MPUT RM, KCDF = KCRESTRT and PEND RS.

- B
  - when certain operating errors occur (e.g. screen switched on or off, AM key).

Screen restart is only possible because openUTM stores the data on the last screen structure, initially in a buffer in the process-specific system storage area. When PEND RE/FI/FC occurs, i.e. at the end of the transaction, openUTM writes this information to the restart area of the KDCFILE, see also the openUTM manual "Generating Applications".

- B
B
B
B
B
B
  - When formatting output, openUTM writes the entire screen contents to the restart area if the entire screen is to be restructured (KCDF= REPLACE, change of format). During a screen update, only the fields which are also overwritten at the terminal are updated in the restart area. Similarly, during input formatting only the fields received from the terminal are updated in the restart area.

For restart, openUTM always saves only the logical (unformatted) message which is then automatically formatted if a restart is necessary.

Restart for UPIC clients:

A UPIC client can request a service restart. openUTM always provides the UPIC client with just the MPUT message of the last synchronization point.

- B If you use a formatting system, openUTM automatically performs the setup, modification
- B and storage of the restart area in interaction with the formatting system.
- B However, if you use -formats and write the formatting routine yourself, openUTM will save
- B the restart area, but the setup and modification of the area are the responsibility of the
- B formatting routine programmed by you (see event exit FORMAT, [page 461](#)).



---

## 5 Program structure in distributed processing

The term "distributed processing" applies to all types of processing where server applications with the same authorization levels interact (server/server communication). openUTM supports distributed processing via the LU6.1, LU6.2 and OSI TP protocols. Partners of a UTM application for distributed processing can be other UTM applications, OpenCPIC applications or applications that are based on transaction systems from other manufacturers.

You will need the openUTM-LU6.2 product to communicate via the LU6.2 protocol; the program interfaces for the communication via LU6.2 are identical to those for the communication via OSI TP.

openUTM is designed to operate with global transaction management when the LU6.1 is used, i.e. the transactions of all participating partners are synchronized.

When using the OSI TP protocol you can decide whether or not openUTM is to operate with global transaction management.

Client/server communication is not regarded as distributed processing, despite the fact that the client performs processing tasks in client/server communication, e.g. plausibility checks. However, fixed client and server roles are assigned. The client does not make services available which could be used by other applications.

Communication with OpenCPIC partners is done via the OSI TP protocol; this means that all statements and rules that are agreed upon in this manual for the communication with OSI TP partners are valid for these partners, even when a client of a UTM application is implemented with an OpenCPIC application.

## 5.1 Addressing remote services

Before you can send a message to a remote service, you must first address this remote service. To do this, use the **APRO (Address PROgram)** call.

In distributed processing you can use the functions of both remote services and, by using openUTM's message queuing functionality, asynchronous services. You use **APRO DM (Dialog Message)** to address remote dialog services. and **APRO AM (Asynchronous Message)** to address asynchronous services.

If a remote service is addressed in a transaction with APRO AM/DM, then a message must also be sent to the remote service in the same transaction.

Under certain circumstances you can also use APRO AM to address a dialog service via OSI TP, i.e. to address an asynchronous job to a dialog service (see [page 192](#)).

The following APRO call parameters are of relevance when a remote service is addressed:

- In the **KCRN (Referenced Name)** field, enter the LTAC name of the remote service. The LTAC name (Local TransAction Code) is the name under which the remote service is known in the configuration of the local application. The LTAC name is assigned at generation.

If this LTAC name is already associated with a fixed partner application in the configuration you only have to specify the name for the APRO call to unambiguously determine the remote service. This is called single-step addressing. Single-step addressing is always advisable in cases where there are no alternative partner applications, e.g. because the requested service is only available from one partner application.

In two-step addressing, the partner application is specified explicitly when calling APRO:

- The **KCPA (Partner Application)** field is used to specify the (OSI-)LPAP name of the partner application in two-step addressing. The (OSI-)LPAP (Logical Partner Application) name is the name by which the partner application is known in the configuration of the local application. The (OSI-)LPAP name is assigned at generation. A MASTER-OSI-LPAP can also be specified as the LPAP name.

Two-step addressing is always advisable in cases when the requested service is available from multiple partner applications. Which partner application is selected depends on the particular situation.

If you specify the (OSI-)LPAP name of a partner application in KCPA, although the LTAC name specified in KCRN is already assigned to a partner application in the configuration, then the specification in KCPA takes precedence over the configuration assignment.

- In the **KCPI** (Partner Identification) field, you assign an identification to the remote service for the duration of the interaction. This identification, also called the service ID, is freely selectable. However, the first character must be ">". The scope of the validity of the service ID is the local service; in other words, if two services use the same service ID at the same time, they are thus addressing different remote services. You have to specify this identification in the KCRN field for all communication calls to the addressed service. When interaction with the remote service terminates, you can use the service ID in further APRO calls to identify the same service or other services.

For the following calls you have to specify the service ID in the KCRN field:

- MPUT calls to send dialog messages to the remote service
- FPUT/DPUT calls to send asynchronous messages to the remote service
- MGET calls to read dialog messages or status information from the remote service
- MCOM BC calls to start a job complex whose basic job is directed to the remote service
- CTRL calls to control OSI TP dialogs

## 5.2 Distributed dialogs

The rule of strict dialog also applies to distributed processing, i.e. in communications between two partners, after a message to the partner you have to receive a response before you can send another message to this partner.

The period between receiving a dialog message and sending the next dialog message is called a **processing step**. The next message may either be a response, in which case the processing step is equivalent to a **dialog step**, or a message to another partner.

A job submitter can simultaneously interact with multiple job receivers. The next programming step in the job submitter only starts when all the responses have been received from the job receivers. A separate program unit run is generally started for each processing step (exception: PGWT).

The job-submitting service can be either a dialog service or an asynchronous service.

### Exchange of dialog messages

You use MPUT to send and MGET to receive dialog messages between partner services in the same way as with dialog messages without distributed processing.

If a job-receiving service exchanges messages with a job submitter it behaves as if it were communicating with a terminal: you have to enter blanks in the KCRN field when using MGET and MPUT.

When exchanging messages with a job receiver, a job-submitter service must specify the service ID of the job receiver in the KCRN field (see [page 122](#)).

### 5.2.1 Controlling communication in the program

In distributed processing, multiple program units in various UTM applications are used to process a job. A program unit can be addressed by a client (terminals or client programs), from the application's own programs or from other applications. The program unit has therefore to decide independently from the partner which task to process, to whom it sends a message, and whether or not it has to terminate a distributed processing.

For this reason, openUTM provides a range of information about the communication partner and the current communication status. This information can be evaluated and used for communication control. For a description of this information refer to the sections "Programming aids", for LU6.1 [page 132ff](#), and [page 149ff](#) for OSI TP.

## 5.2.2 Error handling by the program unit

There are two situations in which a program unit can respond to errors:

- In the case of less serious errors:  
The program unit retains control and can react with a programmed reset (see below).
- After a service restart:  
The program unit is restarted, receives status information and possibly a reset message and can respond specifically to the error situation.

### 5.2.2.1 Programmed reset

If less serious errors occur, the program unit retains control and can reset the distributed transaction with PEND RS or PEND ER/FR, or the local transaction with RSET. If communication is via OSI TP transactions can also be reset using PGWT RB, see [page 158](#).

The comments below apply to distributed processing with global transaction management, i.e. for LU 6.1 and for OSI TP with functional unit commit. The effect of these calls in the case of OSI TP without functional unit commit is described on [page 158](#).

#### PEND RS

PEND RS can be called in a job-submitting or job-receiving service.

In a distributed transaction, a PENDING RS has the following effect:

With PENDING RS, all the services participating in a distributed transaction are reset to the last synchronization point. As opposed to the PENDING ER/FR call, with PENDING RS a service remains open if it has already reached a synchronization point. Services which have not yet reached a synchronization point are terminated.

The following situations are possible:

- PENDING RS in the first transaction of the uppermost job-submitting service.  
All participating services are terminated without any service restart. Chained services are restarted after the reset.
- PENDING RS in the first transaction of a job-receiving service  
The job-receiving service is terminated. If the uppermost job-submitting service has already reached a synchronization point, openUTM executes a service restart (with the message K034 to the terminal). The job-submitting service receives status information (service status "R" / "Z" and transaction status "R") which you read using MGET NT.

- PEND RS in a follow-up transaction of the job-submitting service or job-receiving service.

In this case a reset message has to be sent with MPUT RM prior to the PEND RS, otherwise openUTM aborts the service with KCRCCC = 83Z. The reset message goes to the follow-up program unit specified at the last synchronization point of the (local) service. After the reset openUTM executes a service restart. The form this takes depends on the destination of the output message at the last synchronization point and on who called PEND RS:

- In the case of an **output message to the client** the service restart begins in the job-submitting service.
  - If PEND RS was issued in the uppermost job-submitting service, then a screen restart is executed (as without distributed processing).
  - If PEND RS was issued in a job-receiving service, then a screen restart is executed and the message K034 output. The next entry from the client is read by the follow-up program unit of the last synchronization point. If a follow-up program unit sends a new message to the job receiver (which executed PEND RS), then this job receiver first reads the reset message and then the message sent to it. If job-receiving services are terminated by the reset (i.e. the job receiver was also a job submitter), then you also have to read the associated status information using MGET NT.

In the case of LU6.1, follow-up processing in the job-receiving service does not start until a message is present for this service. If the job receiver is not included in the next transaction after the reset, this may not take place until a later follow-up transaction.

In the case of OSI TP, the follow-up program unit runs from the last synchronization point in the job-receiving service which called PEND RS and always runs in the next transaction: The program is started if a message is received from the job submitter or if the job submitter has requested end of transaction.

- With an **output message to a service**, openUTM starts the follow-up program unit specified at the last synchronization point. This program unit has to read the reset message; subsequently the output message can be read; status information from the job-receiving services may be present, too. If the (uppermost) job-submitting service participated in the reset transaction, then openUTM issues message K034.

### Programmed PEND ER/FR

In distributed transactions, the effects of the PEND ER and PEND FR calls are identical; however PEND FR does not, in contrast to PEND ER, generate a DUMP. PEND FR enables you to respond to errors other than programming errors (e.g. meaningless data).

The effect of PEND ER or PEND FR differs in the job-submitting and the job-receiving services:

- If PEND ER/FR is issued in the job-submitting service, this and all job-receiving services are terminated with PEND ER/FR.
- If issued in the job-receiving service, only this service is terminated and the distributed transaction is reset to the last synchronization point. The service restart then begins at this synchronization point and the next program started receives status information from the job-receiving service which called PEND ER/FR. If the job-submitting service has not yet reached a synchronization point, it is also terminated by a PEND ER/FR.

If the preceding distributed transaction was terminated in the job-receiving service, the follow-up program unit specified at the last synchronization point is started in the job-submitting service.

If the job-receiving service wishes to terminate with PEND ER/FR, it first has to execute an MPUT, otherwise PEND causes it to be terminated by the system with KCRCCC = 83Z (service status Z instead of E). Exceptions to this rule are dialogs via OSI TP in which an MPUT to the job submitter is not permitted (KCSSEND=N).

The job-submitting service always receives status information which has to be read with MGET NT.

### RSET

The effect of the RSET call is the same in job-submitting and job-receiving service.

After a RSET call in a program unit run belonging to a distributed transaction, the behavior of openUTM depends on the RSET generation parameter of the UTMD statement.

- If RSET = LOCAL is generated, openUTM allows the RSET call merely to reset the local transaction. Please note that the data environment (GSSBs, LSSBs, TLSs, ULS,...), with the exception of the SPAB, is reset to the last synchronization point. The following applies to job-receiving services addressed within the transaction: if a message was sent to a job-receiving service in a preceding dialog step terminated with PEND KP or PGWT KP, then this service remains addressed, otherwise the service identifier is deleted.
- If RSET = GLOBAL is generated, the program unit run must be terminated with PEND FR/ER/RS. This then causes the distributed transaction to be reset.

### 5.2.2.2 Error handling after service restart

If the distributed transaction has to be reset, service restart tries as far as possible to restart the program for which there was a message at the end of the last distributed transaction, or to which the next input from the client is directed. The programming rules (see 134ff for LU6.1, [page 154ff](#) for OSI TP) ensure that one and only one message exists for the client or for a program in a service at the end of a distributed transaction.

The program unit can tell from the service ID in the KCKNZVG/ kccv\_status field in the KB header if a service restart has occurred: after a service restart this field has the value "R".

If the service restart occurs in the job-submitting service, the program unit normally receives **status information** from the job-receiving service if the job-receiving service caused the reset and is or was consequently terminated. Such status information is read with **MGET NT**; it is a message of length 0 and provides the service and transaction status of the job-receiving service in the KCVGST/kcpcv\_state and KCTAST/kcpta\_state fields in the KB return area (see [page 132ff](#)).

There are 3 different cases with service restart:

1. There was a message for the client at the last synchronization point.

The follow-up program unit in the job-submitting service can now read the new input from the client with MGET and receives as status information the service status O and the transaction status C.

If the reset of the transaction was caused by an error in a job-receiving service and the job-receiving service was subsequently terminated, the KCRPI field receives the service ID of the service which caused the reset. Status information from this service can then be read with MGET NT and KCRN = service ID.

2. There was a message at the last synchronization point from a job-receiving service to a job-submitting service.

The follow-up program unit in the job-submitting service can now use MGET as usual to read the message, and additionally receives the service and transaction status of this job-receiving service.

If this job-receiving service was reset and terminated by an error, you only receive the corresponding status information. If the reset was caused by another job-receiving service, you receive the transaction status C as status information with the first MGET. Subsequently more status information can be read, as in case 1.

If the message cannot be sent by the job-receiver (see next section), then you just receive status information from the job-receiving service.



3. There was a message at the last synchronization point from a job-submitting service to a job-receiving service. If possible, the follow-up program unit is then started in the job-receiving service.

If the follow-up program unit cannot be started (e.g. because the application was terminated and not restarted within the generated wait time or because the service terminated with PENDING), then the follow-up program unit in the job-submitting service is started and receives status information.

There is status information from all job-receiving services which caused reset of the distributed transaction and which were or are being terminated.

If, after a service restart of the job-submitting service, a job-receiving service is addressed again and an error recurs, the job-submitting service can be reset more than once to the same synchronization point. Since the status information from the preceding reset is retained, it is possible to have multiple status information.

If there is multiple status information, then with each MGET you receive the service ID of the next service with status information. Status information from several job-receiving services has to be read in the order proposed by openUTM (KCRPI). The KCMF/kcfn field is to be set with blanks when reading the status information.

If a remote service in the reset transaction is newly created by the APRO call, then there is likely to be status information from it although this service does not actually exist at the synchronization point from which the restart begins.

### No message from job receiver

There are two reasons why a job-receiving service cannot send a result to the job submitter application:

1. The job receiver was not started for one of the following reasons:
  - no logical connection to the job receiver application exists and no connection could be established during the generated wait period
  - no session or association to the job receiver application could be reserved during the generated wait period
  - the job was sent to the job receiver application. However, the transmitted transaction code is unknown or locked
  - The job-submitting service is terminated using KDCSHUT W
2. The job-receiving service was started, but errors have occurred during processing of the job-receiving service or the communication path was disturbed. For this reason the transaction was reset in the job-receiving service and the job-receiving service has been terminated. Additionally, the session/association to the job-receiving service which terminated through error has been released. The following error situations can occur:
  - no response has been received by the job-submitting service from the job-receiving service within the generated wait period
  - the job-receiving service has been terminated abnormally because of a severe error
  - the job-receiving service has terminated using PEND ER or was terminated because of a severe program error.

### 5.2.3 Load distribution using LPAP bundles

openUTM provides the LPAP bundle function for the OSI TP and the LU6.1 protocols.

LPAP bundles allow load distribution and the use of alternative connections to a partner application. If a UTM application has to exchange a very large number of messages with a partner application then load distribution may be improved by starting multiple instances of the partner application and distributing the messages across the individual instances. In an LPAP bundle, openUTM is responsible for distributing the messages to the partner application instances. An LPAP bundle consists of a master LPAP and multiple slave LPAPs.

The slave LPAPs are assigned to the master LPAP on generation. In normal circumstances, the physical connections (CONS) of the individual slave LPAPs address different partner applications.

#### Operating an application

To allow openUTM to distribute the messages to the slave LPAPs, you address the messages to the master LPAP in the program units.

openUTM distributes these messages to the slave LPAPs in sequence. The system always attempts to find a slave LPAP via which the message can also be sent, i.e. to which a connection is established and for which, for instance, the queue level has not yet been exceeded.

For more details on this, refer to the description of the APRO call on [page 208ff](#).

#### Administration

The "Master LPAP" and "Slave LPAP" properties are displayed via the administration interface.

All the slave LPAPs are displayed for a master LPAP and the master LPAP is displayed for a slave LPAP.

Using the administration facilities, you can set the status of master LPAPs to ON or OFF. If you change the status of a Master LPAP, this causes the status to be changed accordingly on all slave LPAPs.

## 5.3 Distributed dialogs via LU6.1

The LU6.1 protocol (Logical Unit 6.1) is a SNA protocol defined by IBM. It has been continuously developed and has become the industry standard. Communication is performed with transaction management over multiple applications.

In addition to using LU6.1 to connect UTM applications, you can also connect UTM applications with CICS and IMS applications running on mainframes.

The sections below explain the programming aids provided by openUTM for distributed processing, the rules for these dialogs and how you can use existing program units as an LU6.1- job submitter.

### 5.3.1 Programming aids

openUTM makes a range of information available to program units. You can evaluate and use this for communication control:

- after the INIT call
- after using MGET to read the dialog message

After the INIT call, you can ascertain the following from the communication area:

- the user ID or session name under which the program unit was started (KCBENID/kcuserid field)
- the communication protocol used by the partner (1 is entered in KCVGST/kcpcv if LU6.1 is used)
- whether it is a service restart (KCKNZVG/kccv\_status is then set to "R").

After the MGET call, openUTM returns the status of the partner service in the KCVGST/kcpcv\_state field and the status of the partner transaction in the KCTAST/kcpta\_state field.

For COBOL, a higher-ranking status field KCRST is defined which contains the KCVGST and KCTAST fields.

Using these stati you can, for example, ascertain whether the partner service has already requested end of transaction and is waiting for the termination of distributed processing. This feature can be used to control the program run and ensure that the programming rules are observed, even in situations where the exact security processes used by the partner are not known. On the other hand, it also allows the job-submitting service to react to errors in the job-receiving services.

**Service status of the partner**

KCVGST/ kcpcv_state	Meaning
I	(inactive): The service is inactive, i.e. it has not yet been started or it has not been possible to start it, e.g. because the TAC is unknown in the job-receiving service.
O	(open): The service is open.
C	(closed): The service terminated with PEND FI.
R	(reset): The service has been terminated by the program with PEND RS.
E	(error): The service has been terminated by the program with PEND ER/FR.
Z	The service has been terminated by openUTM because of an error.
T	(timeout): The service has been or is terminated incorrectly, as no response has been received within the specified wait period or it has not been possible to reserve a free session within the specified wait period.

**Transaction status of the partners**

KCTAST/ kcpta_state	Meaning
I	(inactive): There is no transaction because no service exists.
O	(open): The transaction is open; the last dialog step has been terminated with PEND KP.
P	(prepare to commit): The service has requested end of transaction, but the distributed transaction has not yet terminated. In this situation the service waits for an acknowledgment which it receives as soon as the distributed transaction is terminated. Just as with PEND KP, the resources (GSSB, TLS) remain locked until that point.
C	(closed): The last distributed transaction in which the service participated is terminated.
R	(reset): The distributed transaction and thus also the local transaction have been reset.
M	(mismatch): The services participating in a distributed transaction cannot agree on a common synchronization point for service restart. This can only occur with a timeout or after termination and start of a UTM-F application.

### 5.3.2 Programming rules and recommendations

The following sections explain the rules which have to be observed when using distributed dialogs via LU6.1. If you follow the suggested programming recommendations in the final section (see [page 137](#)) the presented rules are automatically observed.



Note that PGWT CM and PGWT RB calls are not permitted in distributed dialogs via LU6.1.

#### Effect of PEND calls and rules for usage

The PEND calls in job-submitting **and** job-receiving services control the distributed transaction, i.e. they decide when the **common** synchronization point of the two transactions is to be set.

Which PEND calls a service may use and their effect depends on the service and transaction status of the partner service.

The following table describes the effect of the PEND calls in the job-receiving and job-submitting services and the rules for how to use them:

Variant	Effect/rule of usage
PEND KP PGWT KP	<p>No synchronization point is requested and no synchronization point is set.</p> <p>The service remains in transaction status "0" and in service status "0". An open DB transaction remains open. The follow-up program unit specified in KCRN is started in the job-submitting service as soon as all results from the job-receiving services have been received.</p> <p>This is permissible if the program unit sends a message to the terminal or the client (LTERM partner) or if the message is addressed to a partner service and this does not have transaction status "P".</p>
PEND RE	<p>End of transaction (synchronization point) is requested.</p> <p>End of transaction (synchronization point) is requested. If all partner services already have transaction status "P" (i.e. they have already requested end of transaction), then the end of transaction is executed by all participating services, i.e. a common synchronization point is set. If a partner service is not in transaction status "P", the local service passes to transaction status "P" and waits for the partner service to request end of transaction, too. The follow-up program unit specified in KCRN is started in the job-submitting service as soon as all results have been received from the job-receiving services.</p> <p>Permissible in the two cases below:</p> <ul style="list-style-type: none"> <li>– no partner service with an open transaction exists.</li> <li>– there is exactly one partner service with an open transaction. The program unit sends a message to this partner.</li> </ul> <p>If multiple partner services with open transactions exist then PEND RE is always forbidden.</p>

Variant	Effect/rule of usage
PEND FI	The effect is the same as with PEND RE, except that end of service is requested simultaneously. The service is terminated at the next synchronization point.  This is permissible if no job-receiving services are open any longer, i.e. all job receivers must have issued PEND FI before the job submitter is permitted to issue PEND FI.
PEND FC	The effect is the same as with PEND FI, except that after end of service, the chained follow-up service is started immediately.  This is permissible in job-submitting services only (never permitted in job receivers), and only if no further job-receiving services are open, i.e. all job receivers must have issued PEND FI before the job submitter is permitted to issue PEND FC.
PEND PR PEND PA	No effect on the transaction.
PEND SP	The transaction is closed, i.e. a synchronization point is set and the dialog step is continued. Not allowed if there are partner services with open transactions.
PEND RS	With PEND RS all the services participating in the distributed transaction are reset to the last synchronization point. All services created in the distributed transaction are terminated by the reset. Services which have reached at least one synchronization point remain open.

### Programming rules

A set of rules has to be adhered to when programming distributed transactions. Violation of these rules leads to abortion of the service with (internal) PEND ER and KCRCCC=87Z.

These rules determine:

- how transactions and services have to be terminated when using distributed processing and
- where the output message may be sent.

Program unit runs, transactions and services are terminated as usual via the various PEND variants; of particular importance here are the PEND variants KP, RE, SP, FI and FC, and for error handling RS, ER and FR.

Using the variants KP, RE and SP you have to specify the follow-up program unit in KCRN, which is used to continue the service. You start this follow-up program unit with PEND KP/RE after all results from the job receivers have been received, or, if the message was addressed to the job submitter, after the next message is received from the job submitter. If you use PEND SP the follow-up program unit is started immediately.

The PGWT KP call may be used whenever a PEND KP is allowed.

The following rules hold for correct programming of a distributed transaction:

- **Service rule:**

a job-submitting service may only be terminated once all the associated job-receiving services have been terminated (with PEND FI). PEND FC is prohibited in job-receiving services.

After the PEND FI of the job-receiving service, the job-submitting service can terminate the subsequent dialog step with PEND KP, RE, SP, FI or FC. It may not send any more messages to this job-receiver service.

- **Transaction rule (single-level):**

a job-submitting or job-receiving service can terminate a dialog step either with or without end of transaction subject to the following restrictions:

- a job-submitting service **may not request end of transaction** if the transaction is open in the job-receiving service, and the output message is directed to the client
- a service **must request end of transaction** if the output message is directed to a partner service which has requested end of transaction.

With multi-level distributed transactions, i.e. when a job-receiving service is itself a job-submitting service or when a job-submitting service addresses several job-receiving services in one transaction, then the transaction rule is generalized as follows:

- **Transaction rule (multi-level):**

a service can terminate a processing step either with or without end of transaction subject to the following restrictions:

- a service **may not request end of transaction** if there is a partner service with an open transaction and the output message is not directed to this partner service. If multiple partner services with open transactions exist, it is never permitted to request end of transaction.
- a service **must request end of transaction** if the output message is directed to a partner service which has requested end of transaction

If a service has multiple partner services with open transactions, PEND RE, PEND FI and PEND SP are never permitted.

The tables on pages [134](#), [137](#) and [139](#) illustrate the situations when you can use the PEND call.

If you follow the "bottom-up strategy" described below, then the programming rules are adhered to automatically.



**Programming recommendation: Bottom-up strategy**

The bottom-up strategy is such that distributed transactions are always terminated from the bottom up, i.e.

- a job-receiving service always requests end of transaction before its job-submitting service and sends its output message to its job-submitting service, and
- a job-submitting service only requests end of transaction once all its job-receiving services have requested end of transaction. The output message then goes to its own job submitter.

**PEND variants depending on the partner status**

When using the PEND call must always consider the service status and transaction status of the partner service. Following MGET, openUTM returns the status information in the KCVGST/kcpcv\_state and KCTAST/kcpta\_state fields.

The cases illustrated in the following tables contain no new rules. They illustrate the rules described in the preceding sections.

*PEND variants in the job submitter*

The calls PEND PA and PEND PR are not included because they have no special features for distributed processing.

Partner status		PEND variants in the job-submitting service	
KCVGST / kcpcv_state	KCTAST / kcpta_state	permitted variants and their effect and limitations	
"O"	"O"	KP:	transactions in the job-receiving and job-submitting services remain open - default.
		RE:	requests end of transaction. Only permitted if the output message is addressed to this job receiver and no further partner with open transactions exist.
		RS:	resets the job-submitting and job-receiving transaction to the last synchronization point. Services in this transaction are terminated.
		ER/FR:	resets the distributed transaction; the job-receiving service is terminated.

Partner status		PEND variants in the job-submitting service permitted variants and their effect and limitations
KCVGST / kcpcv_state	KCTAST / kcpta_state	
"O"	"P"	<p>KP: only permitted if the output message is addressed to another partner service or another client; the transactions remain open (not recommended because of PTC state)</p> <p>RE: terminates the distributed transaction.</p> <p>SP: terminates the distributed transaction.</p> <p>RS: resets the job submitter and job receiver transaction to the last synchronization point. Services in this transaction are terminated.</p> <p>ER/FR: resets the distributed transaction; the job-receiving and job-submitting service are terminated</p>
"O"	"C"	<p>KP: transaction in job-submitting service remains open.</p> <p>RE: transaction in job-submitting service is terminated if the output message is addressed to the client, otherwise the service passes to transaction status "P".</p> <p>SP: terminates the job submitter transaction.</p> <p>RS: resets the transaction in the job-submitting service to the last synchronization point; services which started in this transaction are terminated.</p> <p>ER/FR: terminates job-submitting and job-receiving service and resets the local transaction.</p>
"C"	"P"	<p>The output message must be addressed to another partner or the client or (with PEND SP/FC) to a follow-up program unit!</p> <p>KP: not recommended, since the job-receiving service waits in state PTC.</p> <p>RE: terminates the distributed transaction and the job-receiving service.</p> <p>SP: as RE</p> <p>FI: terminates the distributed transaction as well as the job-receiving service and job-submitting service.</p> <p>FC: as FI</p> <p>RS: resets job-submitting and job-receiving service to the last synchronization point. Services which started in this transaction are terminated.</p> <p>ER/FR: terminates job-submitting and job-receiving service and resets the distributed transaction.</p>

Partner status		PENDING variants in the job-submitting service permitted variants and their effect and limitations
KCVGST / kcpcv_state	KCTAST / kcpta_state	
"C"	"C"	the output message must be addressed to another partner or to the client or (with PENDING SP/FC) to the follow-up program unit! KP: keep transaction in job submitter open. RE: terminates the transaction in the job submitter. SP: terminates the transaction in the job submitter. FI: terminates transaction and job-submitting service. FC: terminates transaction and job-submitting service. RS: resets the job submitter transaction to the last synchronization point. ER/FR: resets the transaction in the job-submitting service and terminates the job-submitting service. the job-receiving service is already terminated.

The combinations "KCVGST=O, KCTAST=C" and "KCVGST=C, KCTAST=C" cannot occur if you follow the bottom-up strategy (see [page 137](#)).

*PENDING variants in job-receiving service*

The calls PENDING PA and PENDING PR are not included because they have no special features for distributed processing. The variant PENDING FC is not permitted in job-receiving services.

Partner status		PENDING variants in job-receiving service permitted variants and their effects and limitations
KCVGST / kcpcv_state	KCTAST / kcpta_state	
"O"	"O"	KP: transactions in the job-receiving and job-submitting services remain open. RE: job-receiving services passes to transaction status P. FI: the job-receiving service changes to transaction status P. The service is terminated at the next synchronization point (=end of the distributed transaction). RS: resets the distributed transaction to the last synchronization point. Services in this transaction are terminated. ER/FR: resets the distributed transaction; the job-receiving service is terminated.

Partner status		PEND variants in job-receiving service permitted variants and their effects and limitations
KCVGST / kcpcv_state	KCTAST / kcpta_state	
"O"	"P"	RE: terminates the distributed transaction. SP: terminates the distributed transaction. FI: terminates the distributed transaction and the job-receiving service. RS: resets the distributed transaction to the last synchronization point. Services in this transaction are terminated. ER/FR: resets the distributed transaction; the job-receiving service is terminated.
"O"	"C"	KP: permitted RE: job-receiving service passes to transaction status P. SP: job-receiving service passes to transaction status P. The service is terminated at the next synchronization point (= end of the distributed transaction). RS: resets the distributed transaction to the last synchronization point. ER/FR: resets the distributed transaction; the job-receiving service is terminated.

If the bottom-up strategy is adhered to, only the combination "O"/"O" can occur, see [page 137](#).

Prior to a PEND RS in a follow-up transaction, you have to send a reset message with MPUT RM, otherwise openUTM aborts the job-receiving service with 83Z.

### 5.3.3 Existing program units as LU6.1 job receivers

You can use existing UTM program units, which were originally designed to communicate with terminals, unchanged as program units in a job-receiving service. Existing asynchronous programs can also be used as job receivers without any adaptations.

The same service can thus be used by terminals and client programs as well as by other services. In this way openUTM gives you considerable flexibility in application distribution.

If you want to use existing program units as job-receiving program units unchanged, or want to develop program units which can be used by terminals and client programs as well as by other services, you have to take account of the following points:

- Different return information in KB header

The communication partner of the job-receiving service is not the user at the terminal, but the job-submitting service. This is why the job-receiving service receives neither the ID of the terminal user nor the name of the LTERM partner in the KB header. (It is possible that neither are even generated in the application of the job-receiving service). Instead it receives the local session name (LSES name) and the local name of the partner application (LPAP name).

- Different assignments of TLS and ULS

Write and read calls for TLS in the job-receiving service refer to the TLS of the LPAP partner and not to a TLS of an LTERM partner. Similarly, calls for a ULS refer to the ULS of the session.

- Function keys are not supported in distributed processing

The job-submitting service cannot send a message corresponding to the function key to the job-receiving service. When using MGET the job-receiving service can therefore never receive the corresponding KDCS return code (19Z to 39Z).

- The card reader is not supported in distributed processing.

In a job-receiving service, the KCAUSWEIS/kccard field always contains blanks.

- No formatting in distributed processing via LU6.1

It is usually unimportant to a service whether it receives the dialog message from a terminal, an openUTM client program or from an LU6.1 partner. In the case of distributed processing via LU6.1, openUTM transfers the format ID specified in the job-submitting service with the MPUT call, but does not perform any formatting. The format identifier is also transferred with all message segments.

If you specify an incorrect format identifier in MGET, then openUTM operates when distributed processing via LU6.1 is used just like it does when segment formats are used for terminals: openUTM acknowledges an incorrect format identifier in MGET with the KDCS return code 03Z, and no messages or message segments are passed in the message area.

- Special structure of the job-submitting service (with distributed dialogs)

If existing program units of an application are to be used as program units in a job-receiving service, or if the job-receiving programs are programmed in such a way that they cannot evaluate the status indicators in the MGET call, the job-submitting service has to let itself be controlled by the job-receiving service with regard to transaction management. This means the bottom up strategy (see [page 137](#)) must be observed. To ensure this the job-submitting service must take account of the transaction status of the job-receiving service: end of transaction (PEND RE/SP) may not be set in the job submitter until all job receivers have transaction status "P".

### 5.3.4 Example: distributed dialog via LU6.1

The following simple example shows the sequence of calls in distributed dialogs via LU6.1. In each case there are indications as to which fields can or have to be set or evaluated. The field names are specified using COBOL notation.

In this example the job-submitting program consists of two program units: in the first unit, the subjob is submitted to the job-receiving service; in the second unit, the response of the job-receiving service is read and the response is output to the terminal, if necessary.

As explained in [section “Error handling by the program unit” on page 125](#) a job-submitting program unit which follows after a synchronization point is started not only in a normal run, but also after resetting a distributed transaction with service restart. This is why a check is made in the first unit of the job-submitting program whether status information is available. Such a service restart only occurs, however, if the job-submitting service has set a synchronization point prior to the start of this program unit (with or without participation of a job-receiving service).

#### 1. Job-submitting program, first unit

```

INIT
    Evaluate:
    KCKNZVG      →   R   service restart,
                    status information might be available.

MGET  Input message from terminal. If the job-submitting
      service has already reached a synchronization point,
      status information may be present as well.
      Evaluate:
      KCRPI      →   Return information from the job receiver
                    blanks there is no status information present
                    >coid  there is status information from the
                        job-receiving service with the specified
                        service ID. In this case the status
                        information is to be read with a 2nd MGET.

2. MGET
    Read the status information after which error handling is necessary.
    Set:
    KCOM        ←   with NT
    KCLA        ←   with the length 0
    KCRN        ←   with the service ID (>vgid)
    KCMF        ←   with blanks

    Evaluate:
    KCVGST      →   service status:
                    I   job-receiving service inactive
                    E   job-receiving service terminated with
                        PEND ER/FR
                    Z   job-receiving service terminated by openUTM
                        with PEND ER
                    R   job-receiving service terminated
                        with PEND RS
                    T   time has expired (timer)

```

KCTAST	→	transaction status:
I		transaction in job-receiving service inactive
R		transaction in job-receiving service reset
M		mismatch

If there is no status information present:

APRO Address the job-receiving service (if not already done)

Set:

KCOM	←	with DM for dialog service
KCRN	←	with the LTAC of the job-receiving service
KCPA	←	with double-step addressing:
		with the name of the job-receiving application
KCPI	←	with a self-selected service ID (>coid)
KCLM	←	0

MPUT to job-receiving service

Set:

KCOM	←	NT or NE
KCRN	←	with the service ID (>coid)
KCMF	←	possible format ID for job-receiver
KCDF	←	binary 0
KCLM	←	length

PEND (end of first unit)

Set:

KCOM	←	PEND call variant:
KP		normally recommended in the job-submitting service
RE		end of transaction is requested
FI		not allowed because job-receiver still open
ER/FR		also aborts job-receiving service
PA		prohibited after MPUT to job-receiving service
PR		prohibited after MPUT to job-receiving service
SP		prohibited after MPUT to job-receiving service
FC		prohibited after MPUT to job-receiving service
KCRN	←	name of follow-up program unit of job submitter (second unit of the job-submitting program)



**2. Job-receiving program**

INIT

Evaluate:  
 KCBENID → name of the session  
 KCLOGTER → name of the job-submitting application  
 KCTERMN → ID of the job-submitting application  
 KCCP → ID of the protocol used, '1' is entered for LU6.1  
  
 KCRMF → format ID from first MPUT of the job submitter

MGET Read message from job-submitting service

Evaluate:  
 KCRCCC → KDCS error code 19Z through 39Z cannot occur  
 KCRMLM → length from MPUT of the job submitter (KCLM)  
 KCRST Byte1 → service status of job-submitting service:  
     0 job-submitting service is open  
 KCRST Byte2 → transaction status of job-submitting service  
     0 transaction is open (PEND KP for job submitter)  
     P end of transaction initiated (PEND RE)  
     C transaction with job submitter terminated  
 KCRMF → if more message segments still present:  
     format ID of the next segment, otherwise:  
     format ID of segment read.

MPUT unchanged

Set:  
 KCOM ← with NT or NE  
 KCMF ← format ID or blanks  
 KCLM ← length of the message  
 KCRN ← with blanks to send the message to the job submitter  
 KCDF ← any value which the job submitter receives  
     with MGET

PEND End of job-receiving program unit

Set:  
 KCOM ← depending on the transaction status:  
     KP only allowed with KCTAST=0 or C  
     RE terminates the transaction with KCTAST=P or  
     initiates end of transaction with KCTAST=0 or C  
     FI end of the job-receiving service, otherwise  
     as PEND RE  
     ER/FR end of the job-receiving service,  
     transaction is reset, job submitter is informed  
     PA/PR no special points, cannot be used to send  
     a message to the job submitter  
 KCRN ← if necessary (with PEND KP or PEND RE), name of the  
     follow-up program unit of job-receiving service

**3. Follow-up program unit of the job-submitting program (second unit)**

INIT

Evaluate:  
 KCRPI ← service ID of the job-receiving service  
 KCRMF ← format ID from 1st MPUT of the job receiver

MGET Read message from the job receiver

Set:  
 KCOM ← NT  
 KCLA ← length of the message area  
 KCRN ← service ID from KCRPI of the INIT call  
 KCMF ← format ID from KCRMF of the INIT call

Evaluate:  
 KRMLM → actual length of the input message  
 KCRMF → if more message segments present:  
 format ID of next segment, otherwise:  
 format ID of segment read.  
 KCRDF → value from the relevant MPUT of the  
 job-receiving service.  
 KCRPI → service ID if further message  
 segments present  
 KCRST Byte1 → service status of job-receiving service:  
 0 job-receiving service is open  
 C job-receiving service is terminated  
 (PEND FI)  
 KCRST Byte2 → transaction status of job-receiving service:  
 0 transaction is open (PEND KP)  
 P job receiver has requested end of transaction  
 (with PEND RE or FI, PTC status)  
 C transaction is terminated (PEND RE or FI)

MPUT to the terminal

Set:  
 KCRN ← with blanks  
 KCOM ← with NT or NE  
 KCLM ← with the length of the message  
 KCMF ← with the format ID or blanks  
 KCDF ← if necessary, with a screen function

PEND end of the follow-up program unit of the job-receiving service

Set:  
 KCOM ← depending on status indicators in KCRST  
 FI only allowed with KCVGST=C, terminates  
 service and transaction  
 RE terminates the transaction with KCTAST=P,  
 not allowed with KCTAST=0 (because message  
 directed to terminal)  
 ER/FR resets transaction, job-receiving service  
 is also reset and terminated;  
 only exception: KCTAST=C and KCVGST=C.  
 KP not recommended if KCTAST=P  
 PA/PR prohibited since message was sent to  
 terminal.  
 KCRN ← if necessary (with PEND KP or PEND RE), name of the  
 follow-up program unit of  
 the job-submitting service

## 5.4 Distributed dialogs via OSI TP

The ISO (International Organization for Standardization) defined the OSI TP protocol (**O**pen **S**ystems **I**nterconnection **T**ransaction **P**rocessing) for distributed processing between applications.

OSI TP is part of level 7 of the OSI reference model. It was accepted as an international standard under the identification ISO/IEC 10026 in July 1992. In particular, OSI TP allows you to control the processing of distributed, i.e. inter-system, transactions. However, you can also use this protocol in cases where two applications simply exchange data without any transaction management. These types of application are often used in client/server communication.

### 5.4.1 Functional units

The OSI TP functions are divided into so called functional units (FU). Depending on the requirements placed on communication with a partner application, individual functions can be selected for the communication. openUTM supports the following functional units:

#### **Dialogue**

The functional unit Dialogue is required whenever you communicate via the OSI TP protocol. It contains functions for the establishment and disconnection of dialogs and as well as for sending error messages.

You use the KDCS call APRO to establish dialogs. In the APRO call you select the OSI TP function combinations which are used for the dialog. Dialogs are normally terminated with a PEND FI call. Dialogs are abnormally terminated by the PEND ER or CTRL AB call. MPUT EM triggers the protocol element TP-U-ERROR, CTRL AB or PEND ER the protocol element TP-ABORT.

#### **Polarized Control**

You use the functional unit Polarized Control to manage the send authorization for a dialog. Each dialog is assigned a send authorization which only one of the communication partners can possess at any one time.

In UTM services, send authorization for a dialog changes at the end of the processing step when a message is sent to the dialog partner: openUTM creates the protocol element TP-GRANT-CONTROL implicitly.

## Handshake

The handshake functions can be used by the communication partners to coordinate the processing of a dialog at application level. This function makes it possible to request processing confirmations and send positive or negative confirmations. No inter-application transaction management is linked to this function.

You can create a handshake request by calling MPUT HM. Handshake requests from the partner application are displayed by calling MGET. When the KDCS interface is used the messages are not sent until the send authorization is transferred. For this reason openUTM only creates the OSI TP protocol element TP-HANDSHAKE-AND-GRANT-CONTROL, and not TP-HANDSHAKE.

UTM implicitly sends a positive confirmation of a handshake request before the next message to the partner from which the request has been received. However, the confirmation is sent at the next end of transaction at the latest.

You use MPUT EM to send a negative confirmation of a handshake request. The requesting service can read the result of a handshake request with an MGET call.

## Commit and Chained Transactions

The Commit functional unit provides the functions necessary to create distributed transactions. These are, in particular, functions to set forward and reset distributed transactions. If you use these functions you must always select the functional unit Chained Transactions. If processing is to be performed with global transaction management, then only distributed transactions are processed for this dialog.

The MPUT, CTRL and PEND/PGWT calls are used in connection with these function groups.

The operation modifiers of the PEND/PGWT calls, in combination with the target of the MPUT messages created during the last processing step, determine whether a TP-PREPARE is sent and, if this is the case, whether it is sent with DATA-PERMITTED=TRUE or FALSE. However, you can also use the CTRL PR call to create a TP-PREPARE.

The OSI TP protocol elements TP-DEFER(GRANT-CONTROL) and TP-DEFER(END-DIALOGUE) are triggered in the same way. The latter can also be created on its own using the CTRL PE call.

You use the PEND call with the appropriate operation modifiers or the PGWT CM call to request an end of transaction. openUTM negotiates the protocol for the processing of the two-phase commit without the participation of the application program unit. A distributed transaction can be reset using PEND RS or PGWT RB. PGWT RB **must** be used if the previous transaction was terminated with PGWT CM. openUTM handles the protocol for resetting the distributed transaction without the participation of the application program unit.

Following an MGET call, heuristic decisions by communication partners are shown in the transaction status.

### **Recovery**

The Recovery functional unit provides the services which are necessary for resynchronization of the interrupted transaction after a communication failure. This functional unit ensures global data consistency in such cases. However, OSI TP does not permit the continuation of an interrupted connection (dialog restart).

UTM uses the services of the Recovery functional unit internally. They cannot be directly accessed by the application program.

## **5.4.2 Programming aids**

A program unit receives information about its communication partners via different displays. This information enables the program unit to react selectively to special situations. openUTM makes the information available after an INIT or MGET call.

After INIT, the job receiver is informed about being called by an OSI TP partner and the OSI TP functions used by the job submitter for the dialog. Further displays after the INIT call inform the job receiver of whether the job submitter has requested end of transaction or dialog and whether another message has to be sent to the job submitter in the current transaction.

After an INIT or MGET call, the program unit run is informed about the communication partner for which a message is present for reading and about the abstract syntax which was used when sending the message. When receiving the messages, the program unit has to adhere to the predefined order of messages specified by openUTM.

Through the MGET call, the program unit is informed of the type of message received and receives information about the service and transaction status of the communication partner.

The information available after INIT or MGET is explained in more detail below.

### Displaying the selected OSI TP functions for the dialog

After the INIT call, important information is entered in the KCCP and KCOF1 fields in KBKOPF (the KB header).

KCCP contains the communication protocol used by the partner: in the case of OSI TP, '2' is entered here. This tells the program unit that it has been called by an OSI TP job submitter.

KCOF1 contains information about the OSI TP functions available for the dialog with the job submitter. The values in the KCOF1 field have the following meanings:

- B      Basic functions  
The functional units Dialogue and Polarized Control are selected for the dialog with the job submitter.
- H      Basic and handshake functions  
The functional units Dialogue, Polarized Control and Handshake are selected for the dialog with the job submitter.
- C      Basic and Commit functions with Chained Transactions  
The functional units Dialogue, Polarized Control, Commit and Chained Transactions are selected for the dialog with the job submitter.
- O      (other combination)  
A standard combination was not selected for the dialog with the job submitter. If INIT PU was called and OSI TP information requested, the available OSI TPI functions are displayed in the message area.

### Requesting end of transaction or service by job submitter

After an INIT PU call, the KCENDTA field in the message area for the job-receiving service indicates whether the local service has been requested by its job submitter to terminate the transaction and which variant of the PEND call is to be used. The local service must respond to the request to terminate the transaction or dialog at the latest by the end of the processing step in which it next sends a message to the job submitter.

The following values are possible:

- \_      no instructions concerning the termination of the processing step.
- O      no end of transaction may be requested at the end of the processing step.
- R      the transaction and the dialog step must be concluded, the service may not be terminated (PEND RE or PGWT CM with a preceding MPUT to the job submitter). At the end of transaction, the job submitter possesses the send authorization in the job submitter dialog. The local service possesses the end-of-transaction send authorization in all other dialogs.

- S the transaction must be concluded, the dialog step must not be terminated (PEND SP or PGWT CM without a preceding MPUT to the job submitter).  
The local service possesses the send authorization in the dialog with the job submitter at the end of the transaction.
- C the transaction must be concluded, the service may not be terminated (PEND RE/SP or PGWT CM). The local service has end-of-transaction send authorization for the dialog to the job submitter. In another dialog, end-of-transaction send authorization may be passed to the job receiver. This is performed by issuing an MPUT to a job receiver, followed by PEND RE or PGWT CM.
- F both the transaction and the service must be concluded (PEND FI).

### Displaying the send authorization for the dialog with the job submitter

After an INIT PU call, the KCSEND field indicates in the message area in the local service whether the local service may send a message to the job submitter in the current processing step. The following values are possible.

- Y It is necessary to send a message to the job submitter at the end of the dialog step.  
If KCENDTA has the value "S", in this case it is also necessary to send a message to the job submitter at the end of transaction. This combination (KCENDTA=S and KCSEND=Y) can only occur in the case of heterogeneous coupling.
- N No messages are permitted to be sent to the job submitter. However, messages may be sent to job receivers in which case the transaction must remain open after the end of the processing step.

### Displaying the type of message received

After an MGET call, the type of message received is displayed in the KCRMGT field of the return area. The following values are possible:

- C (confirm)  
A positive handshake confirmation has been received.
- E (error)  
An error message or negative handshake confirmation has been received.
- H (handshake)  
A handshake request has been received.
- M (message)  
A normal user message has been received.

### Service status

Following an MGET call, the service status of the communication partner from which a message has been received is displayed in the KCVGST/kcpcv\_state field of the return area. The local service can use this display to draw conclusions about the dialog with this partner.

The following values are possible:

- C (closed)  
The job submitter has terminated the service.
- D (disconnected)  
The communication with the job submitter has been terminated because of loss of connection.
- I (inactive)  
The job-receiving service could not be started because, for example, the TAC is unknown.
- O (open)  
The partner service is open, i.e. end of dialog has not yet been requested.
- P (pending end dialogue)  
This status can only occur in the case of heterogeneous links and in dialogs for which the Commit functionality has not been selected.  
The job receiver wants to end the communication. If the job submitter does not agree, it can continue the communication using MPUT EM.
- T (time out)  
The job receiver has not sent a message within the generated wait period; the dialog with the job receiver has been terminated.
- Z (error)  
The dialog with the job receiver has been terminated because of an error.

In the case of the service stati D, I, and T no message is transferred. The service status in the job-receiving service is always O.



## Transaction status

Following an MGET call, the transaction status of the communication partner from which a message has been received is indicated in the KCVGST/kcpta\_state field of the return area. The local service can use this display to draw conclusions about the dialog with this partner.

The following values are possible:

- H (heuristic hazard)  
The result of a transaction is undetermined since communication with at least one communication partner has been interrupted. The possibility that one of the communication partners involved in the last transaction has made a heuristic decision which conflicts with the actual result of the last transaction cannot be excluded.
- I (inactive)  
The transaction is inactive at the job receiver, e.g. because the TAC is invalid or no connection could be established in the generated wait period.
- M (mismatch)  
It was not possible to synchronize the transaction in the remote service with the transaction in the local service. This may occur after a timeout.  
A mismatch can also occur if at least one of the communication partners involved in the transaction has made a heuristic decision which conflicts with the actual result of the transaction.
- O (open)  
The transaction is open in the remote service.
- P (prepare to commit)  
The partner service has either initiated the end of transaction itself or is requesting the local service to initiate the end of transaction.
- R (reset)  
The transaction in the remote service has been reset.
- U (unknown)  
The transaction status is unknown. This value is only possible in dialogs for which the Commit functionality has not been selected.

In the job-receiving service, only the following transaction stati are possible:

**with** functional unit commit: O, P

**without** functional unit commit: U

### 5.4.3 Programming rules for dialogs without the functional unit commit

#### End of transaction

In a communication which complies with Cooperative Processing, the communication partners may request end of transaction independently of each other. In this type of processing only local transactions occur, i.e. end of transaction in a service has no effect on the transaction in the partner service.

#### End of service

A job-receiving service may terminate its service at any time.

A job-submitting service may not terminate until all dialogs with its job-receiving services are terminated.

PEND FC (chained services) in job-receiving services is not permitted.

### 5.4.4 Programming rules with the functional unit commit

Before explaining the rules which a service must respect in a distributed transaction, it is first necessary to explain certain terms.

#### Explanation of terms

##### *Data transfer phase*

A service is in the data transfer phase until it has either been requested to end the transaction or has requested its job receivers to end the transaction.

##### *Send authorization*

During the data transfer phase, a send authorization exists for each dialog. This is assigned to one or other of the communication partners at any given time.

The service which has send authorization in a dialog may use MPUT to send a message to the partner service. When the message is sent, the send authorization passes to the communication partner.

In a processing step, a service can pass the send authorization for the dialog with the job submitter or pass one or more send authorizations for dialogs with job receivers.

*End-of-transaction send authorization*

The end-of-transaction send authorization controls which of the communication partners owns the send authorization after the current transaction is finished. Usually the job-submitting service owns the end-of-transaction send authorization. However, the job-submitting service can use an MPUT message followed by PEND RE to pass this send authorization to the job-receiving service. Here you have to remember that a service can only pass on send authorization for a maximum of one dialog at the end of the transaction.

**End of transaction**

A service may request end of transaction if it has been requested to do so by its job-submitting service, and it does not possess end-of-transaction send authorization for more than one dialog.

**End of service**

A service may request end of service if it has been requested to do so by its job-submitting service, and the service has not sent a message to a job receiver in the current processing step.

**Other programming rules**

- In the data transfer phase, the local service possesses the send authorization for all dialogs during a program unit run.
- In a processing step, a service cannot simultaneously send messages to its job submitter and to job receivers.
- If the transaction remains open at the end of the processing step, then the local service may send messages to multiple job receivers simultaneously in this processing step.
- A service may pass the end-of-transaction send authorization to no more than one partner. The consequences of this rule are:
  - In a processing step which is terminated by a request to end a transaction, a message may be sent to only one partner.
  - An intermediate node may only transfer the end-of-transaction send authorization in a dialog with a job receiver if it possesses the end-of-transaction send authorization in the dialog with the job submitter.
- A job receiver may only request end of transaction if it has been requested to do so by its job submitter.

**Rules for using the different PEND variants**

- You can use PEND KP if, in the processing step, messages are only sent to partners which have not yet requested end of transaction.
- You can use PEND RE if, in the processing step,
  - messages have been sent to no more than one partner.
  - no request for end of transaction or dialog has simultaneously been sent to this partner.
  - the local service has already received a request for end of transaction or the local service itself is the root in the transaction tree.
- You can use PEND SP if, in the processing step,
  - the local service has already received a request for end of transaction or the local service itself is the root in the transaction tree and
  - the local service possesses the end-of-transaction send authorization for the dialog with the job submitter and
  - no message has been sent to a job receiver and
  - no message has been sent to the client.
- You can use PEND FI if, in the processing step,
  - the local service has already received a request for end of dialog or the local service itself is the root of the transaction tree and
  - no job receiver has received a request (using CTRL PR) for end of transaction, but not for end of service.
  - no message has been sent to a job receiver.

**Rules for using PGWT variants**

- A PGWT KP is possible if PEND KP is allowed.
- A PGWT CM is possible
  - with output of a dialog message if PEND RE is allowed.
  - without output of a dialog message if PEND SP is allowed.
- PGWT RB must be used if a transaction in which the last synchronization point was set with PGWT CM is to be reset without terminating the OSI TP dialog.

### Programming recommendations

In distributed transaction processing via OSI TP protocol, you should preferably terminate the distributed transaction in the following way:

If the uppermost job submitter in the transaction tree wants to terminate the distributed transaction, it issues a CTRL PR and an MPUT call to each of its job receivers and subsequently uses PEND/PGWT KP to terminate the processing step. The uppermost job submitter terminates the transaction after the responses from its job receivers have been received.

If an intermediate node in the transaction tree receives a request to end the transaction, it issues a CTRL PR and an MPUT call to each of its job receivers and subsequently uses PEND/PGWT KP to terminate the processing step. After the responses from its job receivers have been received, the intermediate node sends a response to its job submitter and terminates the dialog step and the transaction.

If the lowest job receiver in the transaction tree receives a request to end the transaction, it sends a response to its job submitter and terminates the dialog step and the transaction.

If you follow this rule, the next transaction starts at the uppermost job-submitting service in a program unit run.

### 5.4.5 Programming rules for communications with BeanConnect

If openUTM calls an OLTP message driven bean via BeanConnect at an J2EE server, then you should note the following:

- The handshake functional units must not be selected.
- Only single-step dialogs are permitted, i.e. in the case of dialogs without functional unit commit, the job receiver terminates the dialog after sending the response.
- In the case of dialogs with functional unit commit, the job submitter must request the receiver to terminate the dialog either immediately on sending the message or on receiving the response. If the job submitter does not request dialog termination until receiving the response then, if an error occurs, the job receiver is still able to send an (error) message to the submitter and the submitter itself can reset the transaction.

## 5.4.6 Particularities of rollback and restart

The OSI TP protocol enables you to work either with global transaction management (functional unit commit) or without global transaction management (Cooperative Processing).

### OSI TP with functional unit commit

If you select the functional unit commit for OSI TP, the resetting of transactions (rollback) with PEND RS is performed in the same way as in LU6.1.

When a service is restarted after PEND RS, the behavior is the same as with LU6.1, with one exception: when using the OSI TP protocol it is not possible to restart an interrupted dialog.

A synchronization point can also be set with PGWT CM when using OSI TP. The next transaction may then be rolled back with PGWT RB only. In this case a return is always made to the program unit that issued the PGWT RB. No service restart is performed.

### OSI TP without functional unit commit

If you do not select this functional unit, the job-submitting service is **not** automatically reset when an error occurs in the job-receiver service or if the connection is lost. The job-submitting service is continued with the program unit specified in the last PEND. However, if it waits for a result from the job receiver, it receives an error message (with service status "Z" and transaction status "U") and can also react with a reset.

If an error occurs in the job-submitter service, openUTM resets the transactions in the job-submitting service and job receiving services and terminates the job-receiving service. If the job-submitting service has already reached a synchronization point, a service restart is performed after the reset of the transaction. The follow-up program unit receives an error message with service status "Z" and transaction status "U".

A global service restart is not possible, since no common synchronization points exist.

If you use calls for programmed resets, you have to take the following into account:

- PENDING

For a call in the job-submitting service:

All job-receiving services with which the job-submitting service communicates without functional unit commit are terminated.

For a call in the job-receiving service:

- If PENDING SP has been used to terminate the preceding transaction, then PENDING RS resets the local transaction and the service is continued with the follow-up program unit specified with PENDING SP.
- If PENDING SP has **not** been used to terminate the preceding transaction **and the service** is running under a user ID without the restart property, then the service is rolled back to the last synchronization point and the dialog with the job submitter is terminated.
- In all other cases, openUTM terminates the service with PENDING FR.

- PGWT RB

PGWT RB rolls back the current transaction and the program unit continues.

- PENDING ER/FR

No special considerations apply when this call is used in the job-submitter service: The transactions in the job submitter and all its job receivers are reset and the job receivers terminated.

If called in the job-receiver service, the job receiver is reset and terminated. However, the job-submitter service is continued in the program unit specified with the last PENDING. The job-submitter service can read the messages which have been sent by the job receiver with MPUT.



To ensure consistent operation even when a transaction is reset, it is advisable to use only PGWT or only PENDING calls within a distributed transaction.

- RSET

The RSET call always applies only to the local service. The RSET=GLOBAL setting in the KDCDEF statement UTMD has no effect. This setting only has an effect in distributed processing with global transaction management (see [page 127](#)).

### 5.4.7 Using existing program units for OSI TP communication

Existing UTM program units which were not specifically designed for communication via OSI TP can be used unchanged for OSI TP communication given certain restrictions (see below). The same service can, for example, be used by clients (e.g. terminals or UPIC client programs) as well as other services. In this way openUTM gives you considerable flexibility when distributing applications.

The various cases which may arise when using existing program units for OSI TP communication are identified and described in the following sections.

#### Program units for communication with clients as OSI TP job receiver

Program units which have been designed for communication with clients (e.g. for communication with terminals or UPIC client programs) can be used unchanged by the job receiver for communication with an OSI TP partner. You must observe the following points:

- Different return information in KB header

The communication partner of the job-receiving service is the job-submitting service, not the user at the terminal. This is why the job-receiving service does not receive the name of the LTERM partner in the KB header. Instead, it receives the name of the OSI-LPAP partner. The entry in the KCBENID/kcuserid field depends on the security type used. You use the APRO call to select the security type in the job submitter (in the KCSECTYP field):

- With security type "N" (None), no user ID is transferred to the job receiver. KCBENID/kcuserid contains the name of the association instead of the user ID.
- With security type "P" (Program), KCBENID/kcuserid contains the user ID which was specified through APRO in the job submitter.
- With security type "S" (Same), KCBENID/kcuserid contains the user ID under which the job submitter was started.

- Different TLS and ULS assignments

Write and read calls for TLS in the job-receiving service refer to the TLS of the OSI-LPAP partner and not to a TLS of an LTERM partner. Similarly, calls for a ULS refer to the ULS of the user ID only if security type P/S is present. If security type N is present it refers to the ULS of the association.

- Function keys are not supported in distributed processing

The job-submitting service cannot send a message corresponding to the function key to the job-receiving service. When MGET is used, the job-receiving service can therefore never receive the corresponding KDCS return code (19Z to 39Z).



- The card reader cannot be used in the job-receiving service.  
In a job-receiving service, the KCAUSWEIS/kccard field always contains blanks.
- Abstract syntax for distributed processing via OSI TP  
In distributed processing via OSI TP, the format identifier is used to transfer the name of the abstract syntax.  
Job receiver programs designed for communication with clients can only be used unchanged for distributed processing via OSI TP if you use UDT Octet String Mapping exclusively as the abstract syntax. Using any other abstract syntax you would have to perform adaptations for the encoding or decoding of messages. The KCMF/kcfn field must therefore always contain blanks.

### **LU6.1 job receiver as OSI TP job receiver**

Job receiver program units written for communication via LU6.1 can only be used unchanged as OSI TP job receivers when the commit functional unit has **not** been selected and the when the following conditions are fulfilled:

- You use UDT Octet String Mapping exclusively as abstract syntax for communication via OSI TP. The KCMF/kcfn field must therefore always contain blanks when exchanging messages.
- The transaction and service status in the job receiver programs are not evaluated.

If the commit functional unit is selected, then the job submitter must be the first to request the end of the transaction and the end of the service, and must also always issue the request when the job receiver expects it.

### **LU6.1 job submitter as OSI TP job submitter**

Job submitter program units written for communication via LU6.1 cannot be used as OSI TP job submitters unchanged. At the very minimum, you will have to adapt the APRO call (selection of OSI function in KCOF field and possible changes in the 2nd parameter area). No adaptations are necessary for distributed processing without global transaction management (Cooperative Processing).

In distributed processing with global transaction management, i.e. when the functional unit commit is selected, the programs must always be extended to take account of end-of-dialog requests (e.g. by inserting CTRL PE).

### 5.4.8 Particularities with heterogeneous coupling

If you want to use OSI TP to connect your UTM applications with transaction applications of other manufacturers you have to consider the following points:

- User data when establishing associations:  
When the connection is being established, only the user data necessary for OSI TP and CCR is exchanged. Other user data is not sent and is ignored when received.
- User syntaxes and CCR association setup:  
openUTM does not permit syntaxes generated by openUTM to be rejected by the Additionally, these syntaxes must be offered with the association setup request. In such cases, openUTM rejects the association setup.
- Disconnection with A-ABORT:  
openUTM uses A-ABORT to disconnect, not A-RELEASE.
- Channels:  
"Two-way-recovery" channels are not supported by openUTM.
- User data for TP-BEGIN-DIALOGUE-RI:  
You can only use TP-BEGIN-DIALOGUE-RI to exchange user data necessary for UTMSEC. Application programs have no direct access to this user data. Other user data is not sent and is ignored when received.
- No user data for TP-BEGIN-DIALOGUE-RC, TP-ABORT-RI:  
openUTM does not send user data with the protocol elements. Received user data is ignored.
- No Shared Control functional unit:  
openUTM does not support the Shared Control functional unit (i.e. does not support the profiles ATP12, ATP22, ATP32).
- Unchained Transactions functional unit:  
openUTM does not support this functional unit as a job submitter. Conversely, you can select the functional unit in a UTM job receiver application. However, the job submitter has to start the distributed transaction before the first send authorization transfer in the dialog. The dialog must terminate with the first transaction, as otherwise openUTM terminates the dialog abnormally.
- Recipient TPSU-Title:  
A Recipient TPSU-Title is always necessary when using TP-BEGIN-DIALOGUE-RI. If openUTM is the receiver, the title must not exceed 8 characters in length and cannot be of the type "integer".

- REQUEST-CONTROL-RI, HANDSHAKE-RI:  
openUTM does not send the protocol elements. If a dialog service receives TP-HANDSHAKE-RI it is terminated abnormally.
- maximum user data length: 32767 octets:  
openUTM sends a maximum of 32767 octets of user data in a protocol element. If user data with a length of more than 32767 octets is received, openUTM disconnects the link.
- End of dialog without Commit:  
No TP-END-DIALOGUE-RI should be sent to a dialog job-receiving service (end of dialog from "above"), as otherwise openUTM terminates the service abnormally. openUTM only uses "Confirmed End Dialogue" for the transfer of asynchronous messages.

## 5.4.9 Examples: distributed dialogs via OSI TP

This section contains examples describing the different programming interface possibilities when you use the Commit functionality and Chained Transactions.

First the simplest case is considered. Here, a job submitter communicates with a job receiver. Secondly, the scenario is extended to describe communication with multiple job receivers and finally the more complicated cases are illustrated in which a service communicates with both a job receiver and a job receiver via the OSI TP protocol.

The data transfer phase, end of transaction, and end of dialog are considered separately.

At the end of the section ([page 188ff](#)) you can find examples which illustrate the abnormal termination of job-receiving services with CTRL AB.

### Comments on the following diagrams

The following diagrams illustrate the communication flow in distributed OSI TP dialogs. In these services, only those KDCS calls which are relevant for communication are shown, other KDCS calls and processing statements are omitted.

The service and transaction stati, displayed to the program unit after the MGET call, are shown to the right of the MGET calls. In the examples, the COBOL field names are used, i.e. KCVGST for the service status and KCTAST for the transaction status. For C/C++ the corresponding fields are called *kcpcv\_state* and *kcpta\_state*.

In the job-receiving services, the KCENDTA and KCSEND are shown to the right of the INIT calls. These can be evaluated by a program unit after an INIT PU call.

In the case of MPUT calls addressed to a job receiver and CRTL calls, the name of the addressed job-receiving service is shown. The format used is ">x", where x represents the job receiver. MPUT calls without this specification are always addressed to either the job submitter or the client.

The arrows between job submitters and job receivers symbolize for the message exchange and protocol flow.

Synchronization points are represented by bold, unbroken lines.

Any PEND KP calls shown in these examples can be replaced by PGWT KP calls. Similarly, a PGWT CM with a preceding MPUT message can be used instead of PEND RE, and a PGWT CM without a preceding MPUT message can be used instead of PEND SP.

### 5.4.9.1 One job receiver

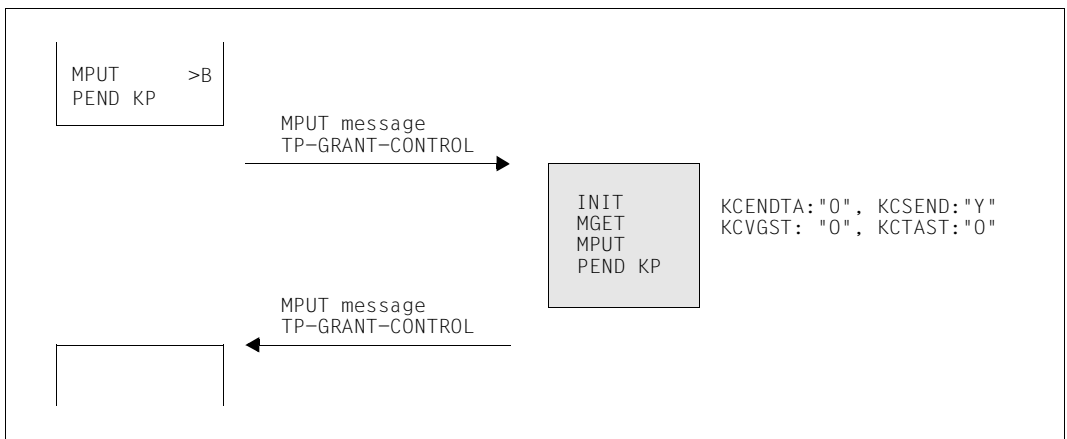
The simplest case possible is where one job submitter (A) has exactly one job receiver (B). This section explains all possible uses for this type of application.

#### Data transfer phase

The message transfer phase is the period in which neither of the two partners requests an end of transaction. For the job receiver, this option is only available if it has been requested explicitly by the job submitter.

Send authorization changes with each message sent to the partner. Displaying the service status "O" and transaction status "O" informs the job receiver that neither end of transaction nor end of service has been requested.

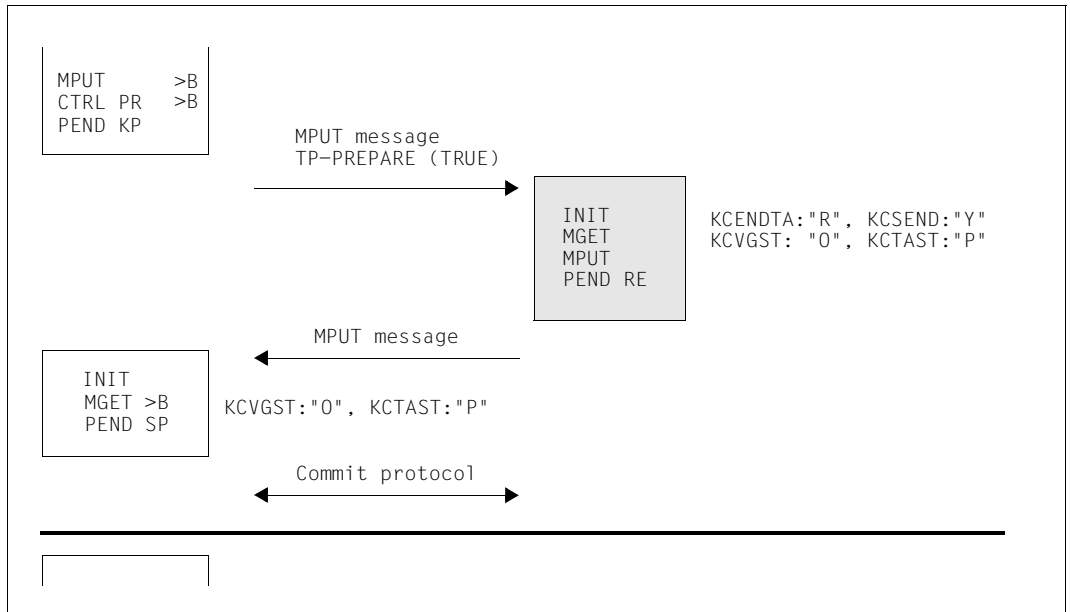
*Example 1: Message to job receiver and PEND KP*



**End of transaction**

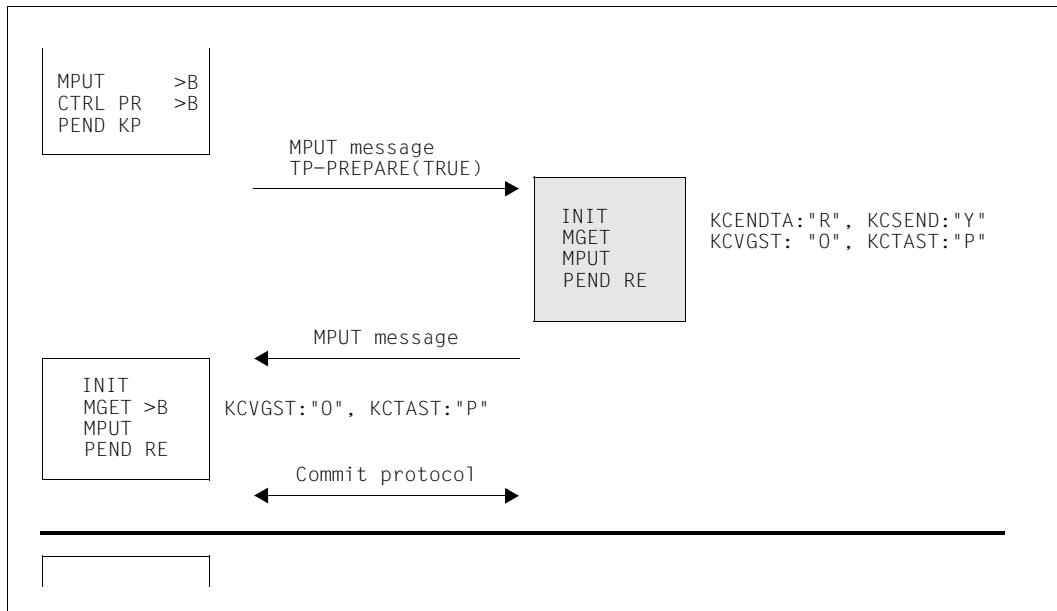
A job receiver may only issue an end of transaction call if this has been requested to do so by the job submitter. The job receiver can read this information from the transaction status after the MGET call.

*Example 2: Message and Prepare to job receiver and PEND KP*

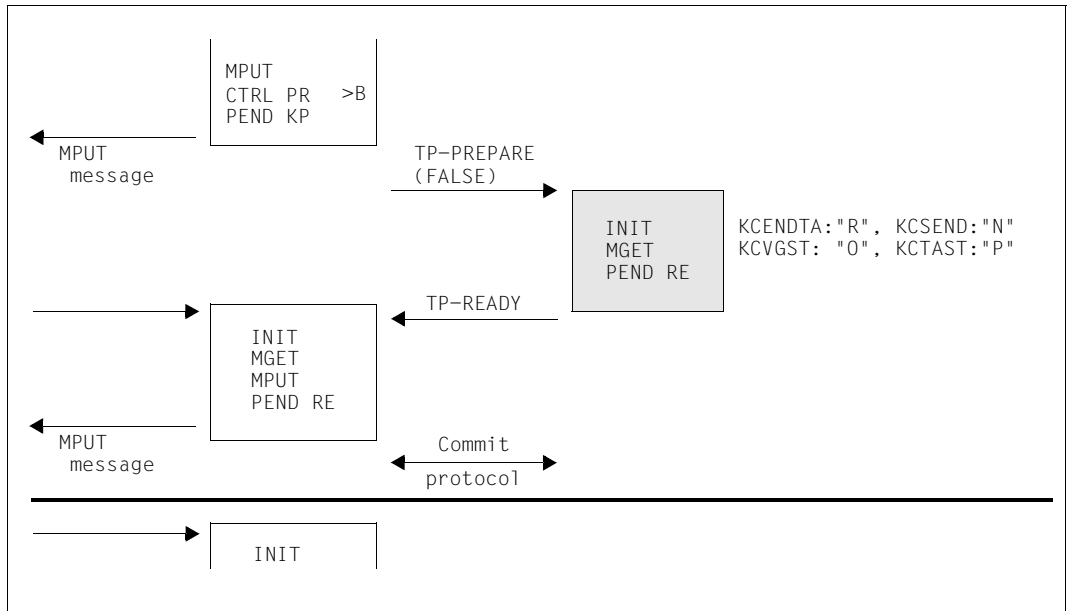


This should be regarded as the normal case when communicating via the OSI TP protocol, since the order of KDCS calls best corresponds to the protocol flow. Additionally, the job-submitting service has control after end of transaction and this simplifies service restart.

In the second job submitter program unit run you can also issue an MPUT to the client and a PEND RE in a dialog service instead of PEND SP. In this case, the command sequence is as follows:



*Example 3: No message to the job receiver and CTRL PR*



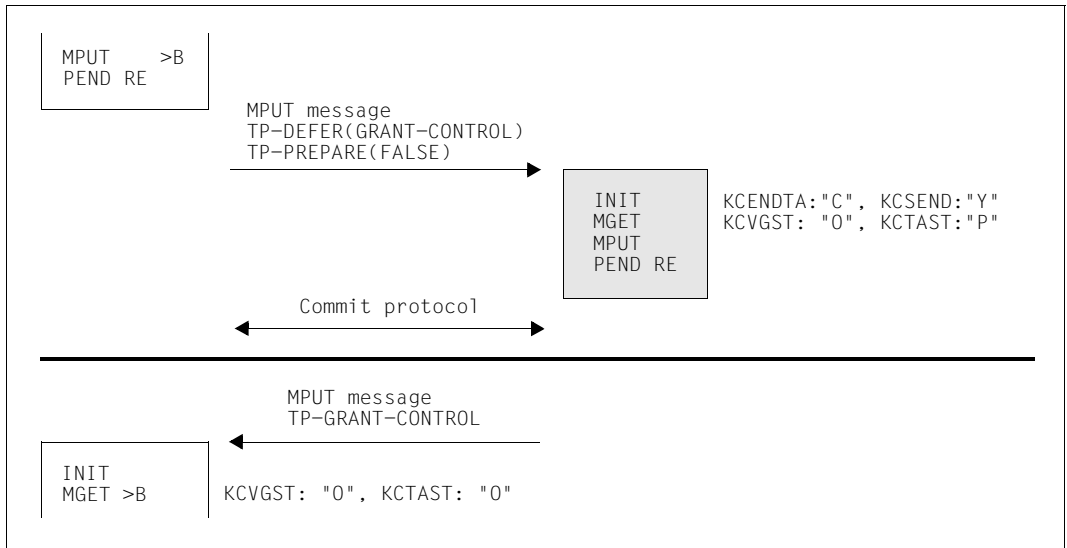
In this case the job receiver simply receives a request for end of transaction, but no message from the job submitter. Send authorization therefore does not pass to the job receiver (DATA-PERMITTED=FALSE). The job submitter possesses the send authorization at end of transaction - as in example 2.

It is also only possible to issue a PEND PA/PR in place of the MPUT, PEND KP the first time the program unit is run. If the job-submitting service is an asynchronous service, only this second variant is possible.

It is also only possible to issue a PEND SP in place of the MPUT, PEND KP the second time the program unit is run. If the job-submitting service is an asynchronous service, only this second variant is possible.



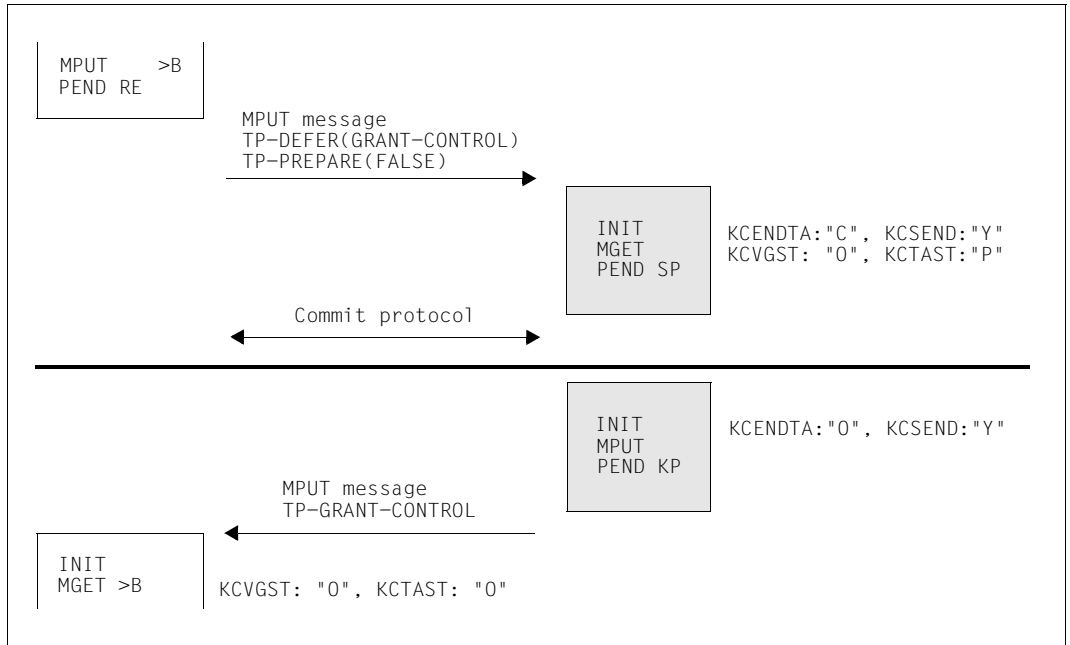
Example 4: Message to job receiver and PENDING



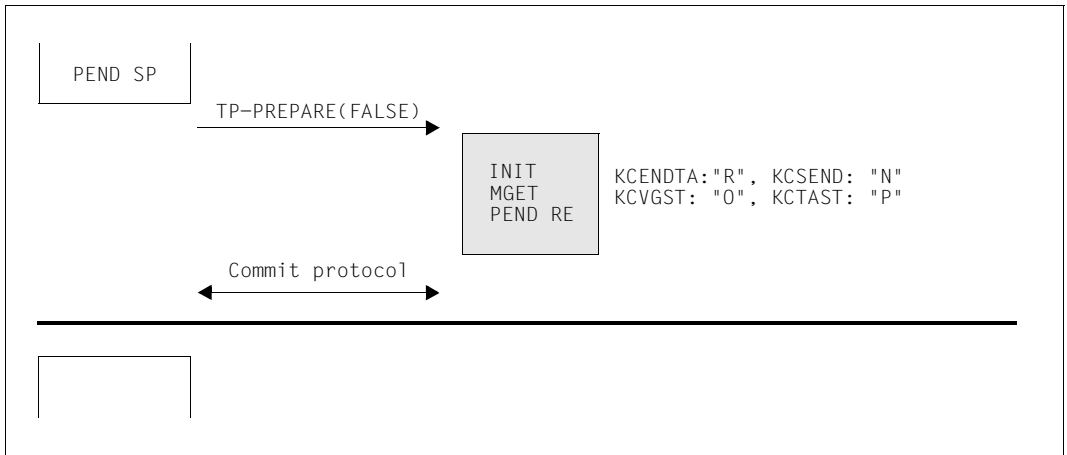
This case most closely resembles the situation when using the LU6.1 protocol. In particular, the service and transaction stati are identical to those for communication via the LU6.1 protocol. This means that you can reuse these programs unchanged. However, you should remember that these programs do not adhere to the bottom up strategy recommended for LU6.1 communication.

In this example, the job receiver possesses control of end-of-transaction send authorization.

In this case, the job receiver can also use a PEND SP instead of the PEND RE call. The command sequence is then as follows:

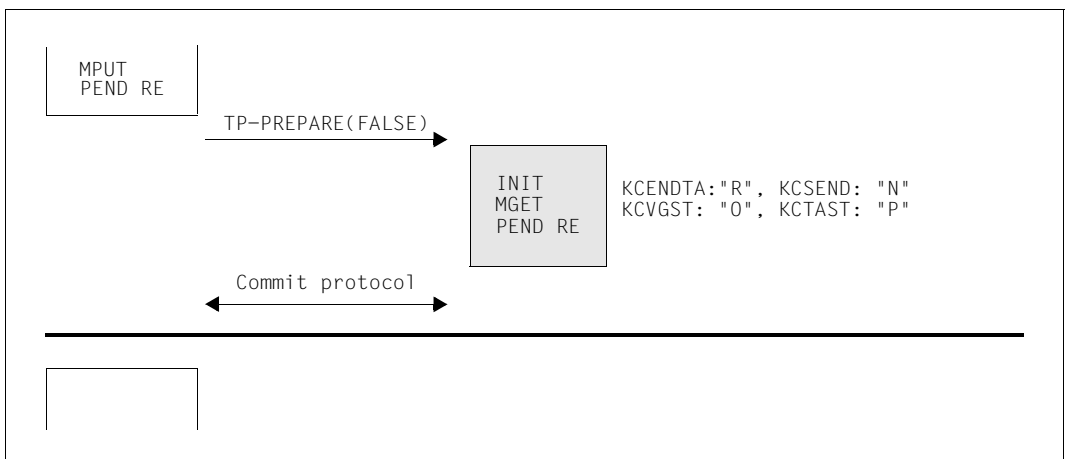


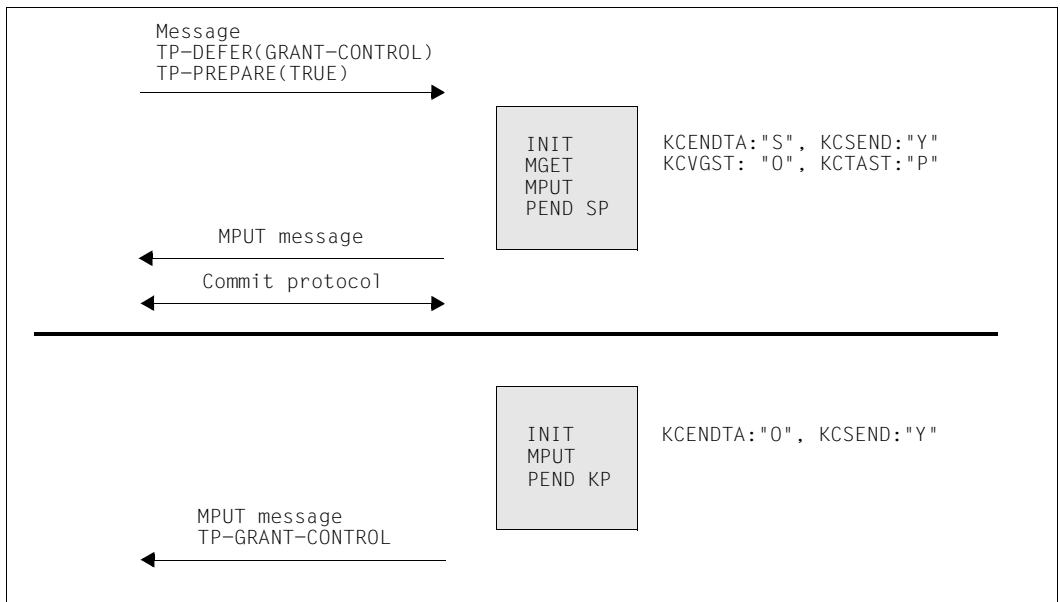
Example 5: No message to job receiver and PENDING SP/RE



This example illustrates that the job-receiving services are included in all transactions, even if the job submitter and job receiver have not communicated in the last transaction. There are no so-called local transactions when using the OSI TP protocol with Chained Transactions. This has to be taken into account when you design your distributed applications.

The job submitter can issue an MPUT to the client and a PENDING RE instead of PENDING SP. The command sequence is then as follows:



*Example 6: Defer-Grant-Control and Prepare(True)*

This is an 'exotic' case which can only occur with heterogeneous coupling. The job receiver has to send two messages in sequence to the job submitter. The first message has to be sent within the first current transaction. Send authorization remains with the job receiver after the end of transaction, which means that the job receiver issues the first message in the follow-up transaction.

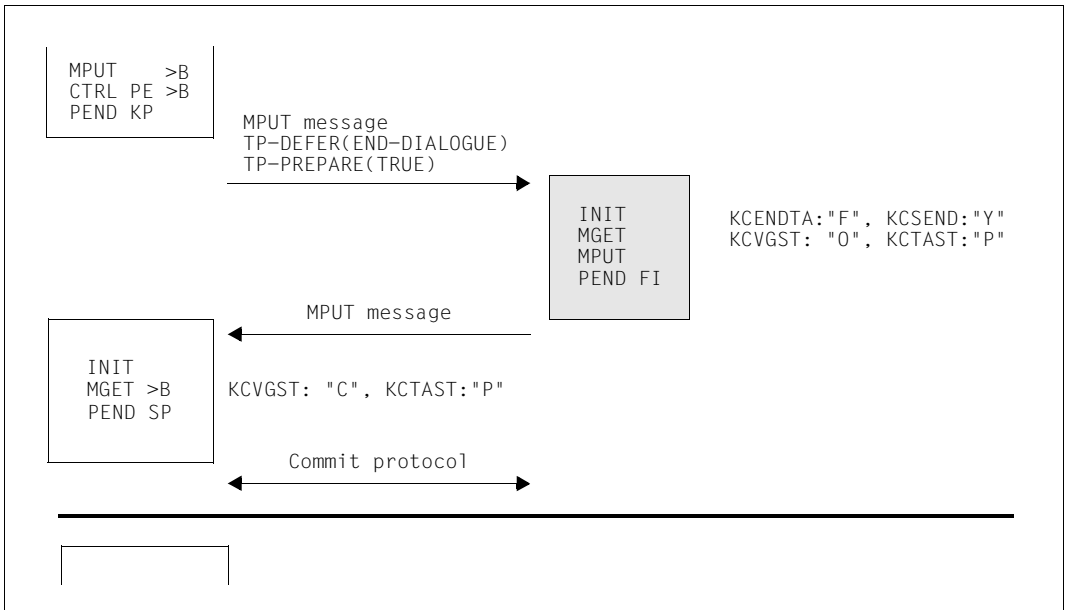
**End of dialog**

If the Commit functionality is used, the job receiver can only terminate the service if requested to do so by the job submitter.

Normally the job-receiving services are terminated first and the job-submitting service can terminate afterwards. It is also possible to terminate the job-submitting and job-receiving service simultaneously.

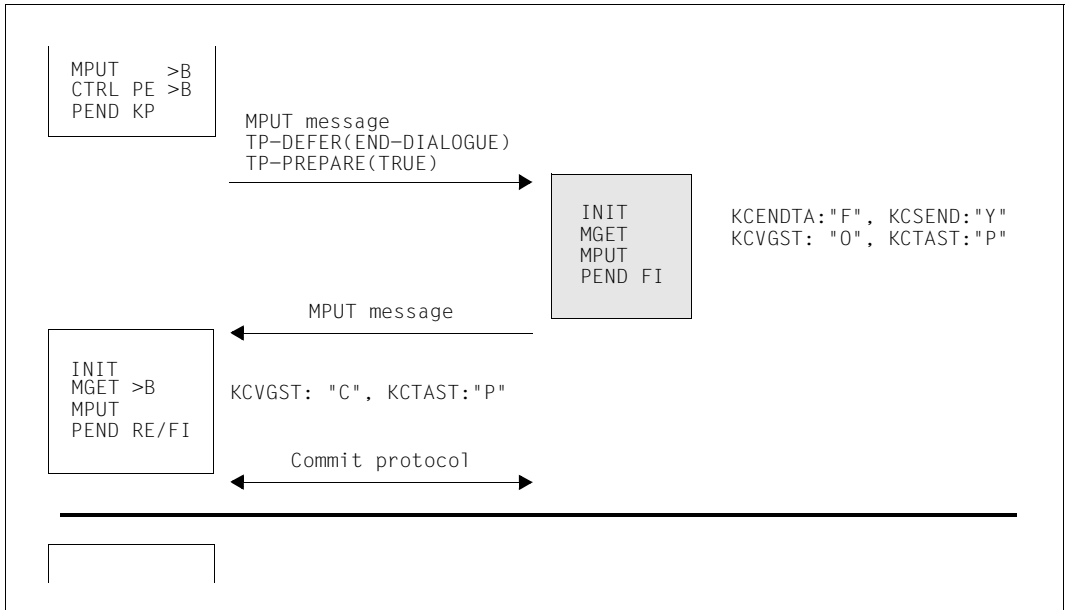
If the job-submitting service is to be continued, then the job receiver must use the CTRL PE call to request the job receiver to end the service.

*Example 7: Message and end dialog to job receiver and PEND KP*

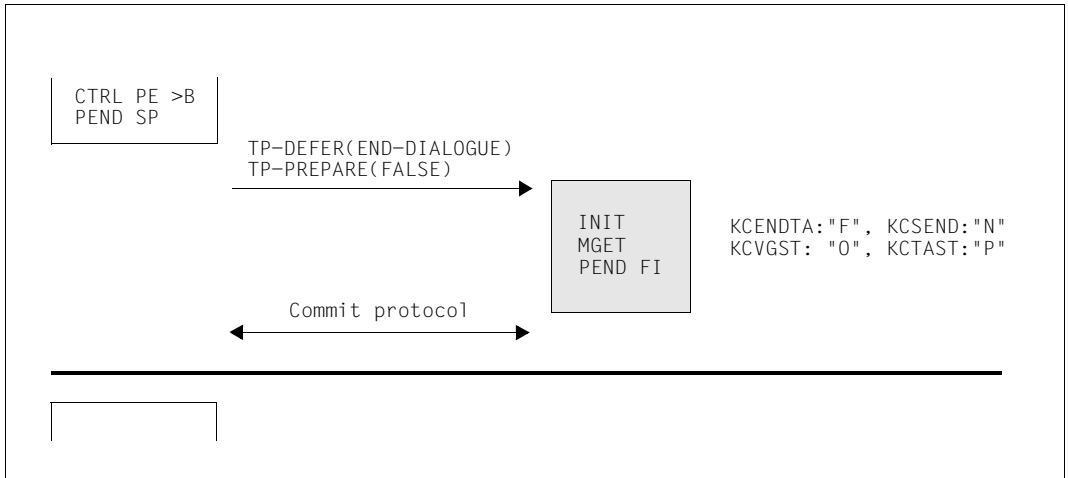


In this example, the job receiver can send a last message to the job submitter before the job receiver terminates the service.

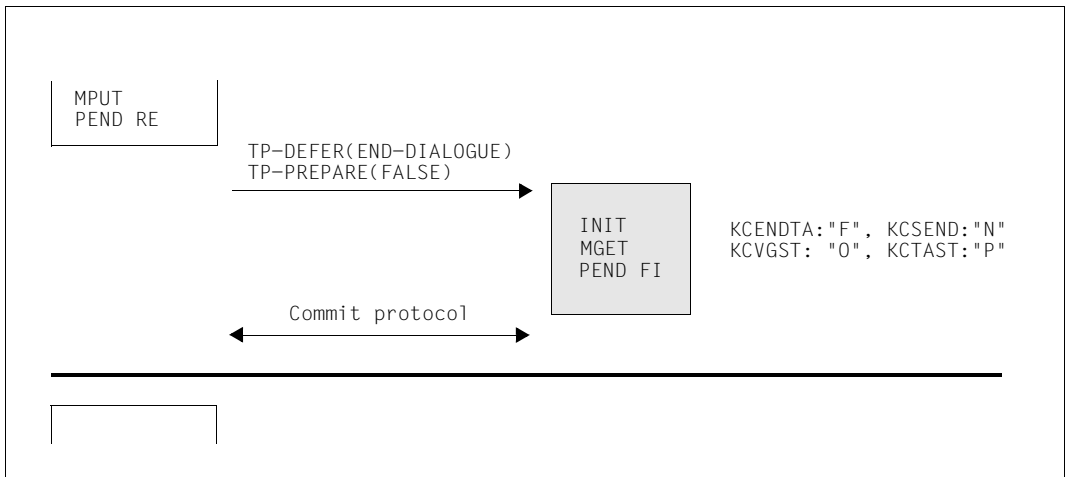
In the second job submitter program unit run, instead of PEND SP, you can issue an MPUT to the client and another PEND call to request end of service or end of transaction. The command sequence is then as follows:



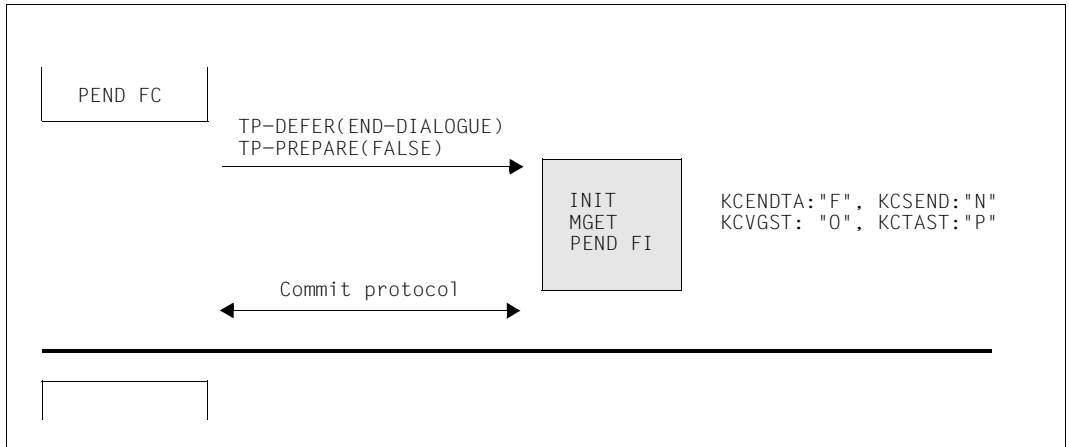
Example 8: No message to the job receiver and PENDING SP/RE



In the first job submitter program unit run, instead of PENDING SP, you can issue an MPUT to the client, and another PENDING call to request end of service or end of transaction. This results in the following command sequence:

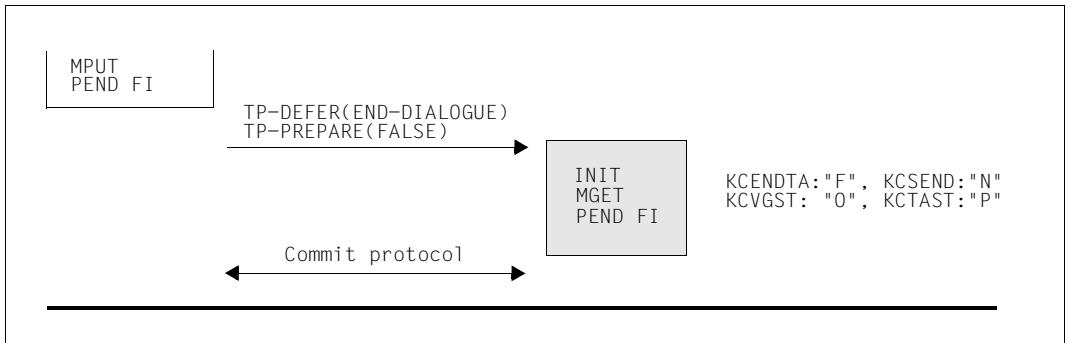


Example 9: No message to the job receiver and PENDING FC/FI



In the example above, the job-submitting service is not to be continued as in examples 6 and 7. Instead, it terminates at the same time as the job receiver. If PENDING FC (service chaining) is used, the dialog step is continued in a follow-up service.

In the first job submitter program unit run, instead of PENDING FC, you can issue an MPUT to the terminal and a PENDING FI. In this case no follow-up service is performed in the job submitter. The command sequence is then as follows:





### 5.4.9.2 Multiple job receivers

When communicating with more than one job receiver, the situation of the job submitter is essentially the same whatever the physical number of job receivers. It is therefore sufficient to consider a configuration involving one job submitter (A) and two job receivers (B and C).

From the job receiver's point of view, this case is identical to the situations illustrated in the previous section, since the only communication partner known to the job receivers is the job submitter.

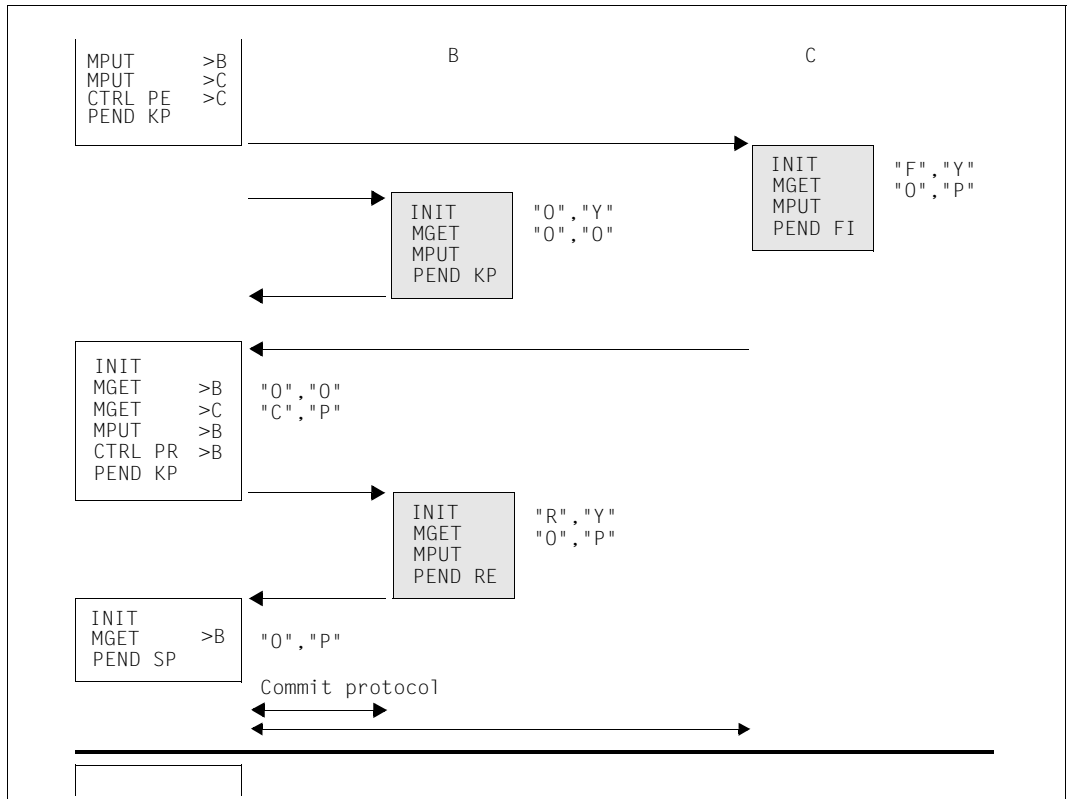
However, this scenario is also not very different for the job submitter. There is simply an increase in the number of possible combinations.

The job submitter can communicate with each individual job receiver as described in the previous section. Additionally, the job submitter can communicate either with one or with multiple job receivers in a single processing step. The follow-up program unit run in the job-submitting service is not started until responses have been received from all the job receivers to which messages were sent in the last processing step.

The job submitter can either use the CTRL call to request individual job receivers to request end of transaction or end of dialog or issue an appropriate PEND call to inform all job receivers of the situation simultaneously.

Since the situation has not changed greatly compared to communication with a single job receiver, a single example will suffice here. For reasons of space, the protocol flow is not illustrated.

Example 10: Multiple job receivers



In this example, job submitter A communicates with job receivers B and C. The dialog with C is to be terminated. However, C has to send a final message to A before terminating. To make this possible, A issues an MPUT and a CTRL PE to partner C. The dialog with partner B is not to be terminated yet. A therefore simply sends a message to B and keeps the transaction open by using PEND KP to terminate the program run.

The "F,Y" specifications inform C that it still has to send a message to A and that the transaction has to be terminated with PEND FI. For B, the transaction and dialog remain open. This is indicated by "O,Y".

In the second program unit run, A now uses CTRL PR to request B to end the transaction. However, A wants to receive the response from B in the current transaction and therefore uses PEND KP to terminate the program run. The stati "R,Y" signal the end of transaction request to B. B then sends a response to A and uses PEND RE to terminate the transaction.

Since both C and B have now requested end of transaction, A can finally terminate the distributed transaction. At the end of transaction the dialog with C is simultaneously terminated, whereas the dialog with B remains open.

#### **5.4.9.3 More complex dialog trees**

Finally we shall look at cases in which a service (B) communicates with a job submitter (A) and a job receiver (C) via the OSI TP protocol using Chained Transactions. Compared to the previous cases, only the intermediate node B is in a new situation since it possesses both a job submitter A and a job receiver C.

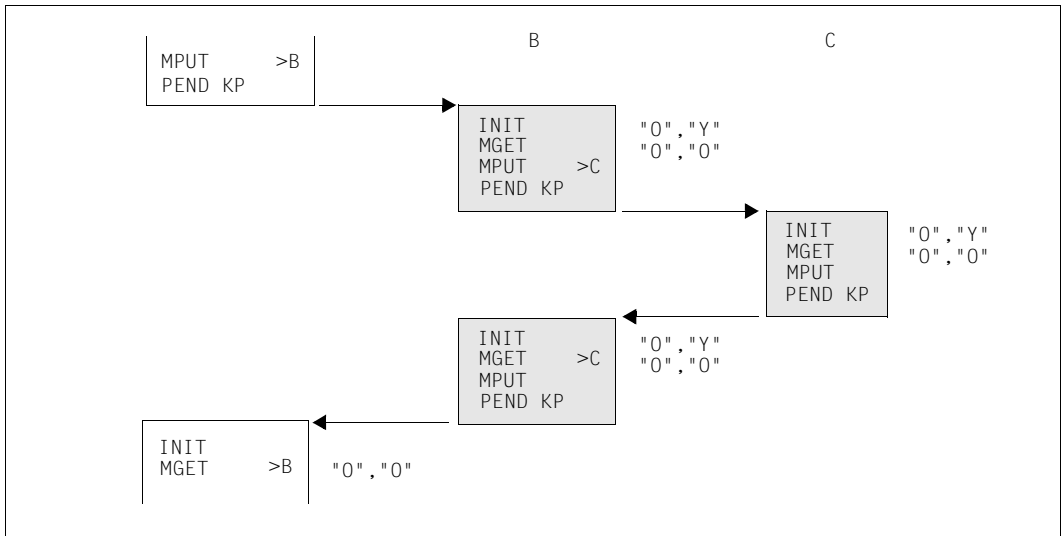
When the OSI TP protocol is used, an intermediate node is not free to decide when an end of transaction or end of dialog is to occur. This also applies to dialogs with its job receiver. The intermediate node B cannot request end of transaction or end of service from job receiver C until B itself has received an end-of-transaction request from its job submitter.

The following examples depict individual, characteristic situations. There are numerous other possibilities which can be constructed by combining the cases described in the sections above.

**Data transfer phase**

The data transfer phase is the period in which no CTRL calls are issued by any of the partners, and the program runs are terminated exclusively by PEND KP.

*Example 11: Data transfer phase in multi-step transfer trees.*



B need not always communicate with A and C in alternation, as is the case in this example. B can also conduct multiple dialog steps consecutively with C or communicate exclusively with A before reintegrating C into the communication.

However, it is important to note that B may not send messages to A and C simultaneously. An intermediate node may pass the send authorization either to the job submitter or to one or more job receivers, but not to the job submitter and a job receiver at the same time.

A service may transfer the end-of-transaction send authorization for a maximum of one dialog.

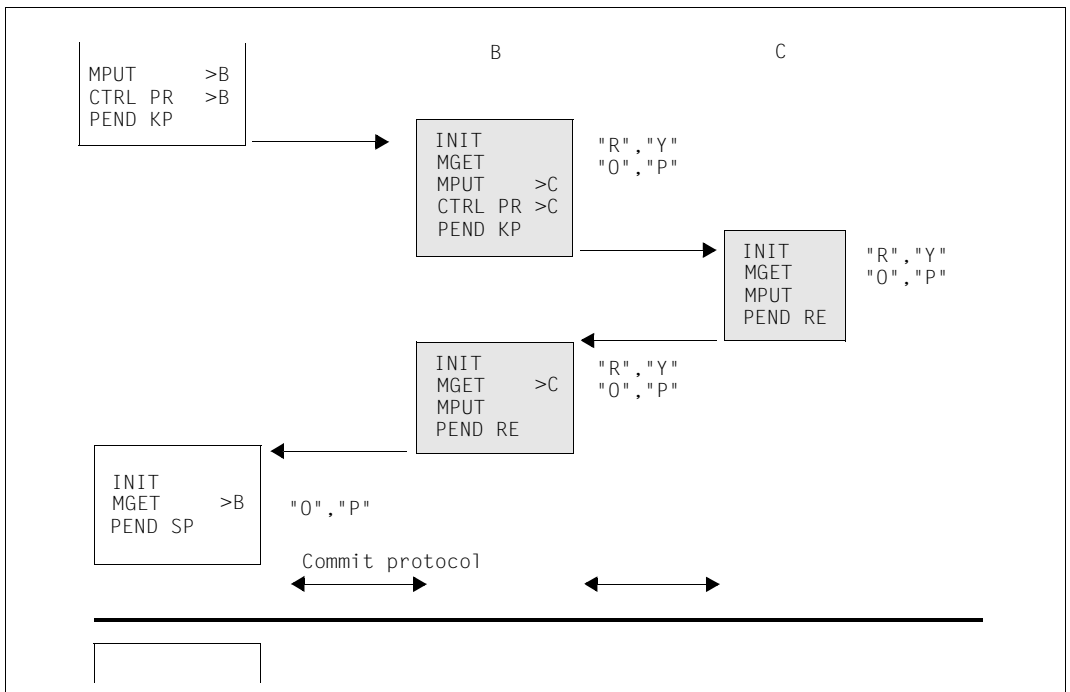
**End of transaction**

After B has received the end-of-transaction request from A, it has 3 options:

- B can send a message to C and simultaneously request C to terminate the transaction (see also examples 12, 14, 15)
- B can continue the data transfer phase with C and request C to terminate the transaction in a later processing step. (see also example 13)
- B can refrain from further communicating with C in the current transaction and itself request end of transaction. (see also example 16)

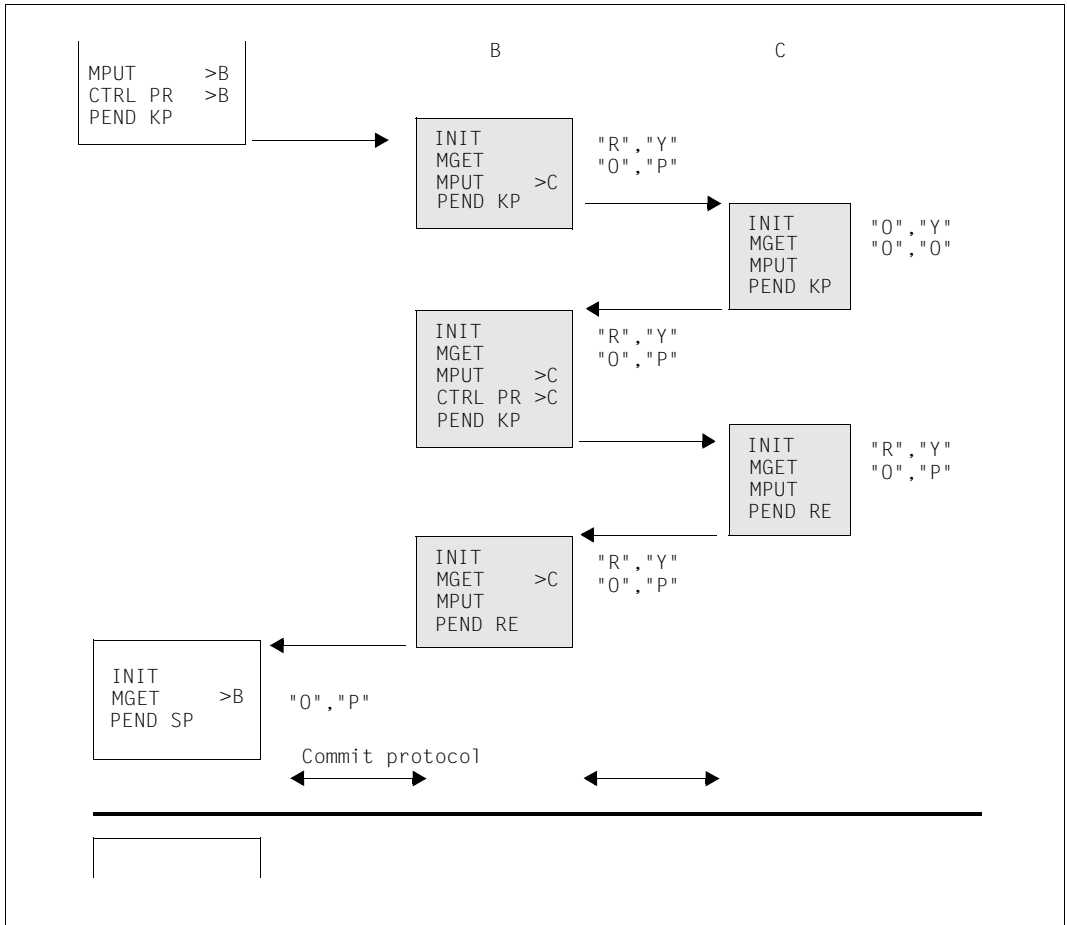
A, B or C may own the end-of-transaction send authorization.

*Example 12: End-of-transaction send authorization is owned by A*



In the example above, B cannot pass the end-of-transaction send authorization to C since A has not yet passed the end-of-transaction send authorization to B.

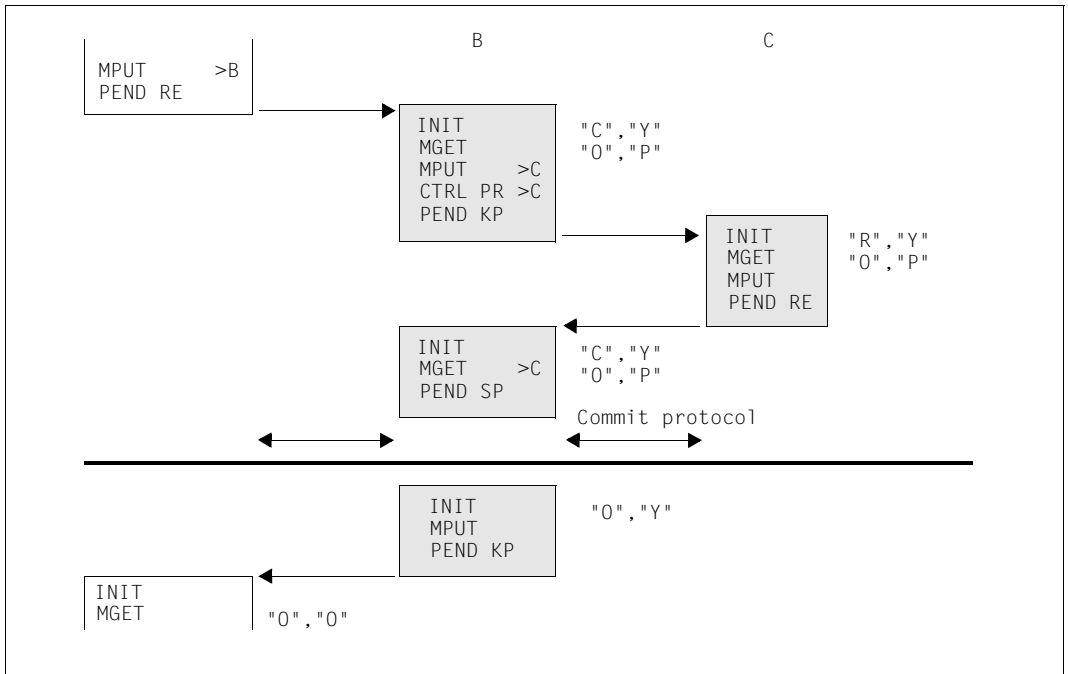
Example 13: B continues the dialog with C - end-of-transaction send authorization is owned by A



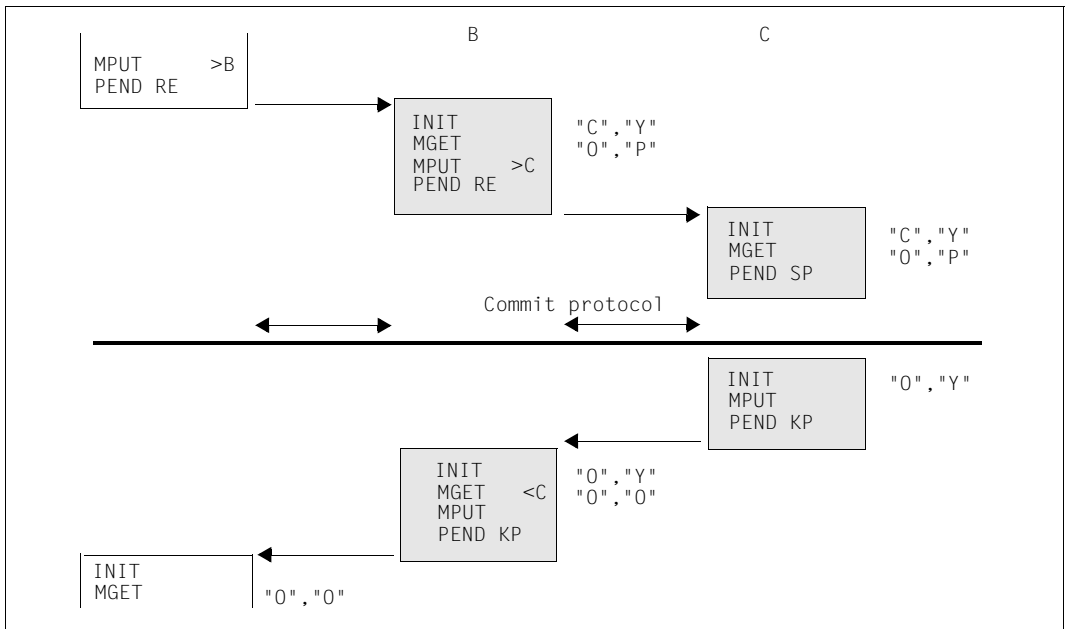
In this example, B does not initially request C to terminate the transaction and, instead, continues the data transfer phase with C. In this example only one more dialog message is exchanged. However, it would be possible to continue the data transfer phase beyond this. In this case, B and C can only use PEND KP to terminate the program runs. At some time B must request C to end the transaction.

In this example, again, B cannot pass the end-of-transaction send authorization to C since A has not yet transferred the end-of-transaction send authorization to B.

Example 14: End-of-transaction send authorization is owned by B

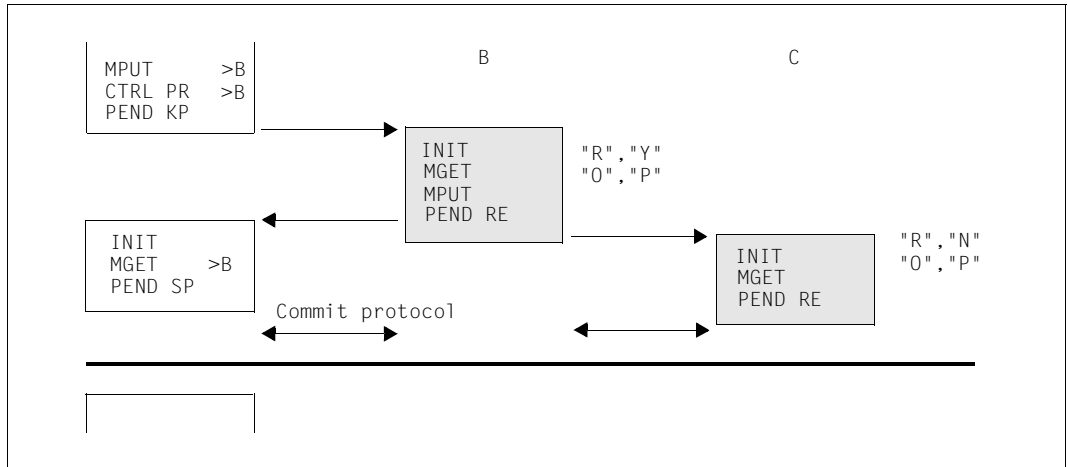


Example 15: End-of-transaction send authorization is owned by C





Example 16: No message to job receiver before end of transaction



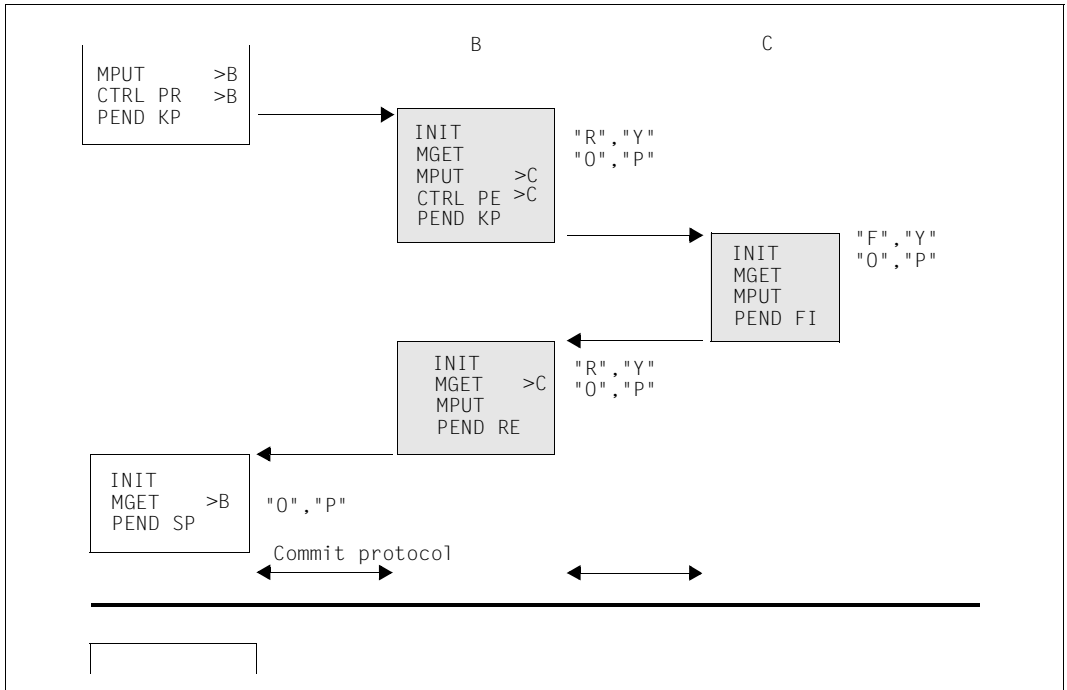
In this example, B refrains from sending another message to C in the current transaction, and requests end of transaction immediately.

After PEND RE from node B, an MPUT message is sent to A and a PREPARE protocol element is sent to C. This requests C to terminate the transaction. C does not then receive any further user messages. The MGET call at C simply reads the status of the dialog with B. This call can be omitted.

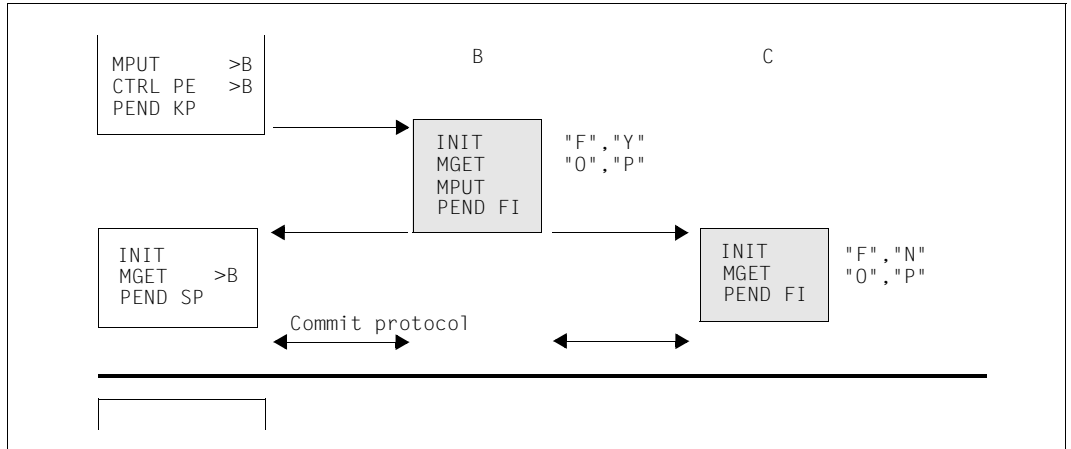
**End of dialog**

An intermediate node can bring about the job-submitter-based end of dialog in the same way as an end of transaction. These possibilities are depicted in the two examples below.

*Example 17: B first terminates the dialog with the job receiver.*



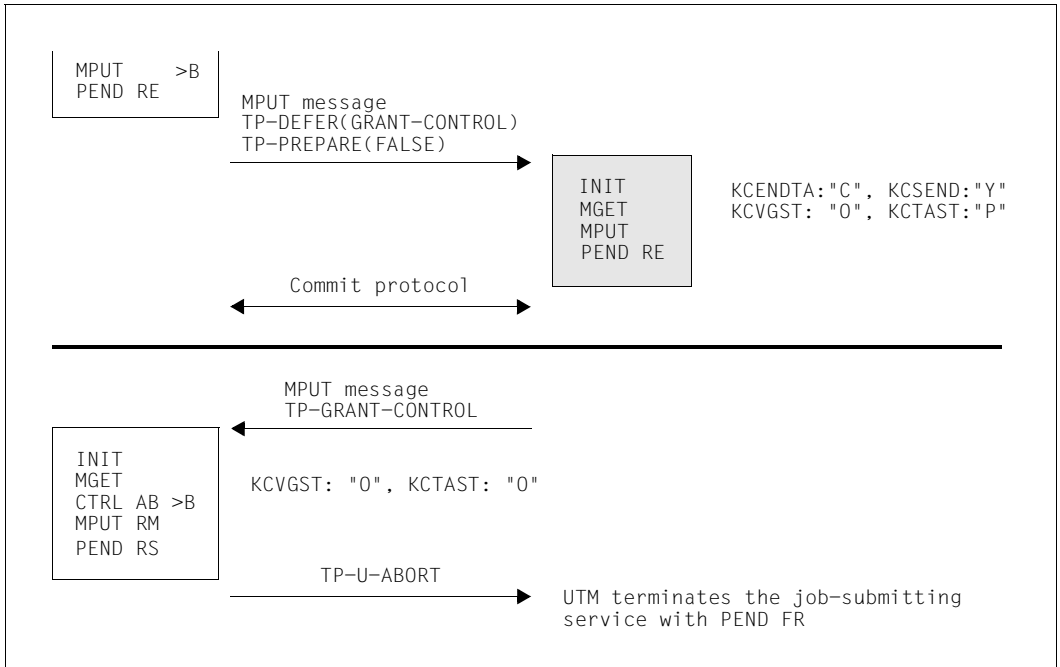
Example 18: Simultaneous end of dialog with job receiver and job submitter.



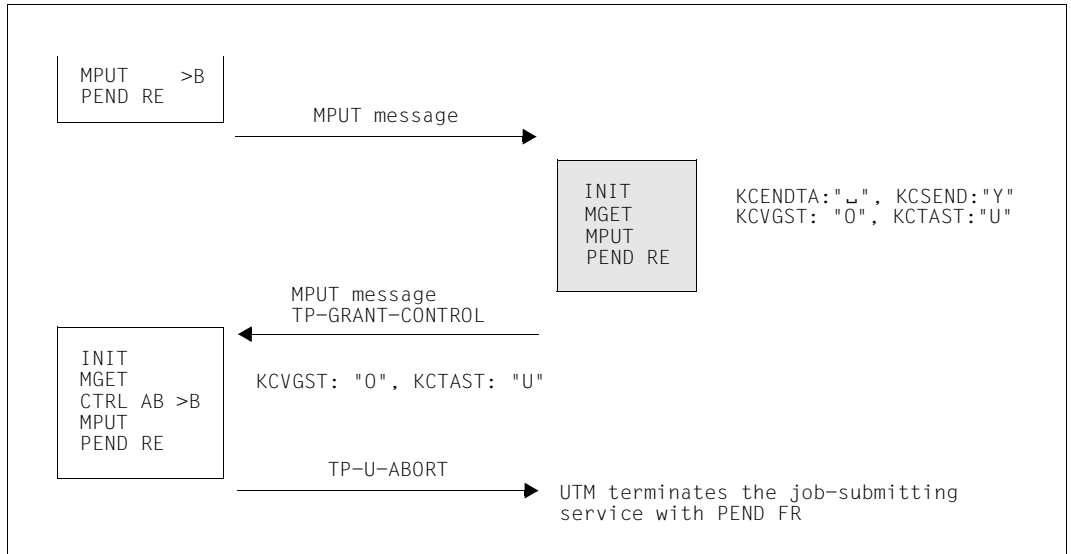
5.4.9.4 Using CTRL AB to terminate a job receiver

CTRL AB terminates a job-receiving service abnormally. The job submitter must reset the distributed transaction after a CTRL AB for a job receiver for which the functional unit Commit is selected. No reset is necessary for a job receiver dialog without Commit.

*Example 19: Terminating a dialog for which the Commit FU is selected.*



Example 20: Terminating a dialog for which Commit FU is not selected



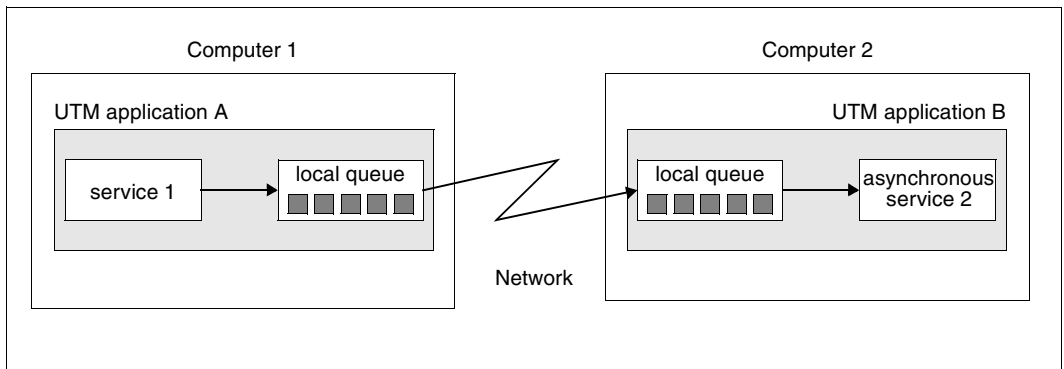
## 5.5 UTM-controlled queues in distributed processing

A job-submitting service can use the FPUT or DPUT call to send an asynchronous job to a remote asynchronous service (Remote Queuing). The job-submitting service may either be a dialog or an asynchronous service.

openUTM uses two local queues for asynchronous jobs to a remote applications: one queue is located in the sending application, the other in the receiver. This deferred delivery principle makes sure that distributed message queuing under openUTM is entirely independent of whether or not a connection is currently established. If no connection can be established, the job remains in the local send queue until the connection is established. The following applies once the connection has been established.

- With LU6.1, the jobs are transmitted to the partner immediately.
- With OSI TP, it may be some time before the jobs are transmitted. This time is limited by the value generated in MAX CONRTIME. Note that the time is set to 10 minutes if CONRTIME=0.

If a serious error occurs while transmitting a job, i.e. while the connection is open, then the job is deleted from the local send queue but is not entered in the corresponding message queue of the job-receiving application. A serious error that results in the loss of the job may occur, for example, if the job is sent to a locked TAC in the job-receiving application. The exact cause can be identified from the K086 (LU6.1) or K119 (OSI TP) message which is output in the job-receiving application:



Remote queuing with openUTM

### 5.5.1 Job submitter side

You use an **APRO AM** call to address the job-receiving service. Enter the service ID in the KCPI field.

In the case of distributed processing via OSI TP you can use the APRO call to select whether or not to transfer an asynchronous job with global transaction management. If global transaction management is used, openUTM ensures that the job is transferred precisely once as long as it is not lost during transmission due to a serious error (see [page 190](#)).

In the event of a connection failure, asynchronous jobs without global transaction management are may sometimes be transferred more than once.

After an APRO AM call, the job-submitting service can:

- enter a service identification as the destination in KCRN and use FPUT to send an asynchronous job or DPUT to send a time-driven asynchronous job to the corresponding remote service.
- use MCOM BC to define the start of a job complex and use DPUT to send an asynchronous job (basic job) to the job receiver application within the complex and to create the associated positive or negative confirmation jobs. The confirmation jobs are processed by the local application.
- use MCOM BC to define the complex ID and enter the service Id in KCRN. For DPUT, you must then enter the complex ID in KCRN.

You **must** issue an FPUT or DPUT call with this service ID within the program unit which addresses the remote service with APRO AM, otherwise openUTM aborts the service with KCRCCC=86Z and releases the service ID when PEND is called.

The service ID is released in the job-submitting service in the following cases:

- after a successful FPUT NE or DPUT NE call
- on the next PEND call (also PEND KP and PEND PA/PR)
- after a RSET call
- after the return code 40Z following an FPUT or DPUT call
- in job complexes with this service ID: When calling MCOM EC or after a return code 40Z following MCOM BC or after calling DPUT

Once released, this service ID can be used for another job submitter/job receiver relationship in the job-submitting service.

The job entry in the job submitter is deleted from the message queue as soon as it has been successfully transferred and inserted into the corresponding message queue of the job-receiving application. Depending on the processing result, the positive or negative confirmation job is then started when message complexes are used.

## 5.5.2 Job receiver side

Here an asynchronous job for a partner application is handled as if it had been created by a service in your own application. Asynchronous jobs from services in the local application and asynchronous jobs issued by remote services are located in a shared message queue assigned to the asynchronous TAC. An asynchronous service is started for each job in turn, as resources become available. The asynchronous service uses the entry in the KCTERMN field of the KB header to identify whether or not the job submitter is a remote service.

Asynchronous services for remote queuing are structured in exactly the same way as for local queuing (see [page 53ff](#)). However, in distributed processing via OSI TP another possibility exists: asynchronous jobs to dialog services.

### **Asynchronous jobs to remote dialog services (only via OSI TP)**

When using the OSI TP protocol for an asynchronous job for which APRO was used to specify global transaction management, the job receiver may be a **dialog** service.

After receipt of an asynchronous job for a dialog service, the service is immediately started in the job receiver application rather than being inserted into a message queue like a job to an asynchronous service. In the job-submitting application, the job is not deleted from the message queue until the dialog service is terminated. Depending on the processing result, the positive or negative confirmation job is then started when message complexes are used.

A dialog service which is started by an asynchronous job must use PEND FI to terminate the transaction and may not contain an MPUT to the job submitter (KCENDTA=F and KCSEND=N).



## 5.6 Service-controlled queues in distributed processing

Messages can also be transferred to TAC queues of remote applications with fail safety by means of LU6.1 and OSI TP. Generation and programming are similar to when you send jobs to dialog or asynchronous TACs in remote applications (see [page 122](#)).

### *Generation*

In an LTAC statement, the name of a TAC queue in the remote application is assigned to a local LTAC name as the RTAC name.

```
LTAC REMOTEQ, RTAC=name-of-TAC-queue-in-remote-application
```

### *Programming*

The message is addressed by means of an APRO AM call and sent by means of a subsequent FPUT call.

```
INIT  
...  
APRO AM, KCPI=>VGID, KCRN=REMOTEQ  
FPUT NE, KCRN=>VGID  
...  
PEND FI
```



---

## 6 Program structure in communication with transport system applications

This chapter describes the points you have to take into account when programming communication with transport system applications (= TS applications).

### 6.1 Communication with TS applications of the type APPLI

From a TS application of the type APPLI it is possible in the UTM application to:

- start a service
- create a message for a TAC queue or temporary queue

The TAC or name of the queue must be at the beginning of the message. If the TAC or the name of the queue is shorter than eight characters, it must be separated from the rest of the message by at least one blank. If a message to a temporary queue is created, there must not be a TAC, LTERM, LPAP or OSI-LPAP in the application under the name of the temporary queue.

If the specified TAC or the name of the queue is invalid, the BADTACS event service is started (provided it has been generated). If not, message K009 is sent to the TS application, unless a separate message module is used in which PARTNER has not been generated as the message destination for message K009.

Messages to a queue can be read using the KDCS call DGET. Messages to a service are read using the KDCS call MGET or FGET. If, during a read call, the KCLA length is shorter than the length of the message segment, only the requested part is read; the rest of the message is lost. The return code 01Z indicates that the message has not been read in its entirety.

At generation you must define the TS application as follows:

```
PTERM/TPOOL . . . ,PTYPE=APPLI
```

You will find information on generating TS applications in the openUTM manual "Generating Applications".

## 6.2 Communication via socket connections

openUTM works on a message-oriented basis and does not start a program unit until a complete message has been received for the program unit. The socket interface is a byte-stream interface.

openUTM therefore requires a communication protocol above and in addition to TCP/IP in order to detect message limits. For this purpose, openUTM provides its own protocol (openUTM Socket Protocol, **USP**), which enables the byte streams received via the socket interface to be converted to messages (see [page 199](#) for the structure of the USP header). TCP/IP is required as the transport protocol.

### Generating socket partners

Socket partners are generated as follows:

```
PTERM/TPOOL . . . PTYPE=SOCKET ,BCAMAPPL=socket-applname
BCAMAPPL socket-applname ,T-PROT=SOCKET
```

X/W  
X/W

In Unix systems and Windows systems, T-PROT=SOCKET must also be specified for PTERM/TPOOL.

In addition, for these socket partners you can specify by means of the USP-HDR= operand in the PTERM or TPOOL statement whether, in the case of messages to the socket partner, openUTM automatically creates a USP header and precedes the message with it. You will find information on generating socket applications in the openUTM manual "Generating Applications".

### 6.2.1 Input messages for openUTM

In the case of input messages of socket applications, a corresponding protocol header must be set up in the partner application, and this must precede each message segment or fragment. The protocol header is truncated and not transferred to the program unit.

The following applies to the data that comes after the protocol header:

- If a service is to be started with a message sent to openUTM, the TAC must be at the beginning of the message.
- If the message is to be inserted in the message queue of a TAC queue or of a temporary queue, the name of the queue must be at the beginning of the message. If the message goes to a temporary queue, there must be no TAC, LTERM, LPAP or OSI-LPAP in the application under the name of the temporary queue.

- If the TAC is shorter than eight characters, it must be separated from the rest of the message by at least one blank.
- If the specified TAC is invalid, the BADTACS event service is started (provided it has been generated). If not, message K009 is sent to the socket application, unless a separate message module is used in which PARTNER has not been generated as the message destination for message K009.

The messages or message segments to a service-controlled queue can be read by means of the KDCS call DGET. The messages or message segments to a service are read by means of the KDCS call MGET or FGET. The following then applies.

- If the KCLA length is shorter than the length of the message segment at the MGET call, only the requested part is read; the rest of the message is not lost, however. The return code 02Z indicates that the message segment has not been read in its entirety. The rest of the message segment can be read with the next read call, another message segment with the next read call but one, and so on.
- If the KCLA length is shorter than the length of the message segment at the DGET/FGET call, only the requested part is read; the rest of the message is lost. The return code 01Z indicates that the message segment has not been read in its entirety. A further message segment is read with the next read call.

If all the message segments are read, this is indicated by the return code 10Z.

If the setting MAP=USER applies to the connection to the socket partner, the socket application must provide transaction codes in the "correct" code for the UTM application (i.e. in EBCDIC for a UTM(BS2000) application and in ASCII for a UTM application under Unix systems or Windows systems).

## 6.2.2 Output messages of openUTM

When messages are sent to socket partners, each message segment or fragment must be sent by means of a separate MPUT NT/NE call. At each MPUT call, a separate message segment is created, even if zero is specified as the length. A message with a length of zero is only sent, however, if a USP header is created automatically for the message to be sent and is placed in front of it (see the table below). The exception is that if the program unit contains only one MPUT call, no message is sent, regardless of the value of the USP-HDR parameter.

In the case of output messages, openUTM does not normally create a USP header. If the socket partner expects a USP header when it receives a message, you can use the USP-HDR operand in the PTERM or TPOOL statement to specify that openUTM should automatically create a USP header for messages to the socket partner and precede them with this header. Alternatively, you can create the USP header yourself in the program unit and then send it.

At generation you can specify whether and for which of the following message transmissions openUTM is to automatically create a header:

- for all messages, i.e. K messages + MPUT/FPUT messages (USP-HDR=ALL)
- for K messages only (USP-HDR=MSG)
- no header (USP-HDR = NO)

### Exchanging long messages

Input messages and dialog output messages of any length can be exchanged with socket partners. If an entire message is longer than 32767 bytes (output) or 32000 bytes (input), the message must be **fragmented** (i.e. it must consist of several message segments). The maximum length of each message segment is 32767 bytes (including the USP header) at output and 32000 bytes at input.

Message segments can be identified by the fact that the relevant flag field and the corresponding message type in the USP header are set (see below). In the case of outputs, the program must set these values.

Fragmented input messages must be read using a corresponding number of MGET/FGET/DGET NT calls (see [page 196](#)). Fragmented dialog output messages must be sent using a corresponding number of MPUT NT calls.

Asynchronous output messages (FPUT/DPUT) cannot be fragmented. Their total length is limited by the value generated in MAX TRMSGLTH=.

### 6.2.3 Structure of the socket protocol header

The header contains the identifier, two version fields, a flag field, a type field and a length field. openUTM expects the protocol described on the input side. This has the following structure:

Identifier ASCII, 4 bytes	Major version 1 byte	Minor version 1 byte	Flags 1 byte	MsgType 1 byte	MsgSize 4 bytes	Data Max. 31988 bytes
UTMS(55544D53)	01	01		Message type	0000xxxx	Data

#### Explanation

##### Identifier

The identification field must always contain the ASCII string "UTMS" (=X'55544D53') for openUTM.

##### Major version

The major version field must always contain the value "1".

##### Minor version

The minor version field must contain the value "1" if functions of openUTM version > 5.1 are to be used. In the case of automatically created USP headers, openUTM always enters the highest version in the major/minor fields (i.e. 1.1 in the case of openUTM V5.3). You will find information on clients that send version 1.0 on [page 201](#).

**Flags** The flag field is an information field. Only bit 0x02 is evaluated: if it is set, the message is followed by another fragment. If it is not set, it is not followed by a fragment. All the other bits are reserved for future versions.

##### MsgType

The type field can contain the values "0", "1" or "7":

- Receive messages from the client:  
In the case of unfragmented messages, the client sets the value "0" for `MsgType`. In the case of fragmented messages, the client sets the value "0" for `MsgType` when the first fragment is received (bit 0x02 is set in the flag field). For the fragments that follow, the value "7" is set for `MsgType` (bit 0x02 remains set). Bit 0x02 is not set for the last of these fragments.
- Send messages to the client:  
In the case a USP header is created (e.g. with USP-HDR=ALL), openUTM sets the value "1" for `MsgType` when the first fragment is sent and the value "7" for fragments that follow.

**MsgSize**

The length field `MsgSize` contains the length of the message or message segment, including the header. This length must not exceed 32000 bytes on the input side and 32767 bytes on the output side. The message length is transferred in network byte order (big endian). The C functions `htonl` (host to network long) and `ntohl` (network to host long) can be used to carry out conversions between the local view and the network view.

**Examples**

1. The client sends messages to openUTM, see the sample source file `SOCBSP.C` shipped with the product:
  - The client sends one message (without fragmentation)
  - The client sends two message segments
  - The client sends three message segments

Number of message segments	Identifier	Major version	Minor version	Flag	MsgType	Size
1 whole message	55544D53	01	01	00	00	0000nnnn
2 segments						
– segment 1	55544D53	01	01	02	00	0000nnnn
– segment 2	55544D53	01	01	00	07	0000mmmm
3 segments						
– segment 1	55544D53	01	01	02	00	0000kkkk
– segment 2	55544D53	01	01	02	07	0000nnnn
– segment 3	55544D53	01	01	00	07	0000mmmm



2. The server sends messages to the client (USP header is created):
- one whole message (without fragmentation)
  - three message segments

Number of message segments	Identifier	Major version	Minor version	Flag	MsgType	Size
1 whole message	55544D53	01	01	00	01	0000nnnn
3 segments						
– segment 1	55544D53	01	01	02	01	0000kkkk
– segment 2	55544D53	01	01	02	07	0000nnnn
– segment 3	55544D53	01	01	00	07	0000mmmm

*kkkk*, *nnnn*, *mmmm* are the lengths of the different message segments.

### Notes on compatibility

If a socket partner expects the protocol interface used in openUTM V5.0, this can continue to be used: if bit 0x80 (most significant bit) of the USP flag field of the last input message is set, openUTM sets up the USP header automatically for the associated dialog output message. In this case, the messages sent by means of MPUT NT are sent with only *one* USP header until the maximum message length is reached so that the socket partners are not sent socket message segments when headers are created automatically.

Until openUTM version 5.0 it was only possible to send one socket message with a length of up to 32000 bytes using several MPUT NT calls.

Example (sending a maximum of 32000 bytes):

```
MPUT NT (length of 20000 bytes including USP header)
MPUT NT (length of 5000 bytes)
MPUT NT (length of 5000 bytes)
PEND FI
```

If the size of all the MPUT NTs exceeds the maximum MPUT size, a PEND ER dump is written.



---

## 7 KDCS calls

This chapter gives you all the information you need to use the KDCS program interface in your program. You call the openUTM linkage program with the KDCS call. On the basis of the entries in the KDCS parameter area, openUTM recognizes and performs the desired function.

## Complete overview of KDCS calls

UTM calls implement the KDCS interface as standardized under DIN 66 265 ("Interfaces of a Kernel for Transaction-oriented Application Systems"). The UTM program interface is an upward-compatible extension to this DIN standard. The table below lists these extensions.

CALL	Included in DIN 66 265	Extensions to DIN 66 265
APRO	no	Distributed processing
CTRL	no	Distributed processing (only for OSI TP)
DADM	no	Administration of message queues (asynchronous jobs)
DGET	no	Read from service-controlled message queues
DPUT	yes	Confirmation jobs and user information
FGET	yes	Distributed processing
FPUT	yes	Distributed processing
GTDA	yes	none
INFO	no	Information services
INIT	yes	Distributed processing
LPUT	yes	none
MCOM	no	Definition of job complexes
MGET	yes	Distributed processing
MPUT	yes	Distributed processing
QCRE	no	Create temporary queue
QREL	no	Delete temporary queue
PADM	no	Administration of printers and printer output
PEND	yes	Distributed processing, PEND KP/PS/FC/SP/RS/FR not contained in DIN 66 265
PGWT	no	Program management
PTDA	yes	none
RSET	no	Reset operation
SGET	yes	none
SIGN	no	Signing on and off, changing password, checking authorization data
SPUT	yes	none
SREL	yes	none
UNLK	no	Unlocking storage areas

The table below shows the KDCS calls and their functions.

<b>CALL</b>	<b>Function</b>	<b>Function group</b>
APRO	Address job-receiving service	Message communication (with distributed processing)
CTRL	Control OSI TP dialogs	Message communication (with distributed processing)
DADM	Administer asynchronous jobs	Management of message queues and printers
DGET	Read messages from a service-controlled message queue	Message communication-message queuing
DPUT	Generate time-driven asynchronous job and confirmation jobs	Message communication-message queuing
FGET	Receive asynchronous message	Message communication-message queuing
FPUT	Generate asynchronous job	Message communication-message queuing
GTDA	Read from TLS	Memory management
INFO	Request information	Information services
INIT	Signing on a program to openUTM	Program management
LPUT	Write to log file	Logging facility
MCOM	Define job complex	Message communication
MGET	Receive dialog message	Message communication dialog
MPUT	Send dialog message	Message communication dialog
QCRE	Create temporary message queues	Management of temporary message queues
QREL	Delete temporary message queues	Management of temporary message queues
PADM	Control printers and printer outputs	Management of message queues and printers
PEND	Terminate program	Program management
PGWT	Set wait point in a program unit run	Program management
PTDA	Write to TLS	Memory management
RSET	Reset requested changes and operations	Program management
SGET	Read from secondary storage area	Memory management
SIGN	Sign on and off, change password, check authorization data	Sign-on management

<b>CALL</b>	<b>Function</b>	<b>Function group</b>
SPUT	Write to secondary storage area	Memory management
SREL	Release secondary storage area	Memory management
UNLK	Unlock TLS, ULS or GSSB	Memory management

The next table lists the function groups and the calls associated with them.

<b>Function group</b>	<b>Associated UTM calls</b>
Sign-on management	SIGN
Program management	INIT, RSET, PEND, PGWT
Message communication-dialog	MGET, MPUT
Message communication message queuing	MCOM, DGET, DPUT, FGET, FPUT
Management of message queues and printers	DADM, PADM
Management of temporary message queues	QCRE, QREL
Memory management	GTDA, PTDA, SGET, SPUT, SREL, UNLK
Information services	INFO
Logging facility	LPUT
Distributed processing	APRO, CTRL, DPUT, FPUT, INIT, MCOM, MGET, MPUT, PEND, RSET, PGWT

After each call (except PEND) openUTM returns information in the KDCS communication area.

The fields of the KDCS parameter area set prior to a call or returned after its execution have particular names. These are meant to help you deal with the KDCS interface and its description. Their use also enhances the maintenance and transferability of programs. The appendix gives an overview of all operand fields and their use in the various calls.

## Comments on the description of the KDCS calls

This section describes all the KDCS calls in alphabetical order. This makes it easier for you to look them up.

Every call description consists of 4 parts:

- First, the functions of the call are described.
- Then there is a schematic diagram of the call with all necessary entries. The fields to which you have to assign a value before the call are grayed:

gray you must assign a value to this field before the call

The corresponding C/C++ macros are also listed for each call. For a detailed description of these macros refer to [section "C/C++ macro interface" on page 497](#).

- This diagram is followed by the description of the statements in the KDCS parameter area and in the 2nd parameter, as well as the return information from openUTM.
- Finally the particularities of the call are explained.

If "—" is entered in a table, then the entry is irrelevant for the function concerned.



The field names of the KDCS interface for COBOL and C/C++ are mostly identical and differ only in the rules for lower case/upper case specification. In cases where further differences exist, the field name for C/C++ is printed after the COBOL field name (separated by a slash), e.g. "KCTAG/kcday".



The return code 79Z is a general error code which indicates that the value in the field KCOP=operation code has an invalid value. It cannot therefore be assigned to any operation code.

## APRO Address job-receiving service

The APRO call (address process) enables you to address a job-receiving service or a TAC queue in the job-submitting service. The APRO call is only applicable with distributed processing with UTM-D. Data is sent to the job-receiving service using either MPUT or FPUT/DPUT. In these calls the receiver is specified by means of the service identifier defined in the APRO call.

### Setting the 1st parameter (KDCS parameter area)

The following table shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area						
	KCOP	KCOM	KCLM	KCRN	KCPA	KCPI	KCOF
Address a dialog service	"APRO"	"DM"	0/19/58	LTAC name	(OSI-)LPAP name / Master-LPAP name / blanks	service ID	Permitted OSI function
Address an asynchronous service or a TAC queue	"APRO"	"AM"	0/19/58	LTAC name	(OSI-)LPAP name / Master-LPAP name / blanks	service ID	Permitted OSI function

The entry in the KCPA field depends on the type of addressing involved:

- for single-step addressing the field must contain blanks
- for double-step addressing the field must contain the name of the partner application ((OSI-)LPAP name or Master-LPAP name).

Further information on single and double-step addressing of job-receiving service with distributed processing can be found in the openUTM manual "Concepts und Functions".



**Setting the 2nd parameter (selecting special OSI TP function combinations)**

The second parameter area is only used for communication via the OSI TP protocol. It allows you to specify function combinations other than those available via the standard selection using the KDCS KOCF parameters. It also allows you to select whether SIGNON data should be transferred to the job-receiving service. A language-specific data structure is available for the second parameter area: for COBOL in the KCAPROC COPY element, for C/C++ in the *kcapro.h* include file.

If the second parameter area is used, you must specify the values 19 or 58 in the field KCLM for the length of the data structure and the field KCDF must contain the value "O".

**Setting the parameters in the KDCS parameter area**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"APRO"
2.	KCOM	"DM"/"AM"
3.	KCLM	0/19/58
4.	KCRN	LTAC name
5.	KCPA	(OSI-)LPAP name/Master-LPAP name / blanks
6.	KCPI	service ID
7.	KCOF	OSI functions

**Setting the parameters in the second parameter area (only for KCOF=0)**

	Field name in the 2nd parameter area	Contents
8.	KCVERS	Version number of data structure
9.	KCFUPOL	Polarized FU (Y)
10.	KCFUHSH	Handshake FU (Y/N)
11.	KCFUCOM	Commit FU (Y/N)
12.	KCFUCHN	Chained FU (Y/blanks)
	KCFUFILL	empty - for future extensions
13.	KCSECTYP	Security type (N/S/P)
14.	KCUIDTYP	Data type of user ID (P/T/O)
15.	KCUIDLTH	Length of user ID
16.	KCUSERID	User ID
	KCSECFIL	empty - for future extensions
17.	KCPWDTYP	Data type of the password (P/T/O)
18.	KCPWDLTH	Length of the password
19.	KCPSWORD	Password

**KDCS call**

	1st parameter	2nd parameter
20.	KDCS parameter area	Second parameter area

21. **C/C++ macro calls**

Macro name	Parameters
KDCS_APRODM / KDCS_APROAM	(kcrn,kcpa,kcpi)
KDCS_APRODM_OSI / KDCS_APROAM_OSI	(kcrn,kcpa,kcpi,kcof)
KDCS_APRODM_OSI_O / KDCS_APROAM_OSI_O	(nb,kclm,kcrn,kcpa,kcpi)

**openUTM return information**

	Field name in the KB return area	Contents
22.	KCRCCC	Return code
23.	KCRCDC	Internal return code

You can enter the following information for the APRO call in the KDCS parameter area:

1. In the **KCOP** field, enter the APRO operation code.
2. In the **KCOM** field, enter the operation modifier:
  - DM (Dialog message) for addressing a dialog service
  - AM (asynchronous message) for addressing an asynchronous service or a TAC queue.
3. In the **KCLM** field, enter the length of the 2nd parameter area:
  - length zero, if no second parameter area is used
  - length 19, if a second parameter area is used and the "N" or "S" security type is selected
  - length 58, if a second parameter area is used and "P" security type is selected.
4. In the **KCRN** field, enter the logical transaction code (LTAC name) of the job-submitting service.

5. In the **KCPA** field, you may have to identify a job-submitting application depending on the type of addressing:
  - enter the logical name of the job-receiving application in the case of double-step addressing, i.e. the LPAP name, OSI-LPAP name or master LPAP name of an OSI-LPAP or LU6.1-LPAP bundle.
  - enter blanks for single-step addressing (the name of the job-receiving application is in this case taken from the generation statement LTAC).

The partner application entry in KCPA has priority over the partner application specified in the LTAC statement at generation.

6. In the **KCPI** field, assign the service ID to be used by the job-submitting service in the MPUT, MCOM, FPUT, DPUT or MGET calls to address the job-receiving service. The service ID must begin with the character ">".
7. The **KCOF** field contains the functions permitted for communication with an OSI TP partner (irrelevant for communication via LU6.1 protocol).  
Possible values:

- **B** (basic functions)  
Basic functions
- **H** (handshake functions)  
Basic and handshake functions (only possible with APRO DM)
- **C** (chained transactions)  
Basic and commit functions with chained transactions
- **O** (other combination)  
The functions are selected via the second parameter area, i.e. if KCOF=O is specified, a second parameter area must be passed during the KDCS call.



When addressing an OSI TP job receiver, you must specify binary zero for all unused fields of the KDCS parameter area.

You specify the following in the second parameter area:

8. In the **KCVERS** field, enter the version number of the data structure: this is 1 in this openUTM version.
9. In the **KCFUPOL** field, specify whether the "polarized control" functional unit should be selected. Since "shared control" is not supported in this version, the only permitted entry is "Y".

10. In the **KCFUHSH** field, specify whether the "Handshake" functional unit should be selected (Y/N).
- If an asynchronous service is addressed (APRO AM), then "N" must be specified for KCFUHSH.
11. In the **KCFUCOM** field, specify whether the "Commit" functional unit should be selected (Y/N). The "Commit" functional unit can only be selected if the addressed OSI-LPAP partner contains the abstract syntax CCR in the application context (**C**ommitment, **C**oncurrency and **R**ecovery).
12. In the **KCFUCHN** field, specify whether the "Chained Transactions" functional unit should be selected. This field is only relevant if the "Commit" functional unit has also been selected, i.e. if KCFUCOM=Y is set.
- Since "Unchained Transactions" are not supported in this version, the only permitted entry here is "Y" if the "Commit" functional unit was selected.
- If the "Commit" functional unit was **not** selected, the KCFUCHN field is irrelevant. In this case, enter a blank.
13. In the **KCSECTYP** field, enter the security type.
- The security type controls whether SIGNON data (user ID and password) is transferred to the job-receiving service:
- N (none)  
No SIGNON data is transferred to the job-receiving service.
  - S (same)  
The user ID under which the local service runs, is transferred to the job-receiving service.
  - P (program)  
The values specified in the KCUSERID and KCPSWORD fields are transferred to the job-receiving service as the user ID and password.
- You can only select the security types "S" or "P", if the addressed OSI-LPAP partner in the application context contains the abstract syntax UTMSEC.
14. In the **KCUIDTYP** field, you enter the data type of the user ID specified in the KCUSERID field:
- P The string entered in KCUSERID is a "printable string" type.
  - T The string entered in KCUSERID is a "T61 string" type.
  - O The string entered in KCUSERID is an "octet string" type.
- For the range of values for these data types refer to the openUTM manual "Generating Applications", KDCDEF statement LTAC.

15. In the **KCUIDLTH** field, you enter the length of the user ID specified in the KCUSERID field (in bytes, 16 maximum).
16. In the **KCUSERID** field, you specify the user ID which, if KCSECTYP=P is set, will be transferred to the job-receiving service.
17. In the **KCPWDTYP** field, you enter the data type of the password specified in the KCPSWORD field:
  - P The string entered in KCPSWORD is a "printable string" type.
  - T The string entered in KCPSWORD is a "T61 string" type.
  - O The string entered in KCPSWORD is an "octet string" type.For the range of values for these data types refer to the openUTM manual "Generating Applications", KDCDEF statement LTAC.
18. In the **KCPWDLTH** field, you enter the length of the password specified in the KCPSWORD field (in bytes, 16 maximum). If no password is to be transferred, enter zero.
19. In the **KCPSWORD** field, you enter the password which, if KCSECTYP=P is set, will be transferred to the job-receiving service.



Any field of the 2nd parameter area which is not used must be filled with blanks. Exception: Any length field which is not used must be filled with zeroes.

For the KDCS call you enter:

20. As 1st parameter: the address of the KDCS parameter area.  
As 2nd parameter (if required): the address of the second parameter area (selection of special OSI TP function combinations).
21. The use of C/C++ calls is described in detail in [section "C/C++ macro interface" on page 497](#).

openUTM returns:

22. in the **KCRCCC** field: the KDCS return code, see next page.
23. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").

### KDCS return codes in the KCRCCC field

The following codes can be analyzed in the program:

- 000 Function carried out.
- 40Z UTM cannot perform the function. There is a generation error or a system error, or the APRO function was called although the application was generated without distributed processing, or no connection to the partner application possible at the moment (see value of KCRCDC).
- 41Z Impermissible APRO call. This can be, for example, for one of the following reasons:
  - APRO call was issued in a sign-on service.
  - In a service, communication with certain partners is to be performed via LU6.1 and with others via the OSI TP protocol using the "Commit" functional unit.
  - In a distributed transaction using the OSI TP protocol, associations are to be established via more than one local ACCESS-POINT.
  - An association with a "Commit" functional unit is to be established. However, the abstract syntax CCR (**c**ommitment, **c**oncurrency and **r**ecovery) is **not present** in the application context of the associated OSI-LPAP partner.
  - An "S" or "P" security type association is to be established. However, the abstract syntax UTMSEC is **not present** in the application context of the associated OSI-LPAP partner.
- 42Z Entry in KCOM invalid
- 43Z Entry in KCLM invalid.
- 44Z Value in KCRN does not identify a valid LTAC of a job-receiving service. Possible reasons:
  - configuration does not recognize the associated LTAC
  - LTAC is locked
  - user ID has no keycode for the LTAC
  - entry in KCOM does not match the specified LTAC (value DM in KCOM and LTAC of an asynchronous service or a TAC queue; or value AM in KCOM and LTAC of a dialog service).
- 46Z Entry in KCPA invalid, i.e. no partner application is generated under the specified name, or blanks were entered in KCPA and there is no application name defined in the LTAC generation statement.
- 47Z KCOF=O was specified. However the 2nd parameter area is missing or invalid.
- 48Z Invalid data structure version.

55Z Entry in KCPI invalid (does not begin with ">"), or service ID already assigned by job-submitting service.

58Z Value in KCOF invalid

Additional error codes can be found in the dump:

71Z INIT call missing or was given the form DM in the MSGTAC program.

89Z When addressing an OSI TP job receiver, unused fields of the KDCS parameter area were not specified binary zero.

### Features of the APRO call

- A service which communicates with a partner application via OSI TP protocol using the "Commit" functional unit cannot communicate with another partner application via LU6.1.
- If the OSI TP protocol is used in a service which takes part in a distributed transaction part, all job receivers must be allocated to the same local ACCESS-POINT and, if necessary, this ACCESS-POINT must be identical to the ACCESS-POINT used for the communication with a job-submitting service.
- Security types P and S can be selected both with APRO DM and with APRO AM. In the case of dialog services, the job submitter is signed on by means of the transferred user ID when the service is started and signed off again when it terminates. In the case of asynchronous services, the transferred user ID remains active only until the job has been entered in the appropriate queue.  
For dialog services, the job submitter is rejected when there is already an OSI TP dialog service running under this name, when a user is logged on through an LTERM partner and it is prohibited for users to sign on to the application more than once (SIGNON statement, MULTI-SIGNON=NO) or when the user ID has been configured with RESTART=YES and the functional unit commit has not been selected.  
The job submitter can always sign on under the user ID passed in order to be able to place a job in a queue for asynchronous services.

If security type "P" is selected in the 2nd parameter area, you must specify user ID and password. Additionally, the data type of user ID and password must also be specified. Possible data types are Printable String, T61 String and Octet String. For the range of values for these data types, refer to the openUTM manual "Generating Applications", KDCDEF statement LTAC.

If Octet String is specified, no code conversion is performed. The Octet String type is necessary when you transfer passwords which were generated in Hex-String (PASSWORD=X'aabbcc..') format.

If security type "S" is selected, openUTM transfers the user ID under which the local service is running, to the job-receiving service. It is assumed that T61 String is used.



- A successful APRO DM call means that a virtual connection has been established with the job-receiving application.  
A successful APRO call does not mean that it is possible to exchange messages with the job-receiving service. A session or association is not reserved until the end of the program unit run in which the first message was sent to the job-receiving service.
- If at the time of the APRO call no connection to the remote application has been established, UTM initiates the connection setup.
- If openUTM returns a KDCS error code  $\neq$  000 as a result of an APRO call no message should be sent with MPUT to the job receiver, because the service would then be aborted with KCRCCC=74Z. If the return code  $\neq$  000 it is advisable to output an appropriate message to the terminal user or administrator.
- A message has to be sent to a job-receiving service addressed by APRO DM in the same transaction, otherwise openUTM aborts the service with KCRCCC=87Z at PEND.
- An asynchronous job must be associated with an asynchronous service addressed by APRO AM before the next PEND call, otherwise openUTM aborts the service with 86Z at PEND.
- A RSET call only resets the address of a dialog service on the condition that no dialog message has yet been sent to the job-receiving service (MPUT with subsequent PEND).
- If multiple job-submitting services exist simultaneously in one application, these may use identical service identifiers. These may also be used for addressing differing job-receiving services.
- If a job complex is to be sent to the remote application, the APRO AM call must precede the MCOM BC call and the service identifier for the MCOM BC call must be entered in the KCRN field.
- Life of a service ID

A service ID created with APRO DM is relevant for transaction security, i.e. it forms part of transaction-logged security. It is present until the job-receiving service is terminated and is not released until the end of the transaction in which the job submitter has read the message from the job receiver.

A service identifier created by APRO AM is released

- after a successful FPUT/DPUT NE call,
- at the next RSET or PEND call,
- after 40Z has been returned by an FPUT/DPUT call, or
- with job complexes with that service identifier: following an MCOM EC call or after 40Z has been returned by an MCOM BC or DPUT call.

Once a service ID has been released, it can be used for the next job receiver/submitter relationship.

- Addressing a Master LPAP

If a master LPAP is addressed with the APRO call, a slave LPAP from the LPAP bundle is selected if this is the first APRO call of the current transaction for this master LPAP.

- With an APRO DM call, the first slave LPAP is selected to which a logical connection has been established. If no logical connection has been established to any slave LPAP, the return value is 40Z/KD10.
- With an APRO AM call, the first slave LPAP is selected to which a logical connection has been established. If no logical connection has been established to any slave LPAP, one of the slave LPAPs is selected.

For every slave LPAP checked during selection, and to which no association or session is established, establishment of an association or session is initiated.

If a slave LPAP has been selected, the same slave LPAP is used for every additional APRO call in this transaction addressed to the master LPAP.

A subsequent APRO DM call can return 40Z/KD10 if the first APRO call was APRO AM and no association or session had previously been established to any of the slave LPAPs or if the logical connection has been cleared again in the interim.



For more detailed information on message distribution, see the openUTM manual "Generating Applications".

## CTRL Control OSI TP Dialog

The CTRL (control) function call is used for distributed processing via the OSI TP protocol. It allows you to explicitly control a dialog with an OSI TP partner.

The CTRL PR and CTRL PE calls may only be addressed to job-receiving services for which the "Commit" functional unit was selected.

There are a number of variants of the CTRL call:

- **CTRL PR (Prepare to Commit)**  
CTRL PR requests the job-receiving service to initiate the end of transaction. The local service sends an MPUT call to the partner whether it wants to continue to receive data from the remote service in the current transaction.
- **CTRL PE (Prepare End Dialogue)**  
CTRL PE requests the job-receiving service to initiate the end of dialog. The local service sends an MPUT call to the partner whether it wants to continue to receive data from the remote service in the current transaction.
- **CTRL AB (Abort Dialogue)**  
CTRL AB initiates an abnormal termination of the current dialog with the job-receiving service. MPUT calls to this job receiver are deleted and not sent. If the Commit functionality is selected for the dialog, openUTM ensures that the distributed transaction is reset to the last consistency point before the dialog is terminated. A separate CTRL call must be issued for each dialog which is to be terminated abnormally.

### Setting the 1st parameter

The table below shows the various options and associated specifications in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area				
	KCOM	KCLA	KCLM	KCRN	KCMF/kcfn
Prepare to Commit	"PR"	0	0	service ID	blanks
Prepare End Dialogue	"PE"	0	0	service ID	blanks
Abort Dialogue	"AB"	0	0	service ID	blanks

### Setting the 2nd parameter

Here you have to specify the address of the message area. The message area is not used in this version of openUTM. It is intended for future extensions.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"CTRL"
2.	KCOM	"PR" / "PE" / "AB"
3.	KCLA	0
4.	KCLM	0
5.	KCRN	Service ID
6.	KCMF / kcfn	Blanks

**KDCS call**

	1st parameter	2nd parameter
7.	KDCS parameter area	Message area

8.

**C/C++ macro calls**

Macro names	Parameter
KDCS_CTRLPR / KDCS_CTRLPE / KDCS_CTRLAB	(kcrn)

**openUTM return information**

	Field name in KB return area	Contents
9.	KCRCCC	Return code
10.	KCRCDC	Internal return code

For the CTRL call the following entries are required in the KDCS parameter area:

1. In the **KCOP** field, you must enter the CTRL operation code.
2. The **KCOM** field must contain one of the following operation modifiers:
  - **PR** (PRepare to commit)  
This variant requests the job-receiving service to initiate the end of transaction. The local service sends an MPUT call to the partner whether it wants to continue to receive data from the remote service in the current transaction.
  - **PE** (Prepare End dialogue)  
This variant requests the job-receiving service to initiate the end of dialog. The local service sends an MPUT call to the partner whether it wants to continue to receive data from the remote service in the current transaction.
  - **AB** (ABort dialogue)  
This variant initiates an abnormal termination for the current dialog with a job-receiving service. If the Commit functionality is selected for the dialog, openUTM ensures that the distributed transaction is reset to the last synchronization point using an appropriate PEND call before the dialog is terminated. A separate CTRL call must be issued for each dialog which is to be terminated abnormally. With CTRL AB, messages sent to the partner using MPUT are deleted.
3. The **KCLA** field must be set to zero.
4. The **KCLM** field must be set to zero.
5. In the **KCRN** field, specify the service ID (VGID) of the partner service to which the CTRL call refers.
6. In the **KCMF/kcfn** field, enter blanks.

You must specify binary zero for all unused fields.

For the KDCS call you enter:

7. As 1st parameter: the address of the KDCS parameter area.  
As 2nd parameter: the address of the message area. This address must be specified for all CTRL calls even though CTRL calls do not currently use the message area.
8. The use of C/C++ calls is described in detail in [section “C/C++ macro interface” on page 497ff.](#)

openUTM returns:

9. in the **KCRCCC** field: the KDCS return code.
10. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### **KDCS return codes in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Function carried out successfully.
- 40Z The application was generated without distributed processing.
- 41Z CTRL call is impermissible at this point. Possible reasons are:
  - The call was issued for an asynchronous service, i.e. the specified service ID was defined using an APRO AM call.
  - The call is addressed to a partner to which an MPUT HM has already been sent.
  - A CTRL PE or PR is addressed to a partner for which FU commit has not been selected.
  - A CTRL PE or PR is addressed to a partner to which no MPUT has as yet been sent following an APRO.
- 42Z The function variant in KCOM is invalid.
- 43Z The value specified in KCLA or KCLM is invalid.
- 44Z The service ID specified in KCRN is invalid or no service ID has been specified.
- 45Z The KCMF/kcfn field is not filled with blanks.
- 71Z No INIT call has been issued in the program unit run.
- 77Z Invalid area address.
- 89Z Fields that are not used by the call have not been set to binary zero.

**Features of the CTRL calls**

- No end of transaction may be requested at the end of a processing step in which a CTRL PR or PE call and an MPUT call have been addressed to the same partner.
- You may address the CTRL PR / PE / AB call only to those job-receiving services which are currently involved in a distributed dialog, i.e. which were addressed with an APRO DM.
- You may address CTRL PR and CTRL PE calls only to those job-receiving services for which the Commit functional unit has been selected and for which an MPUT has already been issued after APRO.
- You can issue CTRL calls for multiple partners in a single processing step or program unit run.
- Only those PEND calls with the operation modifiers RS, FR and ER are allowed after a CTRL AB call for a dialog with an AN service in which the commit functionality has been selected.

## DADM Administer message queues

The DADM (delayed free message administration) call provides the following functions for administering message queues that enable you:

- to read summary information about messages in a message queue into the message area (user identification, job identification (job ID), creation time, starting time and existing confirmation jobs, original destination ...)
- to read user information into the message area that has been generated with DPUT NI/QI/+I/-I. The user information can only be read if the confirmation job becomes the main job.
- to change the processing order of messages in a message queue
- to delete individual messages or all the messages from the message queue
- to assign either selected messages or all messages in the dead letter queue each to their original destination or to a new destination

You can only administer message queues of the local node application with a UTM cluster application.

The format of the DADM call is discussed in detail below. For further information on the administration of the message queue refer to the openUTM manual “Administering Applications”.



## Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area							
	KCOP	KCOM	KCLA	KCRN	KCLT	KCQTYP	KCMOD	Time <sup>1</sup>
Read overview information	"DADM"	"RQ"	53	Job ID/ blank	LTERM/ (OSI-) LPAP/ TAC/ Queue	Binary zero  "Q"/"U"	Binary zero	Binary zero
Read user information	"DADM"	"UI"	Length	Job ID	Binary zero	Binary zero	Binary zero	Time point (absolute)
Change order	"DADM"	"CS"	0	Job ID	Binary zero	Binary zero	Binary zero	Time point (absolute)
Delete individual message from a queue	"DADM"	"DL"	0	Job ID	LTERM/ (OSI-) LPAP/ TAC/ Queue	Binary zero  "Q"/"U"	"C"/"N"	Time point (absolute)
Delete all messages from a queue	"DADM"	"DA"	0	Blank	LTERM/ (OSI-) LPAP/ TAC/ Queue	Binary zero  "Q"/"U"	Binary zero	Binary zero
Move single message	"DADM"	"MV"	0	Job ID	TAC/ Blank	Binary zero	Binary zero	Time point (absolute)
Move all messages	"DADM"	"MA"	0	Blank	TAC/ Blank	Binary zero	Binary zero	Binary zero

<sup>1</sup> The time point is specified in the fields KCTAG/kcday, KCSTD/kchour and KCMIN.

All fields of the parameter areas not in use must be assigned the value binary zero.

## Setting the 2nd parameter

Here you have to supply the address of the message area into which openUTM is to read the message. A language-specific data structure enables you to structure the message area when calling DADM RQ, in COBOL this is the KCDADC COPY element and in C/C++ the *kcdad.h* include file.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"DADM"
2.	KCOM	"RQ"/"UI"/"CS"/"DL"/"DA"/"MV"/"MA"
3.	KCLA	Length in bytes/0
4.	KCRN	Job ID/blanks
5.	KCLT	LTERM name/TAC/queue/binary zero/ Blank
6.	KCQTYP	Destination type:"Q"/"U"/binary zero
7.	KCMOD	"C"/"N"/binary zero
8.	KCTAG/kcday	Day (absolute)/binary zero
8.	KCSTD/kchour	Hour (absolute)/binary zero
8.	KCMIN	Minute (absolute)/binary zero
8.	KCSEK/kcsec	Second (absolute)/binary zero

**KDCS call**

	1st parameter	2nd parameter
9.	KDCS parameter area	Message area

10. **C/C++ macro calls**

Macro name	Parameters
KDCS_DADMRQ	(nb,kcla,kcrn,kclt)
KDCS_QADMRQ	(nb,kcla,kcrn,kclt,kcqtyp)
KDCS_DADMUI	(nb,kcla,kcrn,kcday,kchour,kcmin,kcsec)
KDCS_DADMCS	(nb,kcrn,kcday,kchour,kcmin,kcsec)
KDCS_DADMDL	(nb,kcrn,kclt,kcmod,kcday,kchour, kcmin,kcsec)
KDCS_QADMDL	(nb,kcrn,kclt,kcmod,kcday,kchour,kcmin, kcsec,kcqtyp)
KDCS_DADMDA	(nb,kclt)
KDCS_QADMDA	(nb,kclt,kcqtyp)

10. **C/C++ macro calls**

Macro name	Parameters
KDCS_DADMMV	(nb,kcrn,kclt,kcday,kchour,kcmin,kcsec,kcqtyp)
KDCS_DADMMMA	(nb,kclt,kcqtyp)

**openUTM return information**

	Message area	Contents
11.	<input type="text"/> Field name in the KB return area	<input type="text"/>
12.	KCRMLM	Actual length
13.	KCRCCC	Return code
14.	KCRCDC	Internal return code
15.	KCRMF/kcrfn	Job ID/blanks

For the DADM call you make the following entries in the appropriate fields of the KDCS parameter area:

- In the **KCOP** field, enter the DADM operation code.
- In the **KCOM** field, select the operation modifier:
  - RQ to read the overview information on a message queue
  - UI to read the user information of a main job created by means of DPUT NI/QI
  - CS to give preference to a specific message
  - DL to delete a single message
  - DA to delete all messages in the message queue
  - MV to move a single message from the dead letter queue into either the original message queue, any asynchronous TAC or any TAC queue
  - MA to move each message from the dead letter queue either into their original queues, any asynchronous TAC or any TAC queue
- In the **KCLA** field, enter the length of the data to be transferred to the message area. For KCOM = RQ enter 45, for KCOM = CS/DL/DA/MV/MA enter 0.

4. In the **KCRN** field, specify the message in the queue to be administered. The following specifications are required:
  - blanks  
if KCOM = DA/MA or if KCOM = RQ and the call refers to the first message in the queue
  - the job ID  
if KCOM = UI/CS/DL/MV or if KCOM = RQ and the call refers to the subsequent messages in the queue. The job ID is always returned to the KCRMF/kcrfn field in the preceding DADM RQ.
5. In the **KCLT** field, identify the queue. That means:
  - If KCOM = RQ/DL/DA, specify:
    - the LTERM name if the message was for an LTERM partner
    - the (MASTER-)(OSI-)LPAP name if the messages was for an (OSI-)LPAP partner
    - the TAC if the message was for an asynchronous program
    - the name of the queue if you want to administer messages of a USER queue, a TAC queue or a temporary queue
  - If KCOM = UI/CS, specify binary zero.
  - If KCOM = MV/MA, specify the following:
    - the TAC, if the single message or all messages are to be directed to an asynchronous program,
    - the name of a TAC queue, if the single message or all messages are to be directed to a service-controlled queue,
    - blank, if the single message or all messages are each to be assigned back to their original destination.
6. In the **KCTYP** field, specify the type of the queue:
  - If KCOM = RQ/DL/DA, specify:
    - binary zero, if the queue belongs to an LTERM, an (OSI-)LPAP or a TAC, or if it is a TAC queue,
    - Q in the case of a temporary queue created with QCRE,
    - U in the case of a USER queue.
  - If KCOM=UI/CS/MV/MA enter binary zero.
7. In the **KCMOD** field, specify whether openUTM should activate the negative confirmation job when a job complex is deleted. The following values are possible:
 

binary zero  
if KCOM = RQ/UI/CS/DA/MV/MA,

  - C to delete the complete job if KCOM = DL; in the case of job complexes all the confirmation jobs are deleted also,
  - N to activate the negative confirmation job if KCOM = DL; the message itself is deleted.

8. If KCOM = UI/CS/DL/MV, enter the time the message was generated as follows: in the **KCTAG/kcday** field, the day in the year (working day, value 001 to 366), in **KCSTD/kchour** the hour, in the **KCMIN** field the minute and in **KCSEK/ kcsec** the second. This time can be ascertained before with DADM RQ.

If KCOM = RQ/DA/MA enter binary zero.

For the KDCS call you enter:

9. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area into which openUTM is to read the message. You have to enter the address even if you have entered 0 in KCLA.
10. The use of C/C++ calls is described in detail in [section "C/C++ macro interface" on page 497ff.](#)

openUTM returns:

11. If KCOM = RQ/UI: the message in its actual or at most in its desired length in the specified message area.
12. In the **KCRLM** field: the actual length of the message, possibly deviating from the length requested in KCLA of the parameter area.
13. In the **KCRCCC** field: the KDCS return code.
14. In the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").
15. In the **KCRMf/kcrfn** field: if KCOM = RQ the job ID of the next message in the queue (see KCLT) or blanks for the last message in the queue.

### KDCS error codes for the DADM call

The following codes can be analyzed in the program:

000 Operation executed (if KCOM = RQ/UI) or the administration job accepted (if KCOM = CS/DL/DA/MV/MA). The real execution is not determined until the end of the transaction.

A decision about actual execution is not made until the end of the transaction (see [“Features of the DADM call” on page 232](#)).

01Z Length conflict: KCLA < KCRLM, the message is truncated.

40Z The system cannot perform the operation (generation error or system error, original destination no longer exists, no administration privileges, locked by another service), see KCRCDC.

42Z Entry in KCOM invalid.

43Z Length specified in KCLA is negative or invalid.

44Z Job ID specified in KCRN is invalid.

46Z Entry in KCLT is invalid. Possible causes:

- The specified LTERM or (MASTER-)(OSI-)LPAP name does not exist.
- The TAC is invalid or locked.
- The TAC is not a process TAC or a dialog TAC.
- There is no USER queue or temporary queue with the specified name, or the type specified in KCTYP does not fit the queue name.
- with KCOM=MV/MA: an LTERM, (OSI-)LPAP or maste LPAP name was specified, the specified TAC has been deleted or KDCMSGTC or KDCDLETQ was specified.
- with KCOM=MV: a blank was specified but the original destination of the message has been deleted.

47Z Message area missing or cannot be written in the specified length.

49Z Contents of unused fields of KDCS parameter area not equal to binary zero.

56Z Value in KCMOD or time entry in KCTAG/kcday, KCSTD/kchour, KCMIN or KCSEK/ksec is invalid.

An additional return code can be found in the dump:

71Z INIT missing in this program.

### Return information in the message area for DADM RQ

There is a data structure available for the return information for the DADM RQ call, for COBOL the KCDADC COPY element and for C/C++ the *kcdad.h* include file. This data structure can be used to define the message area and has the following structure:

Byte	Field name COBOL/C/C++	Meaning
1 - 8	KCDAGUS	UTM user ID of the job submitter
9 - 16	KCDADPID	Job ID (assigned by UTM)
17 - 25	KCDAGTIM <sup>1</sup>	Time of the FPUT/DPUT call in the form <i>dddhhmmss</i> :
17 - 19	KCDAGDOY	<i>ddd</i> Day in the year (value range 000 - 366)
20 - 21	KCDAGHR	<i>hh</i> Hour (value range 00 - 23)
22 - 23	KCDAGMIN	<i>mm</i> Minute (value range 00 - 59)
24 - 25	KCDAGSEC	<i>ss</i> Second (value range 00 - 59)
26 - 34	KCDASTIM <sup>1</sup>	For time-delayed jobs enter the desired starting time in the form <i>dddhhmmss</i> :
26 - 28	KCDASDOY	<i>ddd</i> Day in the year (value range 000 - 366)
29 - 30	KCDASHR	<i>hh</i> Hour (value range 00 - 23)
31 - 32	KCDASMIN	<i>mm</i> Minute (value range 00 - 59)
33 - 34	KCDASSEC	<i>ss</i> Second (value range 00 - 59)
		For a job without time delay you enter blanks.
35	KCDAPMSG	Y positive confirmation job exists N no positive confirmation job exists
36	KCDANMSG	Y negative confirmation job exists N no negative confirmation job exists
37 - 44	KCDADEST	Destination of the message or the original destination for dead letter queue
45	KCDATYPE	Type of the message destination or type of original destination for dead letter queue
		Q for temporary queue U for USER queue T for TAC queue A for asynchronous TAC L for LTERM
46 - 53	KCDAFCTM	Generation time or time DPUT converted to FPUT in the message. In the case of service-driven queues, the messages displayed with DADM RQ can be unambiguously assigned to the messages read with DGET BF. KCDADPID or KCDAFCTM must correspond to the return values KCRDPID or KCRGTM of the corresponding DGET BF call.

<sup>1</sup> For C/C++, the KCDAGTIM and KCDASTIM summary fields are not defined. However the specific fields for day/hour/minute/second are defined.

## Features of the DADM call

- Ascertaining the job ID

The job ID is assigned internally by UTM. It can be ascertained in the program unit as follows with a DADM RQ call: for the first DADM RQ call (with KCRN = blanks) openUTM supplies information on the first message in the queue. On this occasion openUTM writes the job ID amongst other things to the message area. If there are more messages for the queue in the message queue, openUTM in each case returns the job ID of the next message to KCRMF/kcrfn. When the last message in the queue is reached, openUTM enters blanks in the KCRMF/kcrfn field.
- DADM CS/DL/DA/MV/MA calls (change order, delete or move) are transaction-logged and are not executed until the end of the transaction. After such a call, therefore, KCRCCC = 000 does not necessarily guarantee that the call can be executed successfully, because the message in the queue could be deleted in the meantime by another DADM call. You can check whether a DADM CS/DL/DA/MV/MA call has been successful by issuing a DADM RQ call in a subsequent transaction.
- The DADM CS call puts the relevant message in first position in the queue. In the case of time-driven messages whose starting point has not yet been reached, the call is rejected with 40Z.
- After a delete job with DADM DL/DA no more delete jobs can be submitted within the same transaction and no DADM CS call; openUTM rejects this with 40Z.
- Jobs that are currently being processed cannot be administered. This applies, for example, if messages in a USER queue or a temporary queue are currently being read. If these jobs are destined for printers, printing can be stopped by means of a PADM call.
- A program unit which issues a DADM call must be running under a user ID with administration privileges, otherwise openUTM acknowledges the DADM call with 40Z. There is the following exception:

If a service is started from a printer control LTERM and if only such jobs are administered which are directed to printers of the printer control LTERM, then neither the program unit nor the user need the administration privileges.
- The dead letter queue consists of messages that could not be processed. To be able to process these messages, possibly after eliminating the error, they must either be assigned to their original destination or to a new destination. You can use DADM MV to move a single message or DADM MA to move all messages from the dead letter queue into a specific queue or the original queue. A previously defined QLEV and the STATUS of the recipient queue is thereby ignored. With DADM MA and KCLT= blank, messages whose original destination no longer exists stay in the dead letter queue. However, this is not indicated by a return code.



## DGET Read a message from a service-controlled queue

The call DGET (DATA GET) is used to read a message or message segment into the message area from a service-controlled queue. The following are service-controlled queues: TAC queues, USER queues and temporary queues

DGET provides several ways of reading the messages of a queue:

- sequential processing (DGET FT/NT)
- browsing (DGET BF/BN)
- selective processing (DGET PF/PN)

With sequential or selective processing the message is read and then deleted from the queue, whereas with browsing the message remains in the queue. For the dead letter queue only browsing (read without delete) is allowed.

Each variant also allows several message segments to be read. The first segment is read using DGET FT/BF/PF (first). Subsequent segments are read within the same program unit run using DGET NT/BN/PN (next) without an intervening PGWT call.

In this way, as many message segments as have been sent with DPUT QT can be read. Each message segment sent with DPUT QT must be read using a separate DGET.

Message segments that have not been read are lost

- when a new message is read with DGET FT,
- when PGWT is called,
- when the program unit run is terminated.

If the transaction is reset, processed messages are placed back in the queue (redelivery) and can then be read and processed again using DGET. The maximum number of redeliveries can be set in the generation (MAX REDELIVERY=). Once this limit has been reached, then, depending on the TAC statement's DEAD-LETTER-Q parameter for the generation, the processed message is either deleted at the end of the transaction or is saved in the dead letter queue unless (in the case of message complexes) no negative acknowledgment job has been defined.

Messages from USER or temporary queues cannot be saved in the dead letter queue. They are therefore lost after the maximum number of redeliveries.

The format of the DGET call is described in detail in the following. You will find more information on the subject of message queues in [section "Message Queuing \(asynchronous processing\)" on page 50](#).

### Setting the KDCS parameter area (1st parameter)

The following tables show the different options and the entries that have to be made in the KDCS parameter area.

Sequential processing									
Function of the call	Entries in the KDCS parameter area								
	KCOP	KCOM	KCLA	KCMF/ kcfn	KCRN	KCQTYP	KCWTIME	KCQRC	KCDPID
Read new message or first message segment from queue	"DGET"	"FT"	Length	Blank	Queue name	Type of the queue	Wait time in seconds	0	Binary zero
Read next message segment from queue	"DGET"	"NT"	Length	Blank	Queue name	Type of the queue	0	0	Binary zero

Browsing									
Function of the call	Entries in the KDCS parameter area								
	KCOP	KCOM	KCLA	KCGTM	KCRN	KCQTYP	KCWTIME	KCQRC	KCDPID
Read first message segment of the first or next message	"DGET"	"BF"	Length	Blank/ creation time*	Queue name	Type of the queue	Wait time in seconds	-1 / redelivery counter	Blank/ DPUT ID
Read next message segment	"DGET"	"BN"	Length	Creation time*	Queue name	Type of the queue	0	0	DPUT ID

Selective processing									
Function of the call	Entries in the KDCS parameter area								
	KCOP	KCOM	KCLA	KCGTM	KCRN	KCQTYP	KCWTIME	KCQRC	KCDPID
Process first message segment of a specific message	"DGET"	"PF"	Length	Creation time*	Queue name	Type of the queue	0	0	DPUT-ID
Process next message segment	"DGET"	"PN"	Length	Creation time*	Queue name	Type of the queue	0	0	DPUT-ID

\* Value in the KB return area KCRGTM of the preceding DGET BF

## Setting the 2nd parameter

Here you have to make available the address of the message area to which openUTM is to read the message.

### Setting the parameters

	Field name in the KDCS parameter area	Contents
1.	KCOP	"DGET"
2.	KCOM	"FT"/"NT"/"BF"/"BN"/"PF"/"PN"
3.	KCLA	Length in bytes
4.	KCMF/kcfn/ KCGTM	Blanks Blanks/creation time
5.	KCRN	Name of the queue
6.	KCQTYP	"T"/"U"/"Q"
7.	KCWTIME	Wait time in seconds/0
8.	KCQRC	0/-1/redelivery counter
9.	KCDPID	Binary zero/blank/DPUT ID

### KDCS call

	1st parameter	2nd parameter
10.	KDCS parameter area	Message area

### 11. C/C++ macro call

Macro name	Parameter
KDCS_DGETFT / KDCS_DGETNT	(nb,kcla,kcfn,kcrn,kcqtyp,kcwtime,kcft1, kcft2)
KDCS_DGETBF / KDCS_DGETBN / KDCS_DGETPF / KDCS_DGETPN	(nb,kcla,kcrn,kcqtyp,kcwtime,kcqrc,kcgtm, kcdpid)

## Returns from openUTM

	Message area	Contents
12.	<input type="text"/> Field name in KB return area	Data
13.	KCRLM	Actual length
14.	KCRMF (DGET FT/NT only)	In the case of OSI TP partner: name of the abstract syntax
15.	KCRWVG (DGET FT only)	Number of services that are waiting
16.	KCRUS (DGET FT only)	UTM user ID of the message creator
17.	KCRQRC (DGET BF only)	Queue-specific redelivery counter
18.	KCRGTM (DGET BF only)	Creation time of the message read
19.	KCRDPID (DGET BF only)	DPUT ID of the message read
20.	KCRRRC (DGET FT/BF/BN/PF only)	Redelivery counter of the message read
21.	KCRCCC	Return code
22.	KCRCDC	Internal return code

In the KDCS parameter area you make the following entries for the DGET call:

1. In the **KCOP** field, enter the DGET operation code.
2. In the **KCOM** field, enter:
  - FT to process the first message segment of the first/new message
  - NE to process a further message segment of the first/new message
  - BF to read the first message segment of a message (browse without deleting)
  - BN to read a further message segment of the message (browse without deleting)
  - PF to process the first message segment of a specific message
  - PN to process a further message segment of a specific message

3. In the **KCLA** field, specify the length of the message area to which the message is to be read. openUTM enters the length of the message segment that has actually been read in the KCRLM return field.  
  
If the message and all its message segments is not to be read, you can specify the value 0 here for DGET FT. A subsequent DGET NT is then rejected with the return code 10Z.
4. The **KCGTM** or **KCMF/kcfn** field must be supplied as follows:
  - with blank for KCOM = FT/NT.
  - with blank for KCOM = BF if the first segment of the first message of the queue is to be read.
  - with the creation time of the message for KCOM = BN/PF/PN or if, for KCOM = BF, the next message is to be read. The creation time is returned in the KCRGTM field for the last DGET BF call.
5. In the **KCRN** field, specify the name of the queue from which the message is to be read.
6. In the **KCQTYP** field, specify the type of the queue:
  - T for a TAQ queue
  - U for a USER queue
  - Q for a temporary queue
7. In the **KCWTIME** field, specify for DGET FT/BF the maximum number of seconds to be waited for the arrival of a message. If you specify 0, there will be no wait. If a wait time is specified, the subsequent program unit when PEND is specified must be assigned to a TAC class.  
  
In the case of DGET NT/BN/PF/PN you must specify 0.
8. In the **KCQRC** field, specify for DGET BF the behavior after a transaction has been processed and reset. You can specify:
  - either the value returned in the KCRQRC field for the previous DGET BF call. This ensures that all messages of the queue are always read. A message that has already been processed may be read again after the transaction has been reset.
  - or the value -1 or the constant KDCS\_NO\_QRC. This may mean that not all messages of the queue are read.  
You must specify 0 for DGET FT/NT/BN/PF/PN.

9. In the **KCDPID** field, specify:
  - binary zero for KCOM = FT/NT
  - blank if, for KCOM = BF, the first segment of the first message is to be read.
  - the DPUT ID for KCOM = PF/PN or if, for KCOM = BF/BN, the next message/message segment is to be read. The DPUT ID is returned in the KCRDPID field in the previous DGET BF call.

In the KDCS call you specify:

10. As the 1st parameter: the address of the KDCS parameter area.  
As the 2nd parameter: the address of the message area to which openUTM is to read the message. You also specify the address of the message area when you enter a length of 0 in KCLA.
11. The [section “C/C++ macro interface” on page 497](#) describes in detail how to use macro calls for C/C++.

openUTM returns:

12. In the specified message area the message segment in its actual length or, at most, the length of the message area.
13. In the **KCRLM** field the actual length of the message segment read providing a value > 0 was specified in KCLA.

For KCOM = FT/NT:

14. In the **KCRMF** field the name of the abstract syntax of the message segment read, provided the message originates from an OSI TP partner. Otherwise, it returns blanks.
15. In the **KCRWVG** field the number of services that are already waiting for messages from the specified queue (the current service is not counted). KCRWVG is only supplied in the case of DGET FT.
16. In the **KCRUS** field the UTM user ID under which the DGET message was created. KCRUS is only supplied in the case of DGET FT.

For KCOM = BF:

17. In the **KCRQRC** field, the queue-specific redelivery counter. This value is needed to supply the KCQRC field in the next DGET BF call.
18. In the **KCRGTM** field, the creation time (binary) of the message read. This value is needed to supply the KCGTM field in the next DGET BF/BN/PF/PN call.
19. In the **KCRDPID** field, the DPUT ID of the message read. This value is needed to supply the KCDPID field in the next DGET BF/BN/PF/PN call.

For KCOM = BF/BN/PF:

20. In the **KCRRC** field, the redelivery counter of the message read. This indicates how often the message was redelivered after the transaction was processed and reset. KCRRC can accept a maximum value of 254. If this value has been reached and if the number of redeliveries was not limited at generation, the value 254 is always returned after each additional DGET BF/BN/PF call.

For all variants:

21. In the **KCRCCC** field the KDCS return code (see next page).
22. In the **KCRCDC** field, the internal return code of openUTM (see openUTM manual "Messages, Debugging and Diagnostics").

### **KDCS return codes for the DGET call**

The following can be evaluated in the program:

- 000 The operation was executed.
- 01Z Length conflict: KCLA < KCRLM; the message was truncated because the message segment is longer than the message area.
- 04Z Not all of the message segments were read at the previous DGET call; as a result of the current DGET FT/BF/PF call, the message segments that have not yet been read are lost.
- 08Z In the case of reading with waiting (KCWTIME>0): there is currently no message.  
  
In this case, no more DGET calls are permitted and the program unit must be terminated with PEND PA/PR or a wait point must be set with PGWT PR. As soon as a message arrives, the maximum wait time elapses or the queue is deleted, openUTM continues the service with the next program unit or openUTM continues the service after the PGWT PR. A follow-up program unit must be assigned to a TAC class.
- 10Z In the case of DGET NT/BN/PN: all the message segments of the message have been read.
- 11Z In the case of reading without waiting (KCWTIME=0): there is no message.
- 40Z In the case of DGET NT/BN/PN: the name or type of the specified queue does not match the previous DGET call of the current program unit run. Message segments may have been lost. No message has been read.  
  
In the case of DGET FT/PF: there is a generation error (the value MAX ...,RECBUF=... is too low).

- 41Z The DGET call is made in the first part of the sign-on, which is impermissible, or the previous DGET call produced the return code 08Z. This means it is necessary to wait, and no further DGET calls are permitted in this program unit.
- 42Z Possible causes:
- The value in KCOM is invalid or does not match the previous DGET call (e.g. DGET PN after DGET BF)
  - or the first message segment was not read in this program unit run,
  - or a PGWT call was issued in the meantime.
- 43Z The value in KCLA or KCWTIME is negative or invalid, or KCWTIME was not supplied with 0 for DGET NT/BN/PF/PN.
- 44Z The value in KCRN is invalid. This means one of the following things:
- There is no queue with the specified name and type.
  - The queue was deleted.
  - The USER that started the service or the user's LTERM has no authorization (KSET) to read the queue.
  - The specified TAC is not generated with TYPE=Q.
  - The dead letter queue has been read in another way than with Browse (read without delete, i.e. DGET BF/BN).
- 45Z The value in KCMF/KCGTM is invalid:
- For DGET FT/NT: KCMF was not supplied with blanks.
  - For DGET BN/PF: there is no message with the specified DPUT ID and creation time, or the message was deleted in the meantime.
- 47Z The message area is missing, or the specified area address is invalid.
- 49Z Unused fields have a value that is not equal to binary zero.
- 53Z For DGET BF/PF: the value in KCDPID is not a valid DPUT ID or does not match the entries in KCRN and KCQTYP.
- For DGET BN/PN: the value in KCDPID or KCGTM does not match the corresponding value of the last DGET BF/PF call.
- 71Z An INIT has not yet been called in the program unit run.



## Features of the DGET call

- Message length

The actual message length is returned in the KCRLM field. The following applies:

- When  $KCRLM \leq KCLA$ , only KCRLM characters (bytes) are transferred to the message area. The contents of the rest of the message area are undefined.
- When  $KCRLM > KCLA$ , only KCLA characters are transferred to the message area. The rest ( $KCRLM - KCLA$ ) are lost and can no longer be read by means of a subsequent DGET.

In the description of the MGET call you will find an example that indicates what openUTM does when there are length conflicts.

- Browsing and processing

- During browsing (DGET BF/BN) messages can be read in parallel by several services.
- During message processing (DGET FT/NT/PF/PN) each message segment can be read once only. All message segments are deleted as soon as the transaction is terminated.
- If a message is to be read (DGET BF/BN) and then processed with DGET PF, the DPUT ID and creation time returned for DGET BF must be specified in the DGET PF call.

- Waiting for messages

In a dialog or asynchronous service DGET calls may be issued for different queues until it is necessary to wait for a message. In other words, if there is no message for a DGET call with waiting, this is indicated in the program by  $KCRCCC = 08Z$  ( $KCWTIME > 0$ ).

The following KDCS calls should be programmed, depending on whether a wait is required for a message.

- If a wait is required, the program unit must be terminated with PEND PA/PR so that it is possible to wait for the arrival of the message outside the program context, or a wait point must be set in the program unit using PGWT PR.
- If no wait is required, the transaction must be reset (RSET, PEND RS or PGWT RB) or the service must be terminated with PEND ER/FR.

Otherwise, the error code 72Z is returned for the PEND call.

- Continuing the program

If there is a wait for a DGET message, openUTM starts the next program unit or continues the program after the PGWT PR:

- a) as soon as a message arrives
- b) when the maximum wait time elapses or
- c) when the queue is deleted
- d) as soon as the processed message is redelivered (redelivery after transaction reset), providing DGET FT is used for waiting or KCQRC was supplied with the value of the KB return field KCRQRC by the previous browse call.

In a) to c) above all services waiting for messages in the queue are continued (not just the first waiting service).

In the case of redelivery (d)) only the services that are able to read redelivered messages are continued. This ensures that arriving or redelivered messages can be read in parallel by the services.

If the program is continued in a PEND PA/PR follow-up program unit, it must be assigned to a TAC class. In other words, TAC classes must have been generated if this functionality is to be available. If they have not been, the PEND call is rejected with KCRCC=74Z.

- Redelivery

If a transaction is reset, any processed message is placed back in the queue and can be read again. The application must have been generated accordingly and the generated maximum number of redeliveries must not have been reached. For more information refer to the openUTM manual "Generating Applications", REDELIVERY operand in the MAX statement.

- In the KDCMSGTC service and in the KDCSGNTC sign-on service, the DGET call can only be used without specifying a wait time.
- If the main message of a job complex is read, and a positive or negative confirmation job is defined for it (only possible in the case of TAC queues),
  - the positive confirmation job is started once the transaction containing the DGET call has been successfully terminated,
  - the negative confirmation job is started once the transaction containing the DGET call has been reset without delivery of the main message again, i.e. there is no redelivery.

- If DGET is used to read from a queue for which there is data access control, the user ID under which the service is running and the LTERM partner must have access authorization (KSET) for the corresponding queue. Users are however always allowed to access their own USER queue. In the case of applications without explicitly generated user IDs, no data access control can be assigned for the queues assigned to the USERS.
- Saving faulty messages into the dead letter queue

If an error occurs, messages from TAC queues can be saved into the global dead letter queue as a last fallback stage. The queue must be generated with DEAD-LETTER-Q=YES for this. A processed message is then placed into the dead letter queue when the transaction is reset, if it cannot be redelivered (see redelivery) and if no negative acknowledgement job was defined.

When a message is saved into the dead letter queue, the number of redeliveries for this message is reset to zero if necessary.

## DPUT Generate time-driven asynchronous messages

The DPUT (delayed free message PUT) call enables you:

- to send messages to USER queues, TAC queues or temporary queues. The messages to TAC queues can also be time-driven.
- to enter time-driven asynchronous jobs with their messages or message segments in a message queue (output jobs to LTERM partner, background jobs to local asynchronous services, background jobs to remote asynchronous services which were previously addressed using an APRO AM call)
- to log user information concerning these jobs
- to create confirmation jobs within a job complex plus the associated user information
- B** – to pass print options (= RSO parameter list) for RSO printers

You enter the desired point in time in the KDCS parameter area (see table on [page 246](#)). At the given time, the job is entered in the queue of jobs to be executed. In other words, the time-driven job is not executed exactly at the specified time, but at this time at the earliest.

The message segments generated using DPUT are collected by openUTM and are terminated at the next PEND call. Following the end of a transaction, the message segments are sent as a **single** message to the appropriate queue. Exception: If formatted message segments are sent to terminals, each of these segments forms a separate message.

Because of the various functions, the DPUT call has two formats:

- DPUT call without job complex, and
- DPUT call within a job complex.

The format of the DPUT call is described in detail below. For further information on message queuing refer to [section “Message Queuing \(asynchronous processing\)” on page 50ff.](#)

## DPUT call without job complex

### Setting the KDCS parameter area (1st parameter)

The following two tables show the various options and entries in the KDCS parameter area.

	Function of the call	Entries in the KDCS parameter area					
		KCOP	KCOM	KCLM	KCRN	KCMF/ kcfn	KCDF
	Output job in format mode	"DPUT"	"NT"/ "NE"	Length	LTERM name	Format identifier	Screen function
X/W X/W	Output job in line mode	"DPUT"	"NT"/ "NE"	Length	LTERM name	Blanks	—
B B B	Output job in line mode	"DPUT"	"NT"/ "NE"	Length	LTERM name	Blanks/ edit profile	Screen function/ binary zero
	Background job for asynchronous program in the same application	"DPUT"	"NT"/ "NE"	Length	TAC	—	—
	Message for service-controlled queue	"DPUT"	"QT"/ "QE"	Length	Name of the queue	—	—
	Job for transport system application	"DPUT"	"NT"/ "NE"	Length	LTERM name of application	Blanks	Binary zero
	Background job for job-receiving service via LU6.1	"DPUT"	"NT"/ "NE"	Length	service -ID	—	—
	Background job for job-receiving service via OSI TP	"DPUT"	"NT"/ "NE"	Length	service-ID	Blanks/name of abstract syntax	—
	Log user information	"DPUT"	"NI"/ "QI"	Length	as for applicable DPUT NT/NE DPUT QT/QE	Blanks	Binary zero
B B	Pass parameter list for RSO printers	"DPUT"	"RP"	Length	LTERM name	Blank	Binary zero

NT, QT      Message segment for job

NE, QE      Last message segment or entire message for job

NI, QI      User information for a subsequent message

B      RP      RSO parameters

B  
~  
B  
B

Entry in KCOM field	Additional entries in the KDCS parameter area (KCPUT/kc_dput)						
	KCMOD	KCTAG/ kcdays	KCSTD/ kchour	KCMIN	KCSEK/ kcsec	KCQTYP	Other fields
"NT"/"NE"	"A"	001 - 365/366	00 - 23	00 - 59	00 - 59	—	—
	"R"	000 - 364/365				—	
	Blanks	—	—		—		
"QT"/"QE"	"A"	001 - 365/366	00 - 23	00 - 59	00 - 59	Binary zero	Binary zero
	"R"	000 - 364/365				Binary zero	
	Blanks	Binary zero	Binary zero		"U"/"Q"/ binary zero		
"NI"	"A"	001 - 365/366	00 - 23	00 - 59	00 - 59	Binary zero	Binary zero
	"R"	000 - 364/365				Binary zero	
	Blanks	Binary zero	Binary zero		"U"/"Q"/ binary zero		
"QI"	"A"	001 - 365/366	00 - 23	00 - 59	00 - 59	Binary zero	Binary zero
	"R"	000 - 364/365				Binary zero	
	Blanks	Binary zero	Binary zero		"U"/"Q"/ binary zero		
"RP"	"A"	001 - 365/366	00 - 23	00 - 59	00 - 59	Binary zero	Binary zero
	"R"	000 - 364/365				Binary zero	
	Blank	Binary zero					

- A absolute time
- R relative time (= time interval)

In the case of DPUT NT/NE or DPUT QT/QE you have to specify the same times as for the associated DPUT NI or DPUT QI.

**Setting the 2nd parameter**

Here you have to supply the address of the message area from which openUTM is to read the message or user information or the RSO parameter list.

**Setting the parameters**

B  
B

	Field name in the KDCS parameter area	Contents
1.	KCOP	"DPUT"
2.	KCOM	"NT"/"NE"/"QT"/"QE"/"NI"/"QI"/"RP"
3.	KCLM	Length in bytes
4.	KCRN	LTERM name/TAC/service ID/ queue name
5.	KCMF/kcfn	Format ID/blanks/ Name of abstract syntax/ Additionally available in BS200/OSD: edit profile
6.	KCDF	Screen function/binary zero
7.	KCMOD	"R"/"A"/_
8.	KCTAG/kcday	Day (rel./abs.)/binary zero
8.	KCSTD/kchour	Hour (rel./abs.)/binary zero
8.	KCMIN	Minute (rel./abs.)/binary zero
8.	KCSEK/kcsec	Second (rel./abs.)/binary zero
9.	KCQTYP	"U"/"Q"/binary zero
	Message area	
10.		Data

**KDCS call**

	1st parameter	2nd parameter
11.	KDCS parameter area	Message area

12. **C/C++ macro calls**

Macro name	Parameters
KDCS_DPUTNT / KDCS_DPUTNE	(nb,kclm,kcrn,kcfn,kcdf,kcmod,kcday, kchour,kcmin,kcsec)
KDCS_DPUTQT / KDCS_DPUTQE	(nb,kclm,kcrn,kcfn,kcdf,kcmod,kcday, kchour,kcmin,kcsec,kcqtyp)

12. **C/C++ macro calls**

Macro name	Parameters
KDCS_DPUTNI	(nb,kclm,kcrn,kcmod,kcday,kchour,kcmin,kcsec)
KDCS_DPUTQI	(nb,kclm,kcrn,kcmod,kcday,kchour,kcmin,kcsec,kcqtyp)
KDCS_DPUTRP	(nb,kclm,kcrn,kcmod,kcday,kchour,kcmin,kcsec,)

**openUTM return information**

	Field name in KB return area	Contents
13.	KCRCCC	Return code
14.	KCRCDC	Internal return code

For the DPUT call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, enter the DPUT operation code.
2. In the **KCOM** field, enter the required operation modifier:
  - NT for the message segment of the job
  - NE for the total message or last message segment of the job
  - NI for the user information associated with the job.
  - QT for the message segment for a service-controlled queue
  - QE for the entire message or last message segment for a service-controlled queue
  - QI for the user information associated with the message for a service-controlled queue
- B** 3. In the **KCLM** field, specify the length of the message to be sent (length zero is permissible).
  - B** For KCOM = RP, this is the length of the RSO parameter list.



4. In the **KCRN** field, enter the destination of the message:
- the name of the LTERM partner if this DPUT call generates an output job or passes an RSO parameter list
  - the name of the USER queue, TAC queue or temporary queue, if this DPUT call creates a message to a service-controlled queue (KCOM=QT/QE/QI).
  - the transaction code of an asynchronous program if this DPUT call generates a background job (without distributed processing)
  - the service ID of a job-receiving service if this background job is directed to a job-receiving service.

5. In the **KCMF/kcfn** field:

B – A format identifier (in format mode)

B If a formatted message is sent to an RSO printer, then the FHS formatting system  
B must support the printer type as it was generated in RSO, otherwise a formatting  
B error will occur when the message is sent.

B Exception:

B As of IFG V8.1B, formats can be created specifically for RSO printers. FHS as of  
B V8.2C then creates only a single logical message. RSO converts this logical  
B message to the physical message, which is why FHS no longer needs to know the  
B printer type.

- Blanks in line mode or for a job sent to another application without distributed processing.

B – When passing an RSO parameter list.

B – Edit profile (for line mode or an RSO printer)

B If the message is to be sent to an RSO printer, then only the CCSNAME parameter  
B of an edit profile is evaluated. The name of the character set is passed to RSO. All  
B other parameters of the edit profile are ignored because these options are VTSU-B  
B edit options, and the message is being prepared by RSO.

- In the case of messages to OSI TP partners:

The name of the abstract syntax of the message. Space characters stand for the abstract syntax of UDT; in this case, BER is used as the transfer syntax and the message is encoded by openUTM. If you enter a value other than blanks, the message must be transferred to openUTM in encoded format, i.e. in the transfer syntax corresponding to this abstract syntax.

In the case of messages to an asynchronous service of the same application, to USER, TAC or temporary queues or to an LU6.1 partner, this field is irrelevant.

6. In the **KCDF** field, enter the screen function for output jobs to terminals. Enter binary zero for user information (KCOM = NI/QI), RSO parameter lists or jobs for transport system applications.

In the case of background jobs, messages to LU6.1 partners and messages to USER, TAC and temporary queues, this field is irrelevant.

B  
B

You also have to specify binary zero if an edit profile or a #format is entered in KCMF/kcfn.

7. In the **KCMOD** field, select the type of time entry:
- A for absolute
  - R for relative
  - blanks, if the job is to be executed without a wait time.  
messages to USER and temporary queues cannot be sent on a time-driven basis, which is why blanks must be specified in KCMOD
8. Here you enter the necessary time specifications for the call, absolute or relative depending on the entry in KCMOD:
- for absolute time entry: desired time with day of the year in **KCTAG/kcday** (working day), hour in **KCSTD/kchour**, minute in **KCMIN** and second in **KCSEK/kcsec**.
  - for relative time entry: interval to desired execution time with number of days in **KCTAG/kcday**, number of hours in **KCSTD/kchour**, number of minutes in **KCMIN** and the number of seconds in **KCSEK/kcsec**.
  - for KCMOD = blanks:  
binary zero if user information is to be logged with KCOM = NI/QI or when a message is to be sent to a USER or temporary queue (KCOM= QE/QT) (otherwise irrelevant)
9. In the **KCQTY** field, specify in the case of messages to a queue the type of the queue (only in conjunction with KCOM=QT/QE/QI):
- Q for a temporary queue created with QCRE
  - U for a queue assigned to a user ID (USER queue)
  - binary zero in all other cases

You enter in the message area:

10. The message or user information you want to output or the RSO parameter list you want to pass.

You enter the following for the KDCS call:

11. 1st parameter: the address of the KDCS parameter area

2nd parameter: the address of the message area from which UTM is to read the message or user information or RSO parameter list. You enter the address of the message area even if you have entered the length 0 in KCLM.

12. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

13. In the **KCRCCC** field, the KDCS return code.

14. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### KDCS return codes in the KCRCCC field

The following codes can be analyzed in the program:

000 Function carried out

06Z Time entry changed without preceding DPUT NE, i.e. at least one of the fields KCMOD, KCTAG/kcday, KCSTD/kchour, KCMIN or KCSEK/kcsec has a value differing from that in the first message segment (for KCMOD=A/R). openUTM takes the time entry from the first DPUT call and continues the message.

40Z openUTM cannot perform the function, see entry in KCRCDC.

Possible causes:

- KCDF does not contain binary zero although this is required in this particular situation
- in the case of jobs without distributed processing, the name changes in KCRN and the type changes in KCQTYP, without the preceding DPUT job being terminated.
- in the case of distributed processing: there is no logical connection to the partner application and KCMOD = “\_”.

41Z The call is not allowed at this location:

- the call was initiated in the first part of the sign-on service or
- the call was initiated in the sign-on service after a SIGNON call and before the PEND PS call.

42Z The entry in KCOM is invalid.

43Z The length entry in KCLM is negative or invalid.

- 44Z The value in KCRN or KCQTYP is invalid. Possible causes:
- the value is neither the transaction code of an asynchronous program nor the name of a LTERM partner and it is also not a valid service ID.
  - although the value is the transaction code of an asynchronous programs, the transaction code is locked or access protected.
  - KCQTYP=U: the value in KCRN does not specify a user or a user with access authorization.
  - KCQTYP=Q: the value in KCRN does not specify a temporary queue.
  - No queued messages can be generated for the dead letter queue (KDCDLETQ).
  - for KCOM = RP: the value is not an RSO printer or the current RSO version does not support this function.

B  
B

No asynchronous messages are allowed for the dead letter queue (KDCDLETQ).

- 45Z The entry in KCMF/kcfn is invalid.  
Possible causes:
- the format identifier in KCMF/kcfn is not valid
  - if the message destination is a partner to which communication is established using the OSI TP protocol, this return code indicates that the abstract syntax in the KCMF/kcfn field has not been generated for the partner application.
  - for KCOM = RP: no blank entered
  - the edit profile has not been generated
  - the edit profile changes in message segments

B  
B  
B

- 47Z The address of the message area is invalid.
- 49Z The contents of unused fields of the KDCS parameter area are not equal to binary zero.
- 51Z After a DPUT NI/QI there is no DPUT NT/NE/QT/QE to the same destination.
- 52Z An attempt was made to send a time-driven message to a USER queue or temporary queue.
- 56Z The entry in KCMOD is invalid or the time entry in KCTAG/kcday, KCSTD/kchour, KCMIN or KCSEK/kcsec is invalid or is outside the generated time span.

An additional return code can be found in the dump:

- 71Z INIT was not issued in this program.

### Features of the DPUT call

- The message area is not changed when the call is executed by UTM.
- More than one job can be generated in one program unit; the corresponding messages can each comprise more than one segment.
- You can also use screen functions when you output +formats, \*formats or messages in line mode, see the section “Screen output functions” on [page 107](#).  
If you use #formats, you have to specify binary zero for KCDF, otherwise openUTM returns 40Z.  
If you use edit profiles, (BS2000/OSD) openUTM also returns 40Z if you have not specified binary zero for KCDF.
- Jobs generated with DPUT are discarded as a result of PEND ER/FR, PEND RS or RSET.
- The TAC must be located at the beginning of the message area for DPUT calls to another UTM application that is generated as a transport system application.
- The message is also formatted before it is output.
- Messages are kept until:
  - the referenced program unit or the printout is terminated, or
  - the transfer has succeeded for jobs to remote asynchronous services, or
  - the message is read at the terminal using KDCOUT and a new input has been made (except for the KDCLAST command).
  - the message to a queue is read by means of a DGET call, and the corresponding transaction is successfully completed.
- Jobs with messages of length 0  
If you generate a message with a length of 0 (so-called "dummy message"), the following applies:
  - a background job is executed, i.e. the asynchronous service is started without receiving a message
  - an empty format is output if the job is an output job in format mode
  - an output job for a transport system application is accepted, but will later be discarded by openUTM.

B  
B

- Output jobs that are for one specific terminal are placed in the terminal message queue and can be read by the user with the KDCOUT command. Exactly one message is read per KDCOUT command. Each message can only be read once. If the KDCOUT command is entered more than once, then the next message is read from the terminal queue.

The user is informed at the end of a transaction that there are asynchronous messages available for a terminal through a message in the system status line.

B This announcement can be suppressed in BS2000/OSD if ANNOAMSG=N was  
 B specified in the configuration for the affected LTERM partner (default value:  
 B ANNOAMSG=Y). Asynchronous messages are then displayed immediately on the  
 B screen. This can disturb the dialog flow. The terminal user can, however, re-display the  
 B last screen using the KDCDISP command.

- There is no interaction between FPUT and DPUT calls, i.e. you can send DPUT calls with KCMOD = "\_" and FPUT calls to the same destination independently of each other.

B ● Print options for RSO printers

B If you use print options for jobs to RSO printers, you should first pass the list with the  
 B print options using DPUT RP, see RSO manual. Then submit the actual print job using  
 B DPUT NT/NE. The time specifications in DPUT RP and DPUT NT/NE must match.

- Handling message segments
  - message segments in line mode are combined and output as **one** message to the LTERM partner. The message segments generated with DPUT are gathered by openUTM and terminated by the next PEND call, provided they have not yet been terminated by the program unit run with DPUT NE. When the transaction ends, the message segments are sent as a single message to the LTERM partner or to the other application.
  - with formatted message segments to terminals, each segment generates a new format. The format name in KCMF may change. At a terminal, each format (each message segment) must be fetched with a KDCOUT command. Each DPUT NT call generates its own message. It is therefore not possible to structure the screen with different partial formats using DPUT NT calls. The formats arrive in the order in which they were sent.
  - with message segments to printer, it is possible to switch between formatted message segments and unformatted message segments (in line mode). With message segments to terminals, this switch causes the old message to terminate and a new one to start.
  - with DPUT NT calls to a local or asynchronous service each message segment must be read with a separate FGET.

- with DPUT QT calls to a service-controlled queue, each message segment must be read by means of a separate DGET.
- with PEND, the message segment most recently generated with DPUT is always assumed to be the final message segment, even if it was output with NT.
- In a sequence of message segments, a change of the destination specified in KCRN with no preceding DPUT NE/QE is only permissible in certain cases (see next point).
- if the edit profile changes within a sequence of message segments addressed to a terminal, openUTM reacts with 45Z.

B  
B

- The maximum possible number of DPUT NT/QE calls in a transaction depends on the RECBUF generation parameter in the KDCDEF statement MAX. 30 bytes per DPUT NE are occupied in this buffer. If the buffer is full, DPUT NE is rejected with KCRDC=K704.

- Parallel messages

Parallel messages (i.e. change of destination before DPUT NE/NQ) are permissible if the destinations belong to different categories. There are three categories:

- LTERM partners, local asynchronous programs and service-controlled queues (KCRN=LTERM/TAC/queue name)
- job complexes (KCRN = complex ID)
- remote asynchronous services (KCRN = service ID) or remote TAC queues

Within these categories, parallel asynchronous jobs are only permitted if they are jobs for remote asynchronous services.

Apart from that, parallel job complexes are not supported, and the change of the LTERM/TAC/queue name requires the message to be concluded by means of DPUT NE/QE.

- Influence of generation parameters on the DPUT call

The following notes concern the generation of the UTM application. Further information on the individual generation parameters can be found in the openUTM manual "Generating Applications".

Limits for the time entry in the DPUT call are defined with the operands DPUTLIMIT1 and DPUTLIMIT2 in the MAX statement. The interval between the desired execution time and the DPUT call must not be greater than the time entry in DPUTLIMIT2 if **before** the time of the DPUT call, nor greater than the time entry in DPUTLIMIT1 if **after** the time of the DPUT call:

Current time - DPUTLIMIT2 < execution time < current time + DPUTLIMIT1

The time of the DPUT call is taken as the current time.

In the case of DPUT calls with KCMOD = A or R to LTERM partners generated with LTERM ..., QAMSG=N, openUTM does not check at the time of the DPUT call to see whether a client or printer is connected to the LTERM partner. It only does this once the time specified in the DPUT call has arrived. If there is then no connection, openUTM will store the message until a connection is set up.

In the case of DPUT calls with KCMOD = "\_" to LTERM partners generated with LTERM ..., QAMSG=N, openUTM checks at the time of the DPUT call whether a client/printer is connected to this LTERM partner. If not, UTM rejects the call with KCRCCC = 44Z, KCRCDC= K705.

For all messages to terminals and transport system applications generated with DPUT the following holds: the entire message must not exceed the maximum length defined at generation time with the TRMSGLTH operand of the MAX control statement.

When the KDCFILE is regenerated with the UTM tool KDCUPD, you can specify in the TRANSFER statement precisely which messages you wish to transfer to the new KDCFILE (see also openUTM manual "Generating Applications").

- The UTM administrator can use the administration call KDCINF STAT to query the number of waiting time-driven jobs (see the openUTM manual "Administering Applications", Administration commands, KDCINF).
- DPUT calls with distributed processing
  - Prior to the DPUT call, the KDCS parameter area in the job-submitting service must be given the same values as for sending messages to a TAC queue or for generating background jobs to an asynchronous program in its own application. The only thing that has to be specified as name in the KCRN field is the service ID which was assigned to the job-receiving service in the APRO call.
  - Once the DPUT NE/QE call has terminated (final message segment or complete message) or after the KDCS return code 40Z, the service ID in the job-submitting service is released. This ID can now be used to address another job-receiving service.
  - Only after the specified time has elapsed will the job generated with DPUT be sent to the partner application (if a free session or association is available).
  - Parallel asynchronous jobs to different job-receiving services are permissible.



- User information (DPUT NI/QI)

User information is always associated with a job generated by a DPUT call. The user information must be generated before the job itself. The addressee (entry in KCRN) and the time entry in the user information and in the job must agree. If this is not sequence is not adhered to, error 51Z occurs. If the associated job for an item of user information does not exist (i.e. if there is user information but no job), openUTM aborts the service at the PEND call with KCRCCC = 86Z.

User information can only be read with a DADM call; it is a type of "log information" and is not transferred to the addressee. The user information of a confirmation job can not be read until the confirmation job has been activated.

- Background jobs for an asynchronous program in the same application

Each background job starts an asynchronous service at the time specified.

Asynchronous programs which run in time-driven mode should check whether their work is still relevant, or whether they should terminate immediately. The program can ascertain the current time and date, as well as the time and date of application start-up, via the INFO call.

If the program needs the time of the DPUT call, it must be contained in the message.

DPUT calls can also be used to implement periodically recurring asynchronous jobs. This is done with an asynchronous program containing the periodically recurring action as well as a DPUT call to the program itself. The time can be specified as relative or absolute.

## DPUT call in a job complex

A DPUT call within a job complex enables you to

- send time-driven messages (so-called basic jobs) and the associated messages/message segments (output jobs for LTERM partners or for asynchronous services, messages to TAC queues or background jobs to asynchronous services or TAC queues previously addressed with an APRO AM call)
- have user information associated with these jobs logged
- generate confirmation jobs and associated user information.

### Setting the KDCS parameter area (1st parameter)

The table below shows the various options and corresponding entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area					
	KCOP	KCOM	KCLM	KCRN	KCMF	KCDF
Basic job for LTERM partner or for local asynchronous service or local TAC queue	"DPUT"	"NT"/ "NE"	Length	Complex ID	Format ID / blanks/ name of abstract syntax  (also possible in BS2000/OSD: edit profile	Screen function/ binary zero
Basic job for TAC queue	"DPUT"	"QT"/ "QE"	Length	Complex ID	—	
Basic job for remote asynchronous service or remote TAC queue	"DPUT"	"NT"/ "NE"	Length	Complex ID	—	
User information for a job	"DPUT"	"NI"/ "QI"	Length	Complex ID	Blanks	binary zero
Create confirmation job	"DPUT"	"+T"/ "-T"	Length	Complex ID	Blanks	binary zero
User information for confirmation job	"DPUT"	"+I"/ "-I"	Length	Complex ID	Blanks	binary zero

B  
B  
B

The operation modifiers in the KCOP field have the following significance:

NT/QT message segment of a basic job

NE/QE last message segment or total message of a basic job

NI/QI user information associated with basic job

+T/-T positive or negative confirmation job

+I/-I user information associated with positive or negative confirmation job

The QE/QT/QI operation modifiers refer to a basic job with a TAC queue as the destination. It is not possible to specify a USER or temporary queue as the destination in the DPUT call in the job complex, because KCQTYP is not evaluated here and the destination is specified for the associated MCOM call.

The time entries for DPUT NT/NE/NI or DPUT QT/QE/QI are made as for the DPUT call without job complex. For DPUT NI/QI/+T/-T/+I/-I you have to set binary zero in all the unused fields of the KDCS parameter area.

### **Setting the 2nd parameter**

Here you have to supply the address of the message area from which openUTM is to read the message or user information.

**Setting the parameters**

B  
B

	Field name in KDCS parameter area	Contents
1.	KCOP	"DPUT"
2.	KCOM	"NT" / "NE" / "NI" / "QT" / "QE" / "QI" / "+T" / "-T" / "+I" / "-I"
3.	KCLM	Length in bytes
4.	KCRN	Complex identifier
5.	KCMF/kcfn	Format ID/blanks/ name of abstract syntax/ also possible in BS2000/OSD : edit profile
6.	KCDF	Screen function/binary zero
7.	KCMOD	""R""A""?"/binary zero
8.	KCTAG/kcday	Day (rel./abs.)""?"/binary zero
8.	KCSTD/kchour	Hour (r./a.)""?"/binary zero
8.	KCMIN	Minute (r./a.)""?"/binary zero
8.	KCSEK/kcsec	Second (r./a.)""?"/binary zero
9.	KCQTYP	irrelevant (not evaluated)
10.	Message area	Data

**KDCS call**

11.	1st parameter	2nd parameter
	KDCS parameter area	Message area

## 12. C/C++ macro calls

Macro name	Parameters
KDCS_DPUTNT / KDCS_DPUTNE	(nb,kclm,kcrn,kcfn,kcdf,kcmmod,kcday,kchour,kcmin,kcsec)
KDCS_DPUTQT / KDCS_DPUTQE	(nb,kclm,kcrn,kcfn,kcdf,kcmmod,kcday,kchour,kcmin,kcsec,kcqtyp)
KDCS_DPUTNI	(nb,kclm,kcrn,kcmmod,kcday,kchour,kcmin,kcsec)
KDCS_DPUTQI	(nb,kclm,kcrn,kcmmod,kcday,kchour,kcmin,kcsec,kcqtyp)
KDCS_DPUTPT / KDCS_DPUTMT / KDCS_DPUTPI / KDCS_DPUTMI	(nb,kclm,kcrn)

## openUTM return information

	Field name in the KB return area	Contents
13.	KCRCCC	Return code
14.	KCRCDC	Internal return code

For the DPUT call within a job complex you make the following entries in the KDCS parameter area:

- In the **KCOP** field, enter the DPUT operation code.
- In the **KCOM** field, enter the required operation modifier:
  - NT/QT for message segment of the basic job
  - NE/QE for total message or last message segment of the basic job
  - NI/QI for user information associated with the basic job
  - +T for a positive confirmation job
  - T for a negative confirmation job
  - +I for user information relating to the positive confirmation job
  - I for user information relating to the negative confirmation job

The QE/QT/QI operation modifiers refer to a basic job with a TAC queue as the destination.
- In the **KCLM** field, specify the length of the message to be sent (length zero is permissible).

4. In the **KCRN** field, enter the complex identifier (complex ID) assigned to the job complex (assigned in the MCOM call).
5. In the **KCMF/kcfn** (irrelevant for messages sent to an asynchronous service of the same application or to an LU6.1 partner):

- Blanks in line mode or for a job sent to another application without distributed processing.

B

- A format identifier (in format mode)

B

If a formatted message is sent to an RSO printer, then the FHS formatting system must support the printer type as it was generated in RSO, otherwise a formatting error will occur when the message is sent.

B

B

B

Exception:

As of IFG V8.1B, formats can be created specifically for RSO printers. FHS as of V8.2C then creates only a single logical message. RSO converts this logical message to the physical message, which is why FHS no longer needs to know the printer type.

B

B

B

B

B

- Edit profile (for line mode or an RSO printer)

If the message is to be sent to an RSO printer, then only the CCSNAME parameter of an edit profile is evaluated. The name of the character set is passed to RSO. All other parameters of the edit profile are ignored because these options are VTSU-B edit options, and the message is being prepared by RSO.

B

B

B

B

In messages to OSI TP partners:

- The name of the abstract syntax of the message. Space characters stand for the abstract syntax of UDT; in this case, BER is used as the transfer syntax and the message is encoded by openUTM.

If you enter a value other than blanks, the message must be transferred to openUTM in encoded format, i.e. in the transfer syntax corresponding to this abstract syntax.

6. In the **KCDF** field, enter the screen function for output jobs to terminals. Enter binary zero for user information (KCOM = NI/QI) or jobs for transport system applications.

This field is irrelevant for background jobs and messages to TAC queues.

B

You must also specify binary zero if an edit profile or a #format is entered in KCMF/kcfn.

7. In the **KCMOD** field, you select the type of time entry for messages to the basic job (KCOM=NT/NE/NI or QT/QE/QI):

- A for absolute

- R for relative

- blanks, if the job is to be executed without wait time

Binary zero must be entered for messages to confirmation jobs (KCOM = +T/-T/+I/-I).

8. For KCOM = NT/NE/NI or QT/QE/QI enter the necessary time specifications as for the DPUT call without job complex in these fields. For KCOM = +T/-T/+I/-I enter binary zero.
9. The **KCQTYP** field is not evaluated.

Make the following entry in the message area:

10. the message or user information you want to output.

You enter the following for the KDCS call:

11. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area from which openUTM is to read the message (or user information). You enter the address of the message area even if you have entered the length 0 in KCLM.
12. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

13. In the **KCRCCC** field, the KDCS return code.
14. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

**KDCS return codes in the KCRCCC field**

The following codes can be analyzed in the program:

- 000    Function carried out.
- 06Z    Time entry changed without preceding DPUT NE, i.e. at least one of the fields KCMOD, KCTAG/kcday, KCSTD/kchour, KCMIN or KCSEK/kcsec has a value differing from that in the first message segment (for KCMOD=A/R). openUTM takes the time entry from the first DPUT call and continues the message
- 40Z    openUTM cannot perform the function, see entry in KCRCDC. For distributed processing: there is no logical connection to the partner applic. and KCMOD = " \_".
- 41Z    The call is not allowed at this location:
  - an additional basic job is to be issued or user information is to be logged after a basic job (DPUT NE/QE) has been completed or
  - the format identifier changes due to several DPUT NTs or
  - the call was initiated in the first part of the sign-on service or
  - the call was initiated in the sign-on service after a SIGNON call and before the PEND PS call.
- 42Z    The entry in KCOM is invalid or KCOM = +T/-T was specified without defining a job complex or destination for the confirmation job
- 43Z    The length entry in KCLM is negative or invalid.
- 44Z    The complex identifier given in KCRN is invalid.
- 45Z    The entry in KCMF/kcfn is invalid. Possible causes:
  - the format identifier in KCMF/kcfn is invalid
  - if the message destination is a partner with which communication is performed using the OSI TP protocol, this return code indicates that the abstract syntax specified in the KCMF/kcfn field has not been generated for the partner application.
- B    – the edit profile has not been generated
- B    – the edit profile changes in message segments
- 47Z    The address of the message area is invalid.
- 49Z    The contents of unused fields of the KDCS parameter area are not equal to binary zero.
- 51Z    Incorrect DPUT call sequence (see below).
- 56Z    The entry in KCMOD is invalid or the time entry in KCTAG/kcday, KCSTD/kchour, KCMIN or KCSEK/kcsec is invalid or is outside the generated timespan.

An additional return code can be found in the dump:

- 71Z    INIT call missing in this program.



### Features of the DPUT call within a job complex

- The following rules hold for the order of the DPUT calls:
  - the user information has to be written before the associated (confirmation) job: DPUT NI before DPUT NT/NE or DPUT QI before DPUT QT/QE, DPUT +I before DPUT +T and DPUT -I before DPUT -T. Only one DPUT +I/-I per DPUT +T/-T is permitted. The user information of a confirmation job cannot be read until the confirmation job has been activated.
  - the basic job must be started before the first confirmation job: DPUT NT/NE or DPUT QT/QE before the first DPUT +T/-T. Two or more DPUT +T or DPUT -T are permissible; these are considered in each case as **one** positive confirmation job (with +T) and **one** negative confirmation job (with -T).

- Changing the format identifier if there are more than one message segments (DPUT NT) is not permissible.

- If a DPUT call has return code 40Z all the messages and information associated with the job complex are lost; the complex ID is released.

For DPUT messages via distributed processing the service ID is also released with 40Z.

For all other return codes smaller than 70Z all the data of the complex is kept including the complex ID.

- Parallel asynchronous jobs (i.e. change of destination before DPUT NE/QE) are permitted if the destinations belong to different categories, i.e. it is possible to have a DPUT with KCRN = LTERM/TAC/Queue or KCRN = VGID before the DPUT NE.

*Example for parallel asynchronous jobs*

KDCS call	Destination and identifier	Remarks
MCOM BC	KCRN = PRINTER1 KCCOMID = *COMPLEX KCPOS = ATAC5	Define start of a job complex and destinations of the job
DPUT NT	KCRN = *COMPLEX	1st message segment of basic job
DPUT NE	KCRN = ATAC1	Background job for a program
APRO AM	KCRN = LTAC KCPA = APPL1 KCPI = > VGID	Address job-receiving service
DPUT NE	KCRN = > VGID	Background job for job-receiving service
DPUT NE	KCRN = *COMPLEX	2nd message segment of basic job
DPUT +T	KCRN = *COMPLEX	Positive confirmation job
MCOM EC	KCRN = binary zero KCCOMID = *COMPLEX	End of job complex

## FGET Receive asynchronous message

The FGET (free message GET) call enables you to read an asynchronous message or message segment into the message area from the message queue allocated to the service.

The FGET call is only permitted in the first program unit run and processing step of an asynchronous service. Each message can only be read once. Each message segment must be read with its own FGET. The message (message segment) can be read again following an RSET call.

The format of the FGET call is described in detail below. For further information on message queuing refer to section [2.4 on page 50](#).

### Setting the KDCS parameter area (1st parameter)

The table below shows the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area		
	KCOP	KCLA	KCMF/kcfn
Read asynchronous message	"FGET"	Desired length	Format identifier/ blanks/ name of abstract syntax also possible in BS2000/OSD edit profile

B  
B

### Setting the 2nd parameter

Here you have to supply the address of the message area into which openUTM is to read the message.

**Setting the parameters**

**B**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"FGET"
2.	KCLA	Length in bytes
3.	KCMF/kcfn	Format identifier/blanks/ name of abstract syntax/ edit profile

**KDCS call**

	1st parameter	2nd parameter
4.	KDCS parameter area	Message area

**C/C++ macro calls**

	Macro name	Parameters
5.	KDCS_FGET	(nb,kcla,kcfn)

**openUTM return information**

	Message area	Contents
6.		Data
	Field name in the KB return area	
7.	KCRLM	Actual length
8.	KCRCCC	Return code
9.	KCRCDC	Internal return code
10.	KCRMf/kcrfn	Format identifier/blanks
11.	KCRRC	Redelivery counter

For the FGET call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, enter the FGET operation code.
2. In the **KCOM** field, specify the length in which the message is to be read. This length must not exceed the message area into which the message is to be read. Length zero means no receiving of messages. Any existing messages are lost.
3. In the **KCMF/kcfn** field, enter the format of the message to be read:

- in line mode: blanks

B  
B

or name of the **edit profile** (returned in the KCRMf/kcfn field in the case of INIT). This name starts with a blank.

- in format mode: format identifier of the expected (partial) format. This is returned in the KCRMf/kcfn field in the case of INIT or a preceding FGET call.
- for a message from a program unit of the same application or from an LU6.1 partner: irrelevant.

in messages to an OSI TP partner:

- name of the abstract syntax of the message.  
This name was returned in the KCRMf/kcfn field in the preceding INIT call. Here, blanks represent the abstract UDT syntax encoded in accordance with BER (Basic Encoding Rules); only in this case does openUTM transfer the message to the program unit decoded.  
If you enter a value other than blanks, openUTM transfers the message to the program unit in encoded format (i.e. in the transfer syntax corresponding to this abstract syntax) and the program unit itself must convert the message into the local representation. This is possible, for example, using an ASN.1 compiler.

You specify the following for the KDCS call:

4. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area into which openUTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLM.
5. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497ff.](#)

openUTM returns:

6. in the specified message area, the message (segment) in its actual or at most in its desired length.
  7. in the **KCRLM** field, the actual length of the message (segment), possibly deviating from the length requested in the KCLA of the parameter area.
  8. in the **KCRCCC** field the KDCS return code (see next page).
  9. In the **KCRDC** field, the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").
  10. In the **KCRMF/kcrfn** field:
    - After reading an entire format: ID of most recently read format (always identical to ID of last output format).
    - After reading a partial format: Identifier of the next format with input data.
    - After reading final partial format: ID of most recently read partial format. In this case KCRMF = KCMF (or: kcrfn=kcfn).
    - After reading in line mode: blanks
- B** or name of the edit profile of the last output message.
- After reading a partner service:  
name of the format identifier or abstract syntax of the next message (segment).
11. in the **KCRRC** field, the redelivery counter of the message read. This contains the number of redeliveries of the FGET message after abnormal termination of the asynchronous services in the first transaction.

Sender information, etc. is located in the KB header (entered by openUTM during INIT).

### KDCS error codes for the FGET call

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 01Z Length conflict:  $KCLA < KCRLM$ ; message was truncated.
- 03Z With partial formats:  $KCMF/kcfn$  does not contain the name of the next returned partial format; the message area is unchanged and  $KCRLM = 0$  (see also 5. and 6.).  
Entry message from a OSI TP partner:  
 $KCMF/kcfn$  does not contain the abstract syntax of the next message to be read.  
No message is transferred to the message area.
- 05Z With individual formats: format in screen display different from format entered in  $KCMF/kcfn$ .  
In line mode: edit profile in screen display different from edit profile entered in  $KCMF/kcfn$ .
- 10Z Message or all message segments already completely read.

Additional error codes to be taken from the dump:

- 70Z System could not perform the operation (system or generation error).
- 71Z Function called in a follow-up program unit or following a PGWT call of an asynchronous service or in a dialog service or still no INIT issued in program unit run.
- 73Z Length entry in  $KCLA$  negative or invalid.
- 77Z The message area is missing or cannot be accessed in the specified length.

### Features of the FGET call

- The actual message length is returned in the  $KCRLM$  field. The following holds:
  - When  $KCRLM \leq KCLA$ , only  $KCRLM$  characters (bytes) are transferred into the message area. The contents of the remainder of the message area are undefined.
  - When  $KCRLM > KCLA$ , only  $KCLA$  characters are transferred into the message area. The remaining bytes ( $KCRLM - KCLA$ ) are lost. They can no longer be read with a subsequent FGET.

The description of the MGET call presents an example explaining how openUTM reacts in case of length conflicts

- A program unit can also receive asynchronous messages of length 0 when, for example,
  - a function key was pressed, but no message was allocated
  - a transaction code was sent without further data
  - if a program in the same application sent a background job with a message of length 0.
- TAC input from the terminal or a transport system application:
  - When an asynchronous TAC is entered in line mode and a format ID is entered in KCMF/kcfn, return code KCRCCC=05Z is not set (as with MGET), but rather KCRCCC=000, assuming the call is otherwise error-free.
  - When partial formats are input, each partial format must be read using a separate FGET.
  - If a message is input together with an asynchronous TAC, the TAC is separated from the message: the TAC is not read into the message area, but is available in the KB header after INIT.
  - openUTM does not convert from lowercase to uppercase letters.

**B**

However, conversion is possible using edit profiles.

- The number of redeliveries is returned in the KCRRC field. A message is always redelivered if an asynchronous service was terminated abnormally in the first transaction. The application must have been generated accordingly and the generated maximum number of redeliveries must not have been reached. For more information refer to the openUTM manual "Generating Applications", REDELIVERY operand in the MAX statement.
- Saving faulty messages into the dead letter queue:

If an error occurs, asynchronous messages to transaction codes can be saved into the global dead letter queue as a last fallback stage. The TAC must be generated with DEAD-LETTER-Q=YES for this. If the asynchronous process terminates abnormally without successful termination of a transaction, the FGET message is placed in the dead letter queue if it cannot be redelivered (see redelivery) and no negative acknowledgement job was defined. As soon as an asynchronous process has reached a save point, neither redelivery nor saving the FGET message into the dead letter queue is possible since the message is taken to have been successfully processed.

When a message is saved into the dead letter queue, the number of redeliveries for this message is reset to zero if necessary.



Each message segment sent with FPUT NT must be read using a separate FGET.

## FPUT Generating asynchronous messages

The FPUT (free message PUT) call enables you to create messages or message segments for message queues, which openUTM enters in receiver-specific queue:

- output jobs for LTERM partners
- background jobs for local asynchronous services
- background jobs for remote asynchronous services previously addressed with APRO AM calls
- asynchronous messages for local or remote TAC queues
- B** – pass print options (= RSO parameter list) for RSO printers

The message segments generated with FPUT are gathered by openUTM and terminated with the next PEND call. When the transaction ends the message segments are input as a single message into the appropriate message queue. Exception: If you send formatted message segments to terminals, each of these segments forms a separate message.

In the case of TAC queues, the recipient must read the message from the queue in a separate transaction.

Messages remain in a message queue until they are:

- successfully sent (LTERM partner)
- successfully processed (asynchronous service) or
- successfully read (TAC queue).

The format of the FPUT call is described in detail below. For further information on message queuing refer to [section “Message Queuing \(asynchronous processing\)” on page 50ff.](#)



## Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area

	Function of the call	Entries in the KDCS parameter area					
		KCOP	KCOM	KCLM	KCRN	KCMF/kcfn	KCDF
	Output job in format mode	"FPUT"	"NT"/ "NE"	Length	LTERM name	Format identifier	Screen function
X/W X/W	Output job in line mode	"FPUT"	"NT"/ "NE"	Length	LTERM name	Blanks	—
B B B	Output job in line mode	"FPUT"	"NT"/ "NE"	Length	LTERM name	Blanks/ edit profile	Screen function/ binary zero
	Message for asynchr. program or TAC queue of the same application	"FPUT"	"NT"/ "NE"	Length	TAC/ Name of a TAC queue	—	—
	Output job for transport system application	"FPUT"	"NT"/ "NE"	Length	LTERM name of application	Blanks	Binary zero
	Background job via LU6.1	"FPUT"	"NT"/ "NE"	Length	service ID	—	—
	Background job via OSITP	"FPUT"	"NT"/ "NE"	Length	service ID	Name of abstract syntax/blanks	—
B B	Pass parameter list for RSO printers	"FPUT"	"RP"	Length	LTERM name	Blank	Binary zero

NT: message segment of the job

NE: last message segment or entire message of the job

B RP: RSO parameter list

## Setting the 2nd parameter

Here you have to supply the address of the message area from which openUTM is to read the message or the RSO parameter list.

**Setting the parameters**

B  
B

	Field name in the KDCS parameter area	Contents
1.	KCOP	"FPUT"
2.	KCOM	"NT"/"NE"/"RP"
3.	KCLM	Length in bytes
4.	KCRN	LTERM name/TAC queue/service ID
5.	KCMF/kcfn	Format identifier/blanks/ Also available in BS2000/OSD edit profile
6.	KCDF	Screen function/binary zero
	Message area	
7.		Data

**KDCS call**

	1st parameter	2nd parameter
8.	KDCS parameter area	Message area

**C/C++ macro call**

	Macro name	Parameters
	KDCS_FPUTNT / KDCS_FPUTNE	(nb,kclm,kcrn,kcfn,kcdf)
	KDCS_FPUTRP	(nb,kclm,kcrn)

**openUTM return information**

	Field name in the KB return area	Contents
10.	KCRCCC	Return code
11.	KCRCDC	Internal return code

For the FPUT call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, enter the FPUT operation code.
2. In the **KCOM** field, enter either NT or NE for total message or last message segment or RP for an RSO parameter list.
3. In the **KCLM** field, specify the length of the message to be sent in the message area (length zero is permissible).

**B**

For KCOM = RP, this is the length of the data structure for the RSO parameter list.

4. In the **KCRN** field, enter the destination of the message:
  - the name of the LTERM partner if this FPUT call generates an output job or passes an RSO parameter list
  - the transaction code of an asynchronous program if this FPUT generates a background job (without distributed processing)
  - the service ID of a job-receiving service if this background job is directed to a job-receiving service
  - the name of the TAC queue if the message is to be sent to a TAC queue
  - the service ID of the remote TAC queue if this message is to be sent to a remote TAC queue
5. In the **KCMF/kcfn** field (If messages are sent to an asynchronous service or a TAC queue of the same application or to an LU6.1 partner then this field is irrelevant), specify:
  - Blanks in line mode or for a job sent to another application without distributed processing or when passing an RSO parameter list.

**B**

- A format identifier (in format mode)

**B**

**B**

**B**

If a formatted message is sent to an RSO printer, then the FHS formatting system must support the printer type as it was generated in RSO, otherwise a formatting error will occur when the message is sent.

**B**

Exception:

**B**

**B**

**B**

**B**

As of IFG V8.1B, formats can be created specifically for RSO printers. FHS as of V8.2C then creates only a single logical message. RSO converts this logical message to the physical message, which is why FHS no longer needs to know the printer type.

- B – Edit profile (for line mode or an RSO printer)
- B If the message is to be sent to an RSO printer, then only the CCSNAME parameter of an edit profile is evaluated. The name of the character set is passed to RSO. All other parameters of the edit profile are ignored because these options are VTSU-B edit options, and the message is being prepared by RSO.
- B
- B

In messages to an OSI TP partner:

- The name of the abstract syntax of the message. Here, blanks represent the abstract syntax of UTD. In this case the BER transfer syntax is used and the message is encoded openUTM.  
If you enter a value other than blanks, the message must be transferred to openUTM in encoded form, i.e. in the transfer syntax corresponding to this abstract syntax.
6. In the **KCDF** field (irrelevant for background jobs and TAC queues):  
the screen function if the FPUT call is intended for an LTERM partner. Enter binary zero for jobs to other applications without distributed processing or when passing RSO parameter lists.

- B You must also specify binary zero if an edit profile or a #format is specified in KCMF/kcfn.
- B

In the message area you specify:

7. the message you want to output or the RSO parameter list you want to pass.

You enter the following for the KDCS call:

8. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area from which openUTM is to read the message or RSO parameter list. You enter the address of the message area even if you have entered the length 0 in KCLM.
9. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

10. In the **KCRCCC** field, the KDCS return code (see next page).
11. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### KDCS return codes in the KCRCCC field

The following codes can be analyzed in the program:

- 000 Function carried out
- 04Z The name in KCRN changes, and FPUT NE was not specified before.
- 40Z openUTM cannot perform the function (system or generation error, deadlock, long-term locks), see KCRCDC.
- 41Z KCRN addressed the LTERM partner that started the current service or FPUT was issued in the first part of the sign-on service or an FPUT call was issued in the sign-on service after the SIGN ON and before the PEND PS.
- 42Z Entry in KCOM invalid.
- 43Z Length entry in KCLM negative or invalid.
- 44Z Value in KCRN is not a TAC of an asynchronous program or a TAC queue, or the TAC is locked or prohibited and there is no name of an LTERM partner, or there is no valid service ID (with distributed processing). See KCRCDC.

No asynchronous messages are allowed for the dead letter queue (KDCDLETQ).

B  
B

For KCOM = RP: the value in KCRN is not an RSO printer or the current version does not support this function.

- 45Z The entry in KCMF/kcfn is invalid. Possible causes:
  - the format identifier in KCMF/kcfn is invalid
  - if the message destination is a partner with which communication is performed using the OSI TP protocol, this return code indicates that the abstract syntax in the KCMF/kcfn field has not been generated for the partner application.
  - for KCOM = RP: no blank entered
  - the edit profile has not been generated
  - the edit profile changes in message segments to terminals

B  
B  
B

- 47Z The message area is missing or not accessible in the specified length.

An additional error code can be found in the dump:

- 71Z INIT call missing in this program.

### Features of the FPUT/DPUT call

- FPUT and DPUT calls which direct program units to an alias LTERM are processed as follows:
  - In an LTERM group without an LTERM bundle, openUTM sends FPUT/DPUT calls via the PTERM which is assigned to the primary LTERM.
  - In the case of an LTERM group whose primary LTERM is the master LTERM of an LTERM bundle, openUTM assigns all the queued messages sent to the group's alias LTERMs during this transaction to one of the slave LTERMs on transaction end. This procedure guarantees that the receiver possesses the same message sequence as was generated for an LTERM group during a transaction.
- FPUT and DPUT calls which direct program units to the master LTERM are assigned to one of the slave LTERMs at transaction end.
- FPUT and DPUT calls can also direct program units directly to the primary LTERM.
- The message area is not changed when openUTM executes the call
- Several jobs can be created in a program unit; the corresponding messages can consist of several segments.
- For outputs in +formats, \*formats or messages in line mode you can also use the screen functions, see the section “Screen output functions in format mode” on [page 107](#).  
If you use #formats, then the KCDF must be set to binary zero, otherwise openUTM reacts with 40Z.  
Similarly, when working with edit profiles (BS2000/OSD) openUTM responds with 40Z if KCDF has not been set to binary zero.
- The jobs created with FPUT are discarded as a result of PEND ER/FR, PEND RS or RSET.
- No output job to the client with which the program is currently working (KCRN ≠ KCLOGTER) may be generated in a dialog program.
- With FPUT calls to another UTM application which was generated as a transport system application, the TAC must be located at the start of the message area.
- If necessary, the message is formatted before being output.

B  
B  
B  
B

- Asynchronous jobs are kept until:
  - the referenced program unit or printout is terminated, or, with jobs to remote asynchronous services, the transfer has been terminated successfully
  - the message is read at the terminal with KDCOUT and a new input has been made (except for the KDCLAST command) or
  - the message is read from a TAC queue by means of a DGET call and the transaction containing the DGET is completed successfully.
- Jobs with messages of length 0

If a message with a length of 0 is created (known as a "dummy message"), the following applies:

  - a background job is executed, i.e. the asynchronous service is started without receiving a message
  - an empty format is output if the job is an output job for a format terminal
  - in the case of a message for a TAC queue, an empty message is created that can be read by means of a DGET call
  - if the job is an output job for a transport system, it is accepted but will later be discarded by openUTM.
- Background jobs for an asynchronous program in the same application:

Each background job starts a separate asynchronous service. If several complete messages are sent to the same TAC in a program unit run, a separate asynchronous service is started for each message.
- If several complete messages are sent to the same TAC queue in a program unit run, each message must be read by means of a separate DGET FT call.
- There is no interaction between FPUT and DPUT calls, i.e. DPUT calls with KCMOD = "\_" and FPUT calls can be sent at a certain location independent of one another.
- Output jobs intended for a terminal are inserted into the message queue and can be retrieved by the user with the KDCOUT command. Each KDCOUT command retrieves exactly one message. Each message can only be retrieved once. If the KDCOUT command is repeated the next message is retrieved from the queue.

At the end of transaction, a message in the system line informs the terminal user whether asynchronous messages are present for the terminal.

- B  
B  
B  
B  
B
- You can suppress this message in BS2000/OSD by setting ANNOAMSG=N for the associated LTERM partner during configuration (default value: ANNOAMSG=Y). Asynchronous messages are then displayed immediately on the screen. This may interrupt the dialog. However, the terminal user can use the KDCDISP command to display the last screen again.
- B
- Print options for RSO printers
- B  
B  
B
- If you use print options for jobs to RSO printers, you should first pass the list with the print options using FPUT RP, see RSO manual. Then submit the actual print job using FPUT NT/NE.
- Handling message segments
    - Message segments in line mode are combined into **one** message and sent to the LTERM partner. The message segments generated with FPUT are gathered by openUTM and terminated with the next PEND call, provided they have not yet been terminated in the program unit run with FPUT NE. When the transaction ends, the message segments are sent as a single message to the LTERM partner or to another application.
    - With formatted message segments to terminals, each segment forms a separate message. The format name in KCMF/kcfn needs not always stay the same. At a terminal, each format (message segment) must be fetched with a KDCOUT command. Each FPUT NT call generates a separate message. It is therefore not possible to structure a screen with different partial formats using FPUT NT calls. The formats arrive in the order in which they were sent.
    - With message segments to printer, it is possible to switch between formatted message segments and unformatted message segments (in line mode). With message segments to terminals, this switch causes the old message to terminate and a new one to start.
    - With FPUT NT calls to local or remote asynchronous services or local or remote TAC queues, each message segment must be read with a separate FGET NT.
    - With PEND, the message segment most recently generated with FPUT is always assumed to be the final message segment, even if it was output with NT.
    - The maximum number of FPUT NE calls which are possible in a transaction depends on the RECBUF generation parameter in the KDCDEF statement MAX. Each FPUT NE occupies 30 bytes in this buffer. If the buffer is full, FPUT NE is rejected with KCR CDC=K704.



- If, in a sequence of message segments, the name in KCRN (i.e. the message receiver) changes but no FPUT with "NE" was entered previously, a warning is issued (04Z) and a new message is started. The previous message (sequence of message segments) is terminated. This means that if the name in KCRN is changed back to the first recipient with a subsequent FPUT, the first message is **not** continued, but a new message is begun. The message is forwarded to the receiver when the next synchronization point is reached. Parallel messages, as used in DPUT, are therefore not possible.

B  
B

- If the edit profile changes within a sequence of message segments addressed to a terminal, openUTM reacts with 45Z.

- Influence of generation parameters on the FPUT call

The following notes concern the generation of the UTM application. Further information on the individual generation parameters can be found in the openUTM manual "Generating Applications".

B  
B  
B  
B

The ANNOAMSG operand in the KDCDEF control statement LTERM defines for each LTERM partner whether asynchronous messages are to be output immediately to this terminal or announced with a message. This message appears in the system status line.

For messages for terminals and transport system applications, the total message may not exceed the value generated for the NB operand of the control statement MAX. In the case of messages to other partners the length of a message segment is limited to 32,767 bytes and the length of the total message is unlimited.

When the KDCFILE is regenerated with the UTM tool KDCUPD, you can specify in the TRANSFER statement precisely which messages you want to include in the new KDCFILE (see also the openUTM manual "Generating Applications").

- FPUT calls with distributed processing

- In the job-submitting service, the KDCS parameter area must be given the same values as for background jobs for an asynchronous program in its own application. The only thing that has to be specified as name in the KCRN field is the service ID which was assigned to the job-receiving service in the APRO AM call.
- Once the FPUT NE call has terminated (final message segment or complete message) or after return of the KDCS error code 40Z, the service ID is released. This ID can now be used for a new job submitter/receiver relationship in this service.
- If the wait time for using a session or association was set to 0 at generation time, and if no connection exists with the partner application at the time the FPUT call is issued, openUTM sets the error codes 40Z in KCRCCC and KD13 in KCRCDC after the FPUT call.

## GTDA Read from TLS

The GTDA (get data) call enables you to read a TLS block (terminal specific long-term storage) into the specified message area. The block name is assigned during generation (TLS statement for KDCDEF).

A program unit of a dialog service can only read blocks from its "own" TLS, i.e. only the TLS of the LTERM, LPAP or OSI LPAP partner, via which the service was started.

A program unit run of an asynchronous service can read blocks from all the LTERM, LPAP or OSI LPAP partners of a UTM application.

### Setting the KDCS parameter area (1st parameter)

The table below shows the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area			
	KCOP	KCLA	KCRN	KCLT
Read from TLS (in dialog program)	"GTDA"	Length	Block name	—
Read from TLS (in asynch. program)	"GTDA"	Length	Block name	LTERM name / (MASTER-)(OSI-) LPAP name

### Setting the 2nd parameter

Here you have to supply the address of the message area into which openUTM is to read the message.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"GTDA"
2.	KCLA	Length in bytes
3.	KCRN	Block name
4.	KCLT	LTERM name/ LPAP name/ -

**KDCS call**

	1st parameter	2nd parameter
5.	KDCS parameter area	Message area

**C/C++ macro call**

	Macro name	Parameters
6.	KDCS_GTDA	(nb,kcla,kcrn,kclt)

**openUTM return information**

	Message area	Contents
7.		Data
	Field name in the KB return area	
8.	KCRLM	Actual block length
9.	KCRCCC	Return code
10.	KCRCDC	Internal return code

For the GTDA call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the **GTDA** operation code.
2. In the **KCLA** field, the length of the data to be transferred from the TLS.
3. In the **KCRN** field, the name of the TLS block from which openUTM is to transfer data.
4. For asynchronous programs only:  
in the **KCLT** field, the name of LTERM, LPAP or OSI LPAP partner whose TLS is to be read from (this field is not evaluated by dialog programs).

You enter the following for the KDCS call:

5. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area to which openUTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLM.
6. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

7. The desired data in the specified message area.
8. In the **KCRLM** field, the actual length of the data in the TLS so that the program can detect deviations from the KCLA entry (important if KCLA entry is smaller). Exception: for KCLA = 0 you always have 0 returned in KCRLM.
9. In the **KCRCCC** field, the KDCS return code, see next page.
10. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### KDCS return codes for the GTDA call

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 40Z System cannot perform operation (generation error or system error, deadlock, time-out); see KCRCDC.
- 41Z Call was issued in the first part of the sign-on service although this is not permitted by the generation.
- 43Z Length entry in KCLA invalid (e.g. negative).
- 44Z Block name in KCRN unknown or invalid.
- 46Z The LTERM name in KCLT invalid (with asynchronous programs only).
- 47Z Message area missing or cannot be accessed in the specified length.

A further error code can be found in the dump:

- 71Z INIT missing in this program.

### Features of the GTDA call

- A GTDA call locks the referenced TLS block against access for all the competing program units. All other TLS block of the referenced LTERM, LPAP or OSI LPAP partner are free  
The TLS block can be explicitly unlocked with the UNLK call.  
The TLS block is also unlocked by the PEND RE/FI/SP/FC/RS/ER/FR and RSET calls.  
With PEND PA/PR/KP and PGWT KP/PR the lock remains in effect.  
  
In the [section “Action with locked storage areas \(TLS, ULS and GSSB\)” on page 88](#), there is a description of how openUTM reacts when the desired TLS block is locked.
- The TLS block is transferred in its actual length, but no longer than the length specified in KCLA. If the contents of KCLA > 0 for the GTDA call, the actual length of the data in the TLS is returned in the KCRLM field.

## INFO Request information

The different variants of the INFO (information) call enable you to obtain the following information:

- B – INFO CD (**C**ard)  
B the information stored on the user's ID card (only if terminal users have to prove their  
B authorization via card reader during sign on) or Kerberos information.
- INFO DT (**D**ate/**T**ime)  
date and time of the start of the application and program unit
- INFO SI (**S**ystem **I**nformation)  
system information (e.g. the name of the application and the host)
- INFO PC (**P**redecessor **C**onversation)  
information on a stacked service
- INFO LO (**L**Ocale **I**nformation)  
Information on the language environment of the LTERM partner
- INFO CK (**C**heck**K**)  
the KCRCCC return code, normally returned by an MPUT, FPUT or PEND call.

These variants of the INFO call differ in the meaning of the 2nd parameters (message area) to be specified in the INFO call.

openUTM provides language specific data structures to structure the message area: for COBOL in the COPY element KCINFC, for C/C++ in the *kcinf.h*. include file.

## Setting the KDCS parameter area (1st parameter) and the 2nd parameter

The table below shows the five types of INFO call and the necessary entries in the KDCS parameter area.

	Function of the call	Entries in the KDCS parameter area				Meaning of the 2nd parameter
		KCOP	KCOM	KCLA	KCLT	
B B	Read ID card or Kerberos information	"INFO"	"CD"	Length	—	Area for ID card or Kerberos information
	Obtain date and time of application and program starts	"INFO"	"DT"	30	—	Area for date and time, see data structure for INFO DT.
	Information about language environment of LTERM partner	"INFO"	"LO"	68	LTERM name	Area for requested name, see data structure for INFO LO.
	Information on a stacked service	"INFO"	"PC"	39	—	Area for requested names, see data structure for INFO PC.
X/W X/W X/W B B	Obtain system information	"INFO"	"SI"	Unix system/ Windows system: 52 BS2000/OSD: 49	—	Area for requested names, see data structure for INFO SI
	Check UTM call	"INFO"	"CK"	—	—	Parameter area, to be checked

Due to the different meanings of the 2nd parameter, the INFO call is presented here in two formats:

Format 1: INFO CD/DT/LO/PC/SI (see next page).

Format 2: INFO CK (see [page 298](#)).

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"INFO"
2.	KCOM	"CD"/"DT"/"LO"/"PC"/"SI"
3.	KCLA	Length of message area in bytes
4.	KCLT	Name of LTERM partner/ —

**KDCS call**

	1st parameter	2nd parameter
5.	KDCS parameter area	Message area

**C/C++ macro calls**

	Macro name	Parameters
6.	KDCS_INFOCD / KDCS_INFODT KDCS_INFOSI / KDCS_INFOPC	(nb,kcla)
	KDCS_INFOLO	(nb,kcla,kclt)

**openUTM return information**

	Message area	Contents
7.	<input type="text"/>	Data
	Field name in the KB return area	
8.	KCRLM	Actual block length
9.	KCRCCC	Return code
10.	KCRCDC	Internal return code



For the INFO call you make the following entries in the KDCS parameter area to read the ID card information, obtain date and time, obtain system information and obtain information on the predecessor or on the language environment of the LTERM partner:

1. In the **KCOP** field, enter the INFO operation code
2. In the **KCOM** field, enter the desired function of the call:

B	CD	to read ID card or Kerberos information (CARD)
	DT	to obtain time and date of the start of the application and program unit (DATE/TIME)
	LO	to obtain information on the language environment of the LTERM partner
	PC	to obtain information on the predecessor in the stack
	SI	to obtain system information

3. In the **KCLA** field, the length of the message area used in which openUTM is to store the information. You have to specify the length in bytes. openUTM transfers the information up to a maximum length of KCLA.

4. Only with INFO LO:

in the **KCLT** field, enter the name of the LTERM partner whose language environment is to be obtained.

If KCLT is set to binary zero before the call, openUTM transfers the data of the LTERM partner via which the service is to be started.

B	If the LTERM partner belonging to the program unit run is specified in KCLT, openUTM transfers additional information about the associated physical terminal (PTERM).
---	---

You specify the following for the KDCS call:

5. 1st parameter:  
the address of the KDCS parameter area

2nd parameter:

the address of the message area into which openUTM is to write the information. openUTM transfers the requested information in a fixed structure. For this, language specific data structures are available: for COBOL in the KCINFC COPY element, for C/C++ in *kcinf.h*. include file.

6. The use of C/C++ calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

7. The desired data in the specified message area.
8. In the **KCRLM** field, the actual length of the data transferred. With  $KCRCCC \geq 40Z$  the length is 0.  
 With INFO CD, **KCRLM** specifies the length of the Kerberos information transferred to the receiving area.
9. In the **KCRCCC** field, the KDCS return code. 0
10. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### KDCS return codes for the INFO CD/DT/LO/PC/SI calls

The following codes can be analyzed in the program:

- 000 The requested information was transferred to the message area in its full length.
- 01Z Information was transferred. However, the message area is too short and the information was truncated.
- 09Z only for INFO CD:  
 The Kerberos dialog returned an error or the Kerberos information was returned in truncated form because it was longer than the value generated with MAX PRINCIPAL-LTH. KCRLM shows the length in which the Kerberos information was transferred to the receiving area.
- 40Z Generation error or system error. See KCRCDC.
- 41Z INFO call with KCOM = CD/PC not permitted in asynchronous program.
- 42Z KCOM invalid.
- 43Z KCLA invalid.
- 46Z Only with INFO LO:  
 the LTERM name specified in the KCLT field is invalid.
- 47Z Message area missing or cannot be accessed in the specified length
- 49Z Only with INFO LO:  
 unused parameters were not set to binary zero for the call.

Return code 71Z can be found in the dump:

- 71Z INIT not yet issued in this program unit

**B** **Features of the INFO CD call**

**B** With this function the INFO call reads the ID card information stored during sign-on to the specified message area if the user used a magnetic strip card for authentication during sign on. For more information see also [section "Support for ID card readers" on page 99](#).


**B** The stored Kerberos information can be read if either the user was signed on by means of Kerberos, or a Kerberos dialog has been executed for the client and afterwards no further user signed on using a magnetic strip card.

**B** The INFO CD call for reading ID card information or Kerberos information is permitted in dialog programs only.

**B** In order to be able to call the ID card information in the program, you have to set the following operands during generation:

**B** – the CARD= operand must be specified for the current user ID in the KDCDEF control statement USER, and

**B** – the CARDLTH= operand in the KDCDEF control statement MAX.

**B**  openUTM sets the identifier "A" in the KCAUSWEIS/kccard field of the KB header if the card was inserted during the preceding input.

### Features of the INFO DT call

With this function the INFO call writes the start times of the application and program unit to the specified area, using a length of 30 bytes. The information is structured as follows:

Field name COBOL	Field name C/C++	Byte	Meaning of the information
KCDATAS	—	1 - 6	Date of application startup in the form <i>ddmmyy</i> , where:
KCTAGAS	as_day	1 - 2	<i>dd</i> - day (value range 01 - 31)
KCMONAS	as_mon	3 - 4	<i>mm</i> - month (value range 01 - 12)
KCJHRAS	as_year	5 - 6	<i>yy</i> - year (value range 00 - 99))
KCTJHAS	as_doy	7 - 9	Day of application startup (working day, value range 001 - 366)
KCUHRAS	—	10 - 15	Time of application startup in the form <i>hhmmss</i> , where:
KCSTDAS	as_hour	10 - 11	<i>hh</i> - hour (value range 00 - 23)
KCMINAS	as_min	12 - 13	<i>mm</i> - minute (value range 00 - 59)
KCSEKAS	as_sec	14 - 15	<i>ss</i> - second (value range 00 - 59))
KCDATAK	—	16 - 21	Date of program unit startup in the form <i>ddmmyy</i> , where:
KCTAGAK	ps_day	16 - 17	<i>dd</i> - day (value range 01 - 31)
KCMONAK	ps_mon	18 - 19	<i>mm</i> - month (value range 01 - 12)
KCJHRAK	ps_year	20 - 21	<i>yy</i> - year (value range 00 - 99)
KCTJHAK	ps_doy	22 - 24	Day of application startup (working day, value range 001 - 366)
KCUHRAK	—	25 - 30	Time of program unit start in the form <i>hhmmss</i> , where:
KCSTDAK	ps_hour	25 - 26	<i>hh</i> - hour (value range 00 - 23)
KCMINAK	ps_min	27 - 28	<i>mm</i> - minute (value range 00 - 59)
KCSEKAK	ps_sec	29 - 30	<i>ss</i> - second (value range 00 - 59))

**B Features of the INFO LO call (BS2000/OSD)**

**B** Using the INFO LO variant of the INFO call the program unit run can request information  
**B** about the location of LTERM partners and application. The location of an LTERM partner  
**B** determines the language environment of the client. The location of the application deter-  
**B** mines the default setting for the language environment.

**B** The call returns the following data:

- B** – the location of the LTERM partner whose name is specified in the KCLT field
- B** – the location of the application

**B** If KCLT is set to binary zero before the call, openUTM transfers the data of the LTERM  
**B** partner via which the service was started.

**B** If the call is executed in a dialog service and the name of the LTERM partner is set in the  
**B** KCLT field associated with the program unit run, openUTM also provides the following data  
**B** about the associated physical terminal (PTERM):

- B** – number of extended ISO character sets supported by the terminal or printer
- B** – ISO variant numbers of all supported character sets
- B** – default user character set, allocated to the BS2000 user ID under which the UTM appli-  
**B** cation is running.

B With the INFO LO call openUTM displays the data in the message area structured as  
B follows:

B	Field name	Byte	Meaning of contents
B	KCLTLANG	1 - 2	Language identifier of LTERM partner
B	KCLTTERR	3 - 4	Territory identifier of LTERM-partner
B	KCLTCCSN	5 - 12	CCS-name of LTERM-partner
B		13 - 20	Blanks
B	KCAPLANG	21 - 22	Language identifier of application
B	KCAPTERR	23 - 24	Territory identifier of application
B	KCAPCCSN	25 - 32	CCS-name of application
B		33 - 40	Blanks
B	KCDEFCCS	41 - 48	User default character set of BS2000 user ID under which the UTM application runs
B	Information about the connected terminal		
B	(only out put in dialog services if KCLT contains the name of the LTERM partner associated with the service. If the partner is not a terminal the values is X"00").		
B	KCCCSNO	49	Number of the extended ISO-character sets supported by the terminal
B	KCHSET1	50	Variant number of the first supported ISO character set
B	KCHSET2, KCHSET3 ... to KCHSET16	51 - 65	Variant numbers of other ISO character sets supported by the physical terminal. Length of each field is 1 byte.
B		65 - 68	Blanks

**X/W Features of the INFO LO call (Unix systems and Windows systems)**

X/W With the INFO LO variant of the INFO call, the program unit run can request information about the language currently set for the LTERM partner.

X/W This function enables you to run program units in multiple languages. In the program unit run, the language set for an LTERM partner can be queried with INFO LO and messages sent in the same language.

X/W INFO LO returns the following information:

X/W – Language and territory identifiers as well as the \$LANG variable of the LTERM partner specified in KCLT. The cell outputs the currently set \$LANG variable of the user ID under which the associated dialog terminal process was started.

X/W – Language and territory identifiers as well as the \$LANG variable of the UTM application. The cell outputs the currently set \$LANG variable of the user ID under which the associated application was started.

X/W Language and territory identifiers are retrieved from the \$LANG variable at program runtime. Example: From \$LANG=En\_US.ASCII, openUTM creates the language identifier En and the territory identifier US.

X/W The INFO LO call provides data structured as follows:

X/W	Field name	Byte	Meaning of the contents
X/W	KCLTLANG	1-2	Language identifier of the LTERM partner
X/W	KCLTTERR	3-4	Territory identifier of the LTERM partner
X/W	KCLTNLSL	5-20	\$LANG variable of the specified LTERM partner
X/W	KCAPLANG	21-22	Language identifier of the application
X/W	KCAPTERR	23-24	Territory identifier of the application
X/W	KCAPNLSL	25-40	\$LANG variable of the application
		41-68	Blanks

### Features of the INFO PC call

With this call you can query information about a stacked service.

If the current service was started by service stacking, the INFO PC call provides, for example, the following information about the preceding service (that is the service that is immediately before the current service in the stack):

- date and time of last processing
- format identifier of last screen output
- the next TAC
- the service TAC

openUTM writes this information to the specified area, using a total length of 39 bytes.

Field name	Byte	Meaning of the information
KCPFN	1 - 8	Format identifier of the last screen output
KCPNXTAC	9 - 16	Name of the next TAC
KGPCVTAC	17 - 24	Name of the service TAC
KCPLDATE <sup>1</sup>	25 - 33	Date of last processing in the form ddmmyy, where:
KCPLDAY	25 - 26	dd - day (value range 01 - 31)
KCPLMON	27 - 28	mm - month (value range 01 - 12)
KCPLYEAR	29 - 30	yy - year (value range 00 - 99)
KCPLDOY	31 - 33	Working day of last processing, value range 001 - 366
KCPLTIME1	34 - 39	Time of last processing in the form hhmmss, where:
KCPLHOUR	34 - 35	hh - hour (value range 00 - 23)
KCPLMIN	36 - 37	mm - minute (value range 00 - 59)
KCPLSEC	38 - 39	ss - second (value range 00 - 59)

<sup>1</sup> For C/C++ the summary fields KCPLDATE and KCPLTIME are not defined. However, the specific fields for day/month/year/working day/hour/minute/second are defined.

If there is no stacking, openUTM returns blanks.



## Features of INFO SI call

With this call, you can query information on the application and system, e.g. the name of the application and the host on which the application is running.

In dialog services INFO SI also provides the PTERM name, host name and application name of the communication partner.

openUTM writes this information to the specified area as follows - byte 17 - 40 are set to binary zero in asynchronous services.

Field name	Byte	Meaning of the information
KCAPPLNM / applnam	1 - 8	Name of the UTM application
KCHOSTNM / hostname	9 - 16	Name of the host
KCPTRMNM	17 - 24	In a dialog service For distributed processing via LU6.1: CON name of the communication partner For distributed processing via OSI TP: OSI-CON name of the communication partner Otherwise the PTERM name of the communication partner In an asynchronous service: Blanks
B B KCPRONM	25 - 32	Processor name of the caller (PRONAM name); only in the dialog service: blanks in the asynchronous service
X/W KCPRONM	25 - 32	Blanks
KCBCAPNM	33 - 40	Application name of the caller (BCAMAPPL name), only in dialog services
KCVERS	41 - 46	openUTM version <i>V<sub>mm.nx</sub></i> (e.g. V06.1A)
KCIVER	47 - 48	Version number of the UTM interface <sup>1</sup>
KCIVAR	49	"B" for BS2000, "X" for Unix system or "N" for Windows system
X/W KCFILL1	50	Not currently used
X/W X/W X/W KCLANG	51 - 52	Language identifier, e.g. "EN" for English (The value of this language identifiers corresponds to the value of the first two bytes of the \$LANG variable)

<sup>1</sup> The version number indicates the version of function extensions available, irrespective of the product variant. As the functionality of the KDCS interface is extended, the version number is increased by 1 each time. For openUTM V6.1 it is 8. In this way, UTM application programs can be created to run in different product variants and UTM versions.

## INFO CK call

With this function the INFO call checks whether the specifications for an MPUT, FPUT or PEND call in KCRN are permitted. openUTM transfers the KDCS return code expected for the checked call to KCRINFCC. INFO CK checks the following calls:

- MPUT NT/NE
- FPUT NT/NE
- PEND PA/PR/KP/SP/RE/FC.

This function enables you, for example, to check prior to a PEND RE call whether the intended follow-up TAC is permitted for this call. This may, for example, prevent the service from aborting.

**Setting the parameters**

	Field name in KDCS parameter area	Contents
1.	KCOP	"INFO"
2.	KCOM	"CK"
	Message area	
3.		KDCS parameter area of the call to be checked

**KDCS call**

	1st parameter	2nd parameter
4.	KDCS parameter area	Message area

5. **C/C++ macro calls**

Macro name	Parameter
KDCS_INFOCK	(nb)

**openUTM return information**

	Field name in the KB return area	Contents
6.	KCRINFCC	KCRCCC return code of the checked call
7.	KCRCCC	Return code
8.	KCRCDC	Internal return code

For the INFO call you make the following entries in the KDCS parameter area to check the KDCS parameter area for a UTM call:

1. In the **KCOP** field, enter the INFO operation code.
2. In the **KCOM** field, enter the desired function of the call: CK to check the KDCS parameter area for an MPUT-, FPUT-, or PEND call.

You also have to

3. Specify the KDCS parameter area to be checked according to the rules valid for the respective call (MPUT, FPUT, PEND).

You specify the following for the KDCS call:

4. 1st parameter: the address of the KDCS parameter area.  
2nd parameter: the address of the area containing the KDCS parameter area of the call to be checked.
5. The use of C/C++ macro calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

6. in the **KCRINFCC** field, the return code expected in KCRCCC of the checked call. openUTM enters this error code only if the INFO call runs according to plan (the return code for the INFO call has the value 000). KCRINFCC has the value 78Z if the specified function form of the call to be checked (KCOP and KCOM entries in the second parameter area) is not supported by the INFO call.
7. in the **KCRCCC** field, the KDCS return code.
8. in the **KCRCDC** field, the internal return code of the checked call (see the openUTM manual "Messages, Debugging and Diagnostics").

**KDCS return codes for the INFO CK call**

The following codes can be analyzed in the program:

000 The call specified in the 2nd parameter was checked.

40Z Generation error or system error (see KCRCDC).

42Z KCOM of the INFO call invalid.

47Z Address of 2nd parameter omitted or invalid.

Return code 71Z can be found in the dump:

71Z INIT not yet issued in this program unit.

**Features of the call INFO CK**

If KCCARD was specified, the INFO CK call in MPUT checks whether the terminal for which the message is destined was generated with an ID card reader or whether the USER is already signed on with an ID card.

- B** This check is also performed if a corresponding edit profile was specified.
- B** This check is only performed with magnetic strip cards, not with chipcards.

## INIT Initialize program unit

The INIT (initiate program) call is used to sign-on a program unit to openUTM. There are the following variants.

- INIT (no entry in KCOM)  
Initialize program unit
- INIT PU (**P**rogram **U**nit)  
Initialize program unit and request additional information.
- INIT MD (**M**odify)  
Initialize program unit and change the size of the KB program area.

The following applies for these variants.

- The INIT call initiates cooperation between the program unit run and openUTM. It is the first KDCS call allowed in a program unit run, i.e. you are not allowed to enter any other KDCS or database calls prior to the INIT call.
- You are not allowed to enter INIT or INIT PU more than once in a program unit run.
- INIT MD may be specified more than once in a program unit. If you specify the INIT MD call as the first INIT call in the program unit, it is treated like an INIT without an operation modifier.
- You may not use the communication area (KB) and the standard primary working area (SPAB) between start of the program unit run and the first INIT call.
- Following the first INIT, openUTM makes the entire communication area (KB), including the KB parameter area, available to the program unit run. This area has the length specified in KCLKBPRG/kclcapa.
- The INIT MD call enables the length of the KB program area to be adjusted during the program unit run. This may be necessary if, for example, the size of the KB program area that openUTM is to save with the PEND call is not determined until processing has started.  
Example: Data that is read from a database is to be saved in the KB program area. If the length of the KB program area is adjusted to the volume of data read by INIT MD, openUTM need not save more data than necessary or less data than necessary at the synchronization point.
- If you use the INIT call with the operation modifier PU (**P**rogram **U**nit), openUTM provides the program unit additional information about application, system and communication partner in the message area.

### Setting the 1st parameter (KDCS parameter area)

The following table shows the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area				
	KCOP	KCOM	KCLKBPRG/kclcapa	KCLPAB/kclspa	KCLI
Initialize program unit	"INIT"	—	Length of KB program area (if used)	Length of SPAB (if used)	—
Initialize program unit and request information	"INIT"	"PU"	Length of KB program area (if used)	Length of SPAB (if used)	Length of message area
Change length of KB program area	"INIT"	"MD"	Length of KB program area	—	—

You only have to specify the length of the KB program area if you want to use it in the program unit run. The specified length must not be greater than the maximum length defined for this application at generation (operand KB in the MAX statement for KDCDEF).

You only have to specify the length of the SPAB if you want to use it in the program unit run. It must not be longer than the value defined at generation (operand SPAB in the MAX statement for KDCDEF).

### Setting the 2nd parameter (only necessary with INIT PU)

Here you enter the address of the message area to which openUTM is to write the requested information.

You can use language-specific data structures to structure the message area. For COBOL, they are defined in the KCINIC COPY element and for C/C++ in the *kcini.h* include file.

Specify the version number of the structure and select the return information requested from openUTM in the header of the data structure. These other fields (return information) are described on [page 316](#).

### Setting the parameters

	Field name in the KDCS parameter area	Contents
1.	KCOP	"INIT"
2.	KCOM	— / "PU" / "MD"
3.	KCLKBPRG/kclcapa	Length in bytes
4.	KCLPAB/kclspa	— /length in bytes
5.	KCLI	— / Length in bytes (with INIT PU)

### Setting the header in the message area (only necessary with INIT PU)

	Field name in message area	Contents
6.	KCVER/if_vers	Version number ( 5 )
7.	KCDATE/dattim_info	Request date and time (Y/N)
8.	KCAPPL/appl_info	Request application information (Y/N)
9.	KCLOCALE/locale_info	Request Locale information (Y/N)
10.	KCOSITP/ositp_info	Request OSI TP information (Y/N)
11.	KCENCR/encr_info	Request encryption information (Y / N )
12.	KCMISC/misc_info	Request miscellaneous information (Y / N )

### KDCS call

	1st parameter	2nd parameter
13.	KDCS parameter area	— / Message area (with INIT PU)



14. **C/C++ macro calls**

Macro name	Parameters
KDCS_INITMD	(kclcapa)

**openUTM return information**

	Message area (only with INIT PU)	Contents
15.	KB header area	Additional information
16.	Field name in KB return area	Current data
17.	KCRLM (only with INFO PU)	Length of transferred data
18.	KCRCCC	Return code
19.	KCRCDC	Internal return code
20.	KCRMF/kcrfn	Format identifier/blanks
21.	KCRPI	Service ID/Reset ID/blanks
22.	KB program area KCKBPRG/kclcapa	Data

For the INIT call you make the following entries in the KDCS parameter area:

1. In the **KCOP field**, enter the INIT operation code.
2. In the **KCOM field**, the operation modifier:
  - PU if openUTM is to make additional information available in the message area
  - MD if the length of the KB program area is to be changed
3. In the **KCLKBPRG/kclcapa field**, enter the length of the KB program area in bytes (if used). It must not exceed the length predefined at generation time (operand KB in the MAX statement), otherwise the generated value is taken.
4. Only with INIT or INIT PU  
In the **KCLPAB/kclspa field**, enter the length of the standard primary working area (SPAB) used in the program unit run in bytes. It must not exceed the length predefined at generation time (operand SPAB in the MAX statement).

5. Only with INIT PU:  
in the **KCLI** field, enter the length of the message area to which openUTM is to transfer the information. Enter the length in bytes. The information transferred to the message area by openUTM has a maximum length of KCLI.

Setting the header of the message area (only necessary with INIT PU):

6. In the **KCVER/if\_ver** field, enter the version number of the data structure. The current version is version 5.
7. Enter Y in the **KCDATE/dattim\_info** field if you want information on the date and time of the start of the application and the program unit run, otherwise enter N.
8. Enter Y in the **KCAPPL/appl\_info** field if you want to request information about the application, system and communication partner, otherwise enter N.
9. Enter Y in the **KCLOCALE/locale\_info** field if you to want to request information about the language environment of the user ID, otherwise enter N.
10. Enter Y in the **KCOSITP/ositp\_info** field if you require OSI TP specific information, otherwise enter N.
11. Enter Y in the **KCENCR/encr\_info** field if you require information on the encryption methods used to encode between the client and the UTM application, otherwise enter N. (The encryption mechanism can be coordinated. See the openUTM manual "Generating Applications".)
12. Enter Y in the **KCMISC/misc\_info** field if you require miscellaneous information (e.g. number of queued messages in the user's queue, password validity, time of last sign-on), otherwise enter N.

You specify the following for the KDCS call:

13. 1st parameter: the address of the KDCS parameter area.  
2nd parameter (only necessary with INIT PU):  
the address of the message area to which openUTM is to write information (see [page 303](#)).
14. The use of C/C++ macro calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

15. only with INIT PU:  
in the **message area**, the information transferred by openUTM up to a maximum length of the value specified in KCLI.
16. in the **KB header area** the current data of the KB header (see table).
17. only with INIT PU:  
in the **KCRLM** field, the length of the information actually transferred by openUTM, provided that KCRCCC = 000 or 01Z. If KCRCCC  $\geq$  40Z, no information is transferred. Thus, in such cases KCRLM=0.
18. in the **KCRCCC** field, the KDCS return code, see below
19. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").
20. only with INIT or INIT PU  
in the **KCRMF/kcrfn** field:
  - For a message from a terminal:  
Blanks (in line mode) or the format name (in format mode) of the last screen output, i.e. the name specified in the KCMF/kcfn field with the MPUT of the last dialog step. If the last output consisted of multiple partial formats, KCRMF/kcrfn contains the name of the first partial format into which data was entered. If no data was entered in any of the partial formats, KCRMF/kcrfn contains the name of the first partial format.
  - If an edit profile was used in the last screen output, KCRMF/kcrfn contains this edit profile.
  - after PEND PA/PR/PS/SP/FC: blanks.
  - if a reset message exists (after PEND RS): blanks.
  - for distributed processing via LU6.1:  
in the job-submitting service:  
in the 1st program unit of a job-submitting service, the format identifier of the terminal format or blanks. In the follow-up program unit of the job-submitting service, the format identifier of the first message segment sent by the job-receiving service to the job-submitting service, or blanks if a status flag exists for the service ID specified in KCRPI.  
in the job-receiving service:  
the format identifier of the first message segment from the job-submitting service that can be read by the job-receiving service with MGET.

B  
B

- for distributed processing via OSI TP:

If the program unit run was started because of a distributed dialog, KCRMF/kcrfn in the job-submitting service contains the name of the abstract syntax which was allocated to the message by the job submitter; if the field contains blanks, the abstract syntax of the UTD syntax is selected.

In the job-receiving service KCRMF/kcrfn contains the name of the abstract syntax which was allocated to the message by the job-receiving service described in KCRPI. If the field contains blanks, the abstract syntax of the UTD syntax is selected or an error message from the partner is present.

21. only with INIT or INIT PU  
in the **KCRP** field:

- For a message from a UPIC client program, terminal or a program unit in the same application: blanks.
- If a reset message exists: the reset ID.
- In the job-submitting service with distributed processing:  
the service ID of the job-receiving service if a message from the job receiver exists.
- In the job-receiving service with distributed processing: blanks

22. In the **KCKBPRG/kclcapa** (KB program area) field, the data of the service, provided the program unit run includes service-specific data from another program. If this is the first program unit run of a service, the area is undefined or set with the generated fill character. If length zero is specified in KCLKBPRG/kclcapa, no data is transferred to the follow-up program unit run.

**KDCS return code in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Operation carried out.  
with INIT PU: the requested information was transferred to the message area in full length.
- 01Z Length specified in KCLKBPRG/kclcapa exceeds value specified when the application was generated.
- 02Z With INIT or INIT PU:  
Length specified in KCLPAB/kclspa exceeds value specified when the application was generated.
- 07Z With INIT PU:  
Function was executed, the available message area is too short (length in KCLI insufficient). No or incomplete information was returned.
- 48Z With INIT PU:  
Invalid data structure version.

Additional error codes can be found in the dump:

- 70Z Operation cannot be performed (system or generation error). Further information is provided by the internal openUTM error code KCRCDC.
- 71Z With INIT or INIT PU:  
INIT call already issued in this program unit run.
- 73Z Negative length specification. With INIT PU: KCLI is invalid.
- 77Z Message area is missing or cannot be accessed in the specified length.
- 88Z Interface version is invalid.
- 89Z With INIT PUT or INIT MD:  
When the function was called, unused parameters were not set to binary zero.

### openUTM return information in the header of the KDCS communication area

openUTM returns the following entries among others to the INIT call: the TAC of the service start, current TAC, date, time and the LTERM partner of the sender .

The generated length of the KB program area is located in the KCLKBPB/kclpa field. The table below shows which entries openUTM returns in the KB header:

Field name COBOL	Field name C/C++	Contents (entered by openUTM)
KCBENID	kcuserid	UTM user ID under which the client is working. When working without user IDs: KCBENID/kcuserid=KCLOGTER. With distributed processing via LU6.1:local session name (LSES). With distributed processing via OSI TP and Security type "N": Local name of the connection (ASSOCIATION), otherwise user ID
Service-specific data:		
KCTACVG	kccv_tac	TAC used to start this service
KCTAGVG	kccv_day	Day of service start
KCMONVG	kccv_month	Month of service start
KCJHRVG	kccv_year	Year of service start
KCTJHVG	kccv_doy	Working day of service start
KCSTDVG	kccv_hour	Hour of service start
KCMINVG	kccv_minute	Minute of service start
KCSEKVG	kccv_second	Second of service start
KCKNZVG	kccv_status	Service ID  F First program unit run of dialog service A First program unit run of an synchronous service N Follow-up program unit run of service C First program unit of a chained service R Restart of a service D End of service through loss of connection (only with LTERM partners which have been generated with RESTART=NO) Z End of service by abort E Normal end of service L End of last process with normal termination  The service identifiers D, Z, E can only occur in the VORGANG exit, the service identifier L only in the SHUT exit. All other service identifiers can occur either in the KDCS call INIT or in the exit VORGANG.

Field name COBOL	Field name C/C++	Contents (entered by openUTM)
Program unit-specific data:		
KCTACAL	kcpr_tac	TAC used to address the program
KCSTDAL	kcpr_hour	Hour of program unit start
KCMINAL	kcpr_minute	Minute of program unit start
KCSEKAL	kcpr_second	Second of program unit start
KCAUSWEIS	kccard	ID card identifier: A (card inserted) or blanks.
KCTAIND	kctaind	Transaction indicator: F (first) or N (next transaction)
KCLOGTER	kclogter	LTERM name (sender), for distributed processing: LPAP or (MASTER-)OSI-LPAP name
KCTERMN	kctermn	Communication partner identifier, for distributed processing via LU6.1: CON...,TERMN=, for distributed processing via OSI TP: OSI-LPAP...,TERMN= otherwise: PTERM ..., TERMN (see also the table by PTERM in the openUTM manual "Generating Applications")
KCLKBPB	kclpa	Maximum length of the KB program area as defined at generation time
Data for service stack:		
KCHSTA	kchsta	Stack height, i.e. the number of stacked services as seen from the current service (0 through 15).
KCDSTA	kcdsta	Change of stack height: + (increased), - (decreased) or 0 (unchanged, also in the event of stacking after returning from an inserted service)
KCPRIIND	kcprind	Program indicator: A =asynchronous service, D = dialog service
KCOF1	kcof1	OSI TP functions in an OSI TP job-receiving service.
KCCP	kccp	Indicator for the client protocol: 0 Asynchronous processing 1 LU6.1 2 OSI TP 3 UPIC 4 DTP 4 TIAM 5 APPLI 6 SOCKET
KCTARB	kctarb	Information on reset of an OSI TP transaction
KCYEARVG	kccv_year4	Year of service start (four positions)

X/W  
B

**Features of the KB program area and the SPAB**

- The KB program area is assigned to a service, the SPAB to a program unit run.
- At the start of the service the contents of the KB program area and the SPAB are undefined or the areas are preset with the generated fill character. Such a fill character can be used, for example, to facilitate error location in the test or for data protection. The SPAB and KB program area are preset with this character at the start of a process and overwritten with it at the end of a processing step, see openUTM manual "Generating Applications".
- If, at the INIT of a program unit, a KB program area is defined with a length of  $n$  bytes and a larger KB program area of  $m$  bytes ( $m > n$ ) is requested in the next program unit run, the last  $(m - n)$  bytes of the KB program area are likewise undefined or padded with the generated fill character.



## Particularities of the INIT calls with distributed processing

- INIT call in the job-submitting service

If the distributed transaction is reset, openUTM returns the service indicator "R" at INIT of the first transaction of a follow-up transaction (in the KCKNZVG/kccv\_status field of the KB header). In this case it is usually status information of the job-receiving service.

In the follow-up program unit the INIT call in the job-submitting service supplies the following additional information in the KDCS return area:

- KCRPI contains the service identifier of the job-receiving service which started this program unit.
- KCRMF/kcrfn contains the format identifier which the job-receiving service entered in the first message segment to the job-submitting service, otherwise blanks.

The first MGET call for reading the results must be issued with KCRN=KCRPI and KCMF=KCRMF (or: kcrfn=kcrfn).

- INIT call in the job-receiving service

There are the following modifications for entries in the KB header:

KCBENID/kcuserid

With LU6 protocol: contains the local session name (LSES name, see the LSES statement for KDCDEF)

With OSI TP protocol with security type "N": local connection name (ASSOCIATION name, see the OSI-LPAP statement for KDCDEF), otherwise user ID

KCAUSWEIS/kccard

Contains blanks, i.e. ID card reader is not supported.

KCLOGTER

Contains the logical name of the job-submitting application (LPAP name, or (MASTER-)OSI-LPAP name, see LPAP statement for LU6 protocol, or OSI-LPAP statement for OSI TP protocol: for KDCDEF).

KCTERMN

Contains the identifier (**T**erminal **M**nemonic) of the job-submitting application, (see operand TERMN= of the CON statement for LU6.1 protocol, or the OSI-LPAP statement for OSI TP protocol: for KDCDEF).

**KCOF1** shows the OSI TP functions in an OSI TP job-receiving service for the dialog used to select the job submitter. The following values are possible:

Blanks

The current service is not a job-receiving service or the OSI TP protocol is not used for communication with the job-submitting service.

**B** Basic functions

**H** Basis and handshake functions

**C** Basic and commit functions with chained transactions

**O** (other combination)

No standard combination of OSI TP functions was selected for the dialog with the job submitter. The selected OSI TP functions can only be read with an INIT PU call.

**KCCP** (**client protocol**)  
shows the protocol used for the communication.

**1** LU6.1

**2** OSI TP

**KCTARB** shows in an OSI TP service whether a situation occurred in a previous PGWT call which requires the reset of a transaction.

Blanks

a situation has occurred which requires the reset of a transaction.

**Y** a situation has occurred in a previous PGWT call which does not permit the set forward of the transaction and the transaction has not been reset yet. Communication with the partner services is permitted. A call to set forward results in an abnormal end of service.

The KCRMF/kcrfn field in the KB return area provides information about the partner service.

## Message area returns with INIT PU

If you use the INIT call with the operation modifier PU, openUTM supplies additional information for the program unit in the message area about application, system and communication partner.

You can use language specific data structures to structure the message area. For COBOL, they are defined in the KCINIC COPY element and for C/C++ in the *kcini.h* include file. In the header of the data structure you have to define which information openUTM is to return.

openUTM supplies the following information:

- The generated lengths for the KB program area and the standard primary working area. Any variant of the INIT PU call supplies this information.
- Date and time of the application starts and the start of the program unit run.
- Information about application and system
- Information about the communication partner:

In a dialog service, this are information about:

- name of the communication partner
- host processor name of the communication partner
- name of the UTM application via which communication with the communication partner was established (BCAMAPPL name)

In an asynchronous service blanks are transferred.

- Information about the language environment of the user ID who started the service.

X/W  
X/W  
X/W  
X/W  
This information comprises language and territory identifier as well as \$LANG variable. Language and territories identifier are retrieved from the \$LANG variable during the runtime of the program. Example: From \$LANG=En\_US.ASCII openUTM creates the language identifier En and the territory identifier US.

X/W  
X/W  
In an asynchronous service, the language of the user who started the service is transferred.

B  
B  
B  
B  
B  
B  
This information comprises:

- Language and territory identifier and the character set of the user. If no user is signed on, openUTM transfers the language and territory identifier and the character set of the LTERM partner.
- Name of the character set of the message.
- Information whether the user is connected to a 7- or 8-bit terminal.

B  
B  
B  
In a locally started asynchronous service, the locale of the user who started the service is transferred. In such an asynchronous service, the 8-bit terminal information contains the value "7" and the character set name of the message contains blanks.

- Information about the job-submitting service when communication is carried out via OSI TP.
- Information about the encryption method used between the UTM application and the client.
- Miscellaneous information, e.g. the number of queued messages in the user's queue, password validity, time the user last signed on, properties of the LTERM and OSI-LPAP partner which started the service with respect to LTERM groups and LTERM/LPAP bundles.
- Information about asynchronous messages for the user.

### Structure of the message area with INIT PU (with KCINIC or kcini.h)

Field name COBOL	Field name C/C++	Length in bytes	Description
Header - Version number and requested information			
KCVER	if_ver	2	To be assigned before the call: Version number of the data structure (5)
KCDATE	dattim_info	1	To be assigned before the call: Request date and time (Y/N)
KCAPPL	appl_info	1	To be assigned before the call: Request application information (Y/N)
KCLOCALE	locale_info	1	To be assigned before the call: Request location information (Y/N)
KCOSITP	ositp_info	1	To be assigned before the call: Request OSI TP information (Y/N)
KCENCR	encr_info	1	To be assigned before the call: encryption information (Y/N)
KCMISC	misc_info	1	To be assigned before the call: miscellaneous information ( Y / N )
		8	Reserved for future extensions
General information, which is always returned:			
KCGPAB	gen_spab_lth	2	Generated length of SPAB
KCGNB	gen_nb_lth	2	Generated length of message area
Information about date and time of the start of the application and the program unit. (only if KCDATE=Y)			
KCADAY	as_dt_day	2	Day of application start
KCAMONTH	as_dt_month	2	Month of application start
KCAYEAR	as_dt_year	4	Year of application start

X/W

B  
B  
B  
B  
B  
B  
B  
B  
B  
B  
B

Field name COBOL	Field name C/C++	Length in bytes	Description
KCADOY	as_dt_doy	3	Day of year of application start
KCAHOUR	as_tm_hour	2	Hour of application start
KCAMIN	as_tm_minute	2	Minute of application start
KCASEC	as_tm_second	2	Second of application start
KCASEAS	as_season	1	Time of application start is specified in normal time (W) or summer time (S). If the operating system does not supply information about summer/normal time, blanks are output.
KCPDAY	ps_dt_day	2	Day of program start
KCPMONTH	ps_dt_month	2	Month of program start
KCPYEAR	ps_dt_year	4	Year of program start
KCPDOY	ps_dt_doy	3	Day of the year of program start
KCPHOUR	ps_tm_hour	2	Hour of program start
KCPMIN	ps_tm_minute	2	Minute of program start
KCPSEC	ps_tm_second	2	Second of program start
KCPSEAS	ps_season	1	Time of application start is specified in normal time (W) or summer time (S). If the operating system does not supply information about summer/normal time, blanks are output.
KCTMZONE	time_zone	12  12	Blanks  Time zone in format <i>sHH:MM-hh:mm</i> , where:  <i>s</i> "+", or "-": sign of time difference between local time zone and UTC (Universal Time Coordinate, equivalent to Greenwich mean time).  <i>HH:MM</i> Time difference between local time and UTC in hours ( <i>HH</i> ) and minutes ( <i>MM</i> ).  <i>hh:mm</i> Time shift in hours ( <i>hh</i> ) and minutes ( <i>mm</i> ) between summer time and normal time in local time zone.
Information about application, system and communication partner (only if KCAPPL=Y)			

Field name COBOL	Field name C/C++	Length in bytes	Description	
KCAPPLNM	applnm	8	UTM application name	
KCHOSTNM	hostm	8	Name of host, where the application is running	
KCPTRMNM	ptrmnm	8	In a dialog service: with distributed processing via LU6.1: CON name of the communication partner; with distributed processing via OSI TP: OSI-CON name of the communication partner; otherwise: PTERM name of the communication partner. In an asynchronous service: blanks	
X/W	KCPRONM	pronm	8	blanks
			8	In a dialog service: Name of processor of communication partner In an asynchronous service: blanks
B B B	KCBCAPNM	bcapnm	8	In a dialog service: BCAM application name under which the communication partner signed on In an asynchronous service: blanks
KCVERS	version	6	openUTM version in form <i>Vnn.nx</i> (e.g. V06.1A)	
KCIVER	iversion	2	Version number of KDCS interface, indicates the state of the available function upgrades; in openUTM V6.1A it has the value 8	
KCIVAR	ivariant	1	Identifier of product variant of openUTM: 'B' for BS2000 or 'X' for Unix systems or 'N' for Windows systems	
		1	Blank	
Information about the location of the user ID which started the service (only if KCLOCALE=Y)				
KCUSLANG	us_lang_id	2	Language identifier of the user; if no user is signed on yet: language identifier of the LTERM partner	
KCUSTERR	us_terr_id	2	Territory identifier of the user; if no user is signed on yet: territory identifier of the LTERM partner	
X/W	KCUSNLSL	us_nlslang	16	\$LANG variable of the user
X/W			10	Blanks

	Field name COBOL	Field name C/C++	Length in bytes	Description
B B B	KCUSCCSN	us_ccsname	8	Character set of the user; if no user is signed on yet: character set of the LTERM partner
B			8	Blanks
B B	KCCSCURR	curr_ccs	8	Character set of the message received from terminal (CCSN of the character set active at the terminal)
B B B	KCDEVCAP	dev_cap	1	Information on whether the terminal is a 7- or 8-bit terminal. The specification is in the form "7" or "8" (printable)
B			1	Blanks
OSI TP information (only with KCOSITP=Y)				
	KCFUPOL	fupol	1	Displays whether the functional unit "Polarized Control" is selected (Y/N).
	KCFUHS	fuhsh	1	Displays whether the functional unit "Handshake" is selected (Y/N).
	KCFUCOM	fucom	1	Displays whether the functional unit "Commit" is selected (Y/N).
	KCFUCHN	fuchn	1	Displays whether the functional unit "Chained Trans- actions" is selected (Y/N).

Field name COBOL	Field name C/C++	Length in bytes	Description
KCENDTA	endta	1	<p>This field indicates whether an end of transaction request is permitted at the end of the current processing step and, if this is the case, which calls you have to use.</p> <p>If messages are sent to job-receiving services only in this processing step, the transaction can remain open after the end of the processing step.</p> <p>Blank</p> <ul style="list-style-type: none"> <li>no instruction for the termination of the processing step</li> <li>O no end of transaction may be requested at the end of the processing step.</li> <li>R the transaction and the dialog step must be finished, the service may not be terminated.</li> <li>S the transaction must be finished, the dialog step may not be terminated.</li> <li>C the transaction must be finished, the service may not be terminated.</li> <li>F the transaction must be finished and the service terminated.</li> </ul>
KCSEND	send	1	<p>This field indicates whether a message may be sent to the job-submitting service in the processing step.</p> <ul style="list-style-type: none"> <li>Y You have to send a message to the job submitter at the end of the dialog step. If KCENDTA is set to "S", you also have to send a message to the job submitter at the end of the transaction.</li> <li>N You cannot send an MPUT to the job submitter; you may send messages to job-receiving services. However, in this case the transaction has to remain open after the processing step.</li> </ul>
		2	Blanks
Encryption information (only with KCENCR=Y)			



Field name COBOL	Field name C/C++	Length in bytes	Description
KCPTERM	pterm_enclev	1	Generated minimum encryption level of the client in the associated PTERM or TPOOL statement: N No minimum encryption level was generated for the client. 1 The minimum encryption level 1 was generated for the client. 2 The minimum encryption level 2 was generated for the client. 3 The minimum encryption level 3 was generated for the client. 4 The minimum encryption level 4 was generated for the client. T The minimum encryption level TRUSTED was generated for the client, i.e. the client is trustworthy.
KCCLIENT	client_enclev	1	Maximum encryption mechanism supported by the client: N The client does not support any encryption mechanism. 1 The maximum encryption mechanism supported is level 1. 2 The maximum encryption mechanism supported is level 2. 3 The maximum encryption mechanism supported is level 3. 4 The maximum encryption mechanism supported is level 4.
KCSESS	session_ enclev	1	Encryption mechanism defined for the current session between the client and server: N No encryption mechanism was defined. 1 An encryption mechanism of level 1 was defined. 2 An encryption mechanism of level 2 was defined. 3 An encryption mechanism of level 3 was defined. 4 An encryption mechanism of level 4 was defined.

Field name COBOL	Field name C/C++	Length in bytes	Description
KCCNVTAC	convtac_ enclev	1	Generated minimum encryption level of the TACs with which the service was started. There is an entry in this field even in the case of asynchronous services:  N No minimum encryption level has been generated for the TAC.  1 The minimum encryption level 1 has been generated for the TAC.  2 The minimum encryption level 2 has been generated for the TAC.
KCCONV	conv_enclev	1	This field indicates whether encryption has been defined for the service. This value is selected by the UPIC partner or by the encryption level of the transaction code of the service, not that of the session. This field is also supplied with a value in the asynchronous service:  N No encryption was defined for the service.  1 An encryption mechanism of level 1 was defined for the service.  2 An encryption mechanism of level 2 was defined for the service.  3 An encryption mechanism of level 3 was defined for the service.  4 An encryption mechanism of level 4 was defined for the service.
KCINPMSG	inputmsg_ enclev	1	This field indicates whether or not the dialog input message was encrypted by the client:  N The input message was not encrypted.  1 The input message was encrypted with an encryption mechanism of level 1.  2 The input message was encrypted with an encryption mechanism of level 2.  3 The input message was encrypted with an encryption mechanism of level 3.  4 The input message was encrypted with an encryption mechanism of level 4.
		2	Reserved for subsequent extensions.

Field name COBOL	Field name C/C++	Length in bytes	Description
Various information about the user who started the service (KCBENID, kcuserid), the LTERM from which the service was started (KCLOGTER, kclogter) and the application (only for KCMISC=Y)			
KCUMSGS	amsgs_user	10	Number of queued messages in the user's queue.
KCPWVMAX	pw_val_max	2	Number of days for which the user's password is still valid. Values with a special significance: 0 The password will become invalid within the next 24 hours. -1 The password was generated without any limitation to validity. The password has unrestricted validity. -2 The password's period of validity has expired. This value can only occur in an asynchronous service or during sign-on. -3 The complexity or minimum length of the password has been increased and the password transferred with the KDCUPD tool may not satisfy the requirements for the generated complexity level or may be too short. This value can only occur in an asynchronous service or during sign-on.
KCPWVMIN	pw_val_min	2	Number of days during which the user's password can only be modified at the administrative level but, for example, not via a SIGN CP call (minimum period of password validity). Values with a special significance: 0 The password may be modified. -1 No password may be set for the user. This may occur in the following cases: - Applications without users - The user is an LU6.1-Session or OSI TP association - Connection user (only for TS or UPIC clients) - internal user KDCMSGUS - Users generated with Kerberos certification or certificate

B  
B

Field name COBOL	Field name C/C++	Length in bytes	Description
KCLSTSGN	last_sign	14	Date and time of the last sign-on. The date and time are specified in the form YYYYMMDDHHMMSS. The following cases return zeros: - During the sign-on operation, after the first successful sign on after regeneration - Internal user KDCMSGUS - User is a LU6.1 session or OSI TP association
KCBNDLMS	bundle_ master	8	Name of the master of the LTERM-/LPAP bundle if the LTERM-/(OSI-)LPAP is a slave of this bundle.
KCISGRMS	is_group_ master	1	The field indicates whether the LTERM (or the master of the LTERM bundle) is the master of an LTERM group.  Y (Master) LTERM is a group master. N (Master) LTERM is not a group master.
KCLTCP	lterm_client_ prot	1	The field indicates the LTERM's client protocol  0 The service is running for the internal LTERM KDCMSGT (only in an asynchronous service) 1 LU6.1 2 OSI TP 3 UPIC 4 Platform-dependent: DTP (Unix systems and Windows systems) TIAM (BS2000/OSD) 5 APPLI 6 SOCKET
KCAPPLST	application_ state	1	The field indicates the status of the application: N The application is running normally. G The application has the "Graceful Shutdown" status. W The application has the "Shutdown Warn" status. S The application is being terminated normally. T The application is being terminated abnormally.

	Field name COBOL	Field name C/C++	Length in bytes	Description
	KCKRBCAP	kerberos_ capability	1	In a dialog service, the field indicates whether the client is Kerberos-capable. Y The client is Kerberos-capable. N The client is not Kerberos-capable. In an asynchronous service: Blank.
B	KCCDINFO	info_cd_ available	1	The following INFO CD information is available in a dialog service: C Card information (magnetic stripe card) K Kerberos information E Kerberos information, but the Kerberos dialog returned an error or the information could not be stored at its full length because it is longer than the value generated with MAX PRINCIPAL-LTH. N No INFO CD information INFO CD is not permitted in an asynchronous service: Blank.
B				
B				
B				
B				
B				
B				
B				
B				
B				
X/W			2	Blank
			23	Reserved for extensions.

## LPUT Write to log file

You use the LPUT (log file PUT) call to write a record to the user log file. UTM prefixes this record with the current contents of the KB header. The maximum length of a record is defined at generation (MAX statement, LPUTLTH operand). The records in the log file are not in exactly the same sequence as the LPUT calls in the application.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area		
	KCOP	KCLA	Message area
Write to log file	"LPUT"	Length	Data to be logged

### Setting the 2nd parameter

Here you enter the address of the message area from which openUTM is to read the data to be logged.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"LPUT"
2.	KCLA	Length in bytes
3.	Message area	Data

**KDCS call**

	1st parameter	2nd parameter
4.	KDCS parameter area	Message area

5. **C/C++ macro call**

	Macro name	Parameters
	KDCS_LPUT	(nb,kcla)

**openUTM return information**

	Field name in the KB return area	Contents
6.	KCRCCC	Return code
7.	KCRCDC	Internal return code

For the LPUT call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the LPUT operation code.
2. In the **KCLA** field, the length of the data to be transferred in bytes. openUTM prefixes the length to the data record.

In the message area, you specify:

3. the data which you want to write to the user log file.

You specify the following for the KDCS call:

4. 1st parameter: the address of the KDCS parameter area.  
2nd parameter: the address of the message area from which UTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLM.
5. The use of C/C++ macro calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

6. in the **KCRCCC** field, the KDCS return code, see next section.
7. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### **KDCS return code in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 01Z Value specified in KCLA too large. It is shortened to the generated maximum length (LPUTLTH operand in the MAX statement).
- 40Z System cannot perform the operation (generation error or system error, timeout), see KCRCDC.
- 43Z Length entry in KCLA is invalid (e.g. negative).
- 47Z Message area missing or cannot be accessed in the specified length.

An additional error code can be found in the dump:

- 71Z INIT call missing in this program.

### **Features of the LPUT call**

- You can still reset the LPUT operation prior to the next synchronization point.
- The PEND ER/FR/RS and RSET calls reset the LPUT operation.

The structure of the user log file is described on [page 86](#).



## MCOM Define job complex

You use the MCOM (message complex) call to

- define the beginning of a job complex and set the destinations of the basic job and the associated confirmation jobs, or
- define the end of a job complex.

### Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area					
	KCOP	KCOM	KCRN	KCPOS	KCNEG	KCCOMID
Beginning of job complex	"MCOM"	"BC"	LTERM/ TAC/ Service ID/ TAC queue	TAC/ blanks, TAC queue	TAC/ blanks, TAC queue	Complex ID
End of job complex	"MCOM"	"EC"	Binary zero	Binary zero	Binary zero	Complex ID

All the fields of the parameter area which are not used have to be set with binary zero.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"MCOM"
2.	KCOM	"BC"/"EC"
3.	KCRN	LTERM name/TAC/service ID// binary zero/TAC queue
4.	KCPOS	TAC/blanks/binary zero/TAC queue
5.	KCNEG	TAC/blanks/binary zero/TAC queue
6.	KCCOMID	Complex ID

**KDCS call**

	1st parameter	2nd parameter
7.	KDCS parameter area	-

8. **C/C++ macro calls**

Macro name	Parameters
KDCS_MCOMBC	(kcrn,kcpos,kcneg,kccomid)
KDCS_MCOMECC	(kccomid)

**openUTM return information**

	Field name in the KB return area	Contents
9.	KCRCCC	Return code
10.	KCRCDC	Internal return code

For the MCOM call you make the following entries in the appropriate fields of the KDCS parameter area:

1. In the **KCOP** field, the MCOM operation code.
2. In the **KCOM** field, either "BC" for beginning or "EC" for end of a job complex.
3. In the **KCRN** field, if KCOM = BC applies:
  - the LTERM name of a communication partner if the basic job is an output job,
  - the TAC of an asynchronous program if the basic job is a background job (without distributed processing), or
  - the name of the TAC queue when the basic job is an output job in a TAC queue (without distributed processing).
  - the service ID of a job-receiving service if the basic job is directed to a job-receiving service.

If KCOM = EC applies, you have to enter binary zero.

4. In the **KCPOS** field, if KCOM = BC is specified as the destination of the positive confirmation job, the TAC of an asynchronous program or a TAC queue, or blanks if no positive confirmation job is to be generated.

If KCOM = EC applies, you enter binary zero.

5. In the **KCNEG** field, if KCOM = BC is specified as the destination of the negative confirmation job, the TAC of an asynchronous program or a TAC queue, or blanks if no negative confirmation job is to be generated.

If KCOM = EC applies, you enter binary zero.

6. In the **KCCOMID** field, the complex identifier (complex ID) of the job complex. It is defined for MCOM BC, may be 2 to 8 characters long and has to be prefixed with the character "\*". It is to be specified for all the DPUT calls of the complex and for MCOM EC.

You specify the following for the KDCS call:

7. 1st parameter: the address of the KDCS parameter area.
8. The use of C/C++ macro calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

9. in the **KCRCCC** field, the KDCS return code, see next page.
10. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").

**KDCS return codes for the MCOM call**

The following codes can be analyzed in the program:

- 000 Function carried out.
- 40Z openUTM cannot perform the function: generation error or system error, or a job complex is to be started without previously terminating the preceding job complex.
- 42Z Entry in KCOM is invalid.
- 44Z Value in KCRN is invalid:
  - no TAC of an asynchronous program or a TAC queue specified, or TAC or TAC queue inhibited/illegal
  - no LTERM name specified
  - no valid service ID specified, or the service ID is already occupied by another message.
  - asynchronous messages for the dead letter queue (KDCDLETQ) have been created.
- 49Z Contents of unused fields of the KDCS parameter area not equal to binary zero.
- 51Z For KCOM = EC: no (confirmation) job to which user information can be related.
- 55Z Entry in KCCOMID is invalid: the name does not begin with "\*" or is already assigned in the program unit or is unknown (for MCOM EC).
- 57Z Value in KCPOS is invalid:
  - no TAC of an asynchronous program or a TAC queue specified, or TAC or TAC queue inhibited/illegal
  - specification is not equal to blanks.
  - acknowledgement jobs with the destination dead letter queue (KDCDLETQ) have been created.
- 58Z Value in KCNEG is invalid:
  - no TAC of an asynchronous program or a TAC queue specified, or TAC or TAC queue inhibited/illegal, or
  - specification is not equal to blanks.
  - acknowledgement jobs with the destination dead letter queue (KDCDLETQ) have been created.

An additional error code can be found in the dump:

- 71Z INIT missing in this program unit.

**Features of the MCOM call**

- You have to terminate a job complex with MCOM EC before the PEND call, otherwise openUTM aborts the service with PEND ER and 86Z.
- The complex identifier has to be unique within a program unit.
- MCOM BC is only allowed after all preceding job complexes have been terminated.
- If a service identifier is occupied by a job complex, it can only be released by MCOM EC (not as otherwise by DPUT NE).
- Confirmation jobs must be directed to asynchronous program units or TAC queues of the local application.
- If an error occurs, a message complex's main job with negative acknowledgment job is (possibly after redelivery) not saved in the dead letter queue but is deleted. The negative acknowledgment job is activated.

## MGET Receive dialog message

You use the MGET (message GET) call to read messages into the message area in a program unit run of a dialog service. In an asynchronous service, the MGET call is only permitted in follow-up program units.

Messages may have the following origins:

- a terminal
- another application (via LU6.1 or OSI TP)
- a transport system application of the type APPLI
- a socket application
- a UPIC client program
- a previous program unit run of a dialog or asynchronous service of the same service

With message segments, a separate MGET call is necessary for each message segment.

In the case of socket partners, a message segment can be read with several MGET calls. Using KCRLM and the return code it is possible to determine whether a message segment has been read in its entirety.

If the message originates from an OSI partner, it can be a message segment, an error message, a handshake request or a handshake confirmation.

If a function key was pressed, two MGET calls are required: the first provides the return code, the second the data.

Function keys are not supported in the job-receiving service when using distributed processing.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area				
	KCOP	KCOM	KCLA	KCRN	KCMF/kcfn
Message in format mode	"MGET"	—	Length	-	Format identifier
Message in line mode	"MGET"	—	Length	-	Blanks  also possible in BS2000/OSD: edit profile
Message from previous program unit of the same application	"MGET"	—	Length	—	Blanks
Reset message from a program unit	"MGET"	"NT"	Length	Reset-ID	Blanks
Dialog message from job-submitting service	"MGET"	—	Length	—	Format identifier/ Blanks/ Name of abstract syntax
Dialog message from job-receiving service	"MGET"	"NT"	Length	Service ID	Format identifier/ Blanks/ Name of abstract syntax
Status information from job-receiving service	"MGET"	"NT"	0	Service ID	Blanks

B  
B  
B

NT message segment

### Setting the 2nd parameter

Here you have to supply the address of the message area into which openUTM is to read the message.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"MGET"
2.	KCOM	"NT"/-
3.	KCLA	Length in bytes
4.	KCRN	Service ID/Reset ID/ -
5.	KCMF/kcfn	Format identifier/blanks/ also possible in BS2000/OSD: edit profile

B  
B

**KDCS call**

	1st parameter	2nd parameter
6.	KDCS parameter area	Message area

7. **C/C++ macro calls**

Macro name	Parameters
KDCS_MGET	(nb,kcla,kcfn)
KDCS_MGETNT	(nb,kcla,kcrn,kcfn)



openUTM return information

	Message area	Contents
8.	<input type="text"/> Field name in KB return area	Data
9.	KCRDF	Screen function/0
10.	KCRLM	Actual length
11.	KCRMGT	Type of message
12.	KCVGST/kcpcv_state	Service status
13.	KCTAST/kcpta_state	Transaction status
14.	KCRCCC	Return code
15.	KCRCDC	Internal return code
16.	KCRMF	Format identifier/ blanks/ - also possible in BS2000/OSD: edit profile
17.	KCRPI	Service ID/ Blanks

B  
B

For the MGET call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, enter the FPUT operation code.
2. In the **KCOM** field,  
(need only be specified for messages from a job receiver and for reset messages)  
"NT" modifier (message part).
3. In the **KCLA** field, specify the length in which the message is to be read. This length must not exceed the message area into which the message is to be read (length zero means no message is received. Any existing messages are lost). The actual length of the message (segment) is returned in the KCRLM field.
4. In the **KCRN** field,  
(need only be specified for messages from a job receiver and for reset messages)
  - for messages from job-receiving services, the service ID of the job-receiving service
  - for reset messages, the reset ID of the reset message.



For the first MGET call of a program unit you can ascertain the reset ID or the service ID from the KCRPI field of the INIT call, for subsequent MGET calls from the KCRPI field of the previous MGET call.

5. In the **KCMF/kcfn field**  
(irrelevant for messages from previous program unit runs of the same service)



The entry in this field will always be correct if, for the first MGET call of the program unit, you enter the value returned by the INIT call in the KCRMF/kcrfn field. For subsequent MGET calls, use the value from the KCRMF/kcrfn field of the preceding MGET call.

- for messages in format mode:  
format identifier
- for messages in line mode:  
blanks  
also possible in BS2000/OSD: Edit profile
- when reading reset messages:  
blanks
- Message from a UPIC client program:  
format identifier that the UPIC client program specified for sending (starting with Version 4.0, openUTM client (UPIC) supports the sending of format identifiers)
- with distributed processing via LU6.1:  
format identifier specified by the partner application in KCMF/kcfn when issuing the MPUT call
- for messages from OSI TP partners:  
Name of the abstract syntax of the message. This name was returned in the KCRMF/kcrfn field in the preceding INIT or MGET call.  
Here, blanks represent the abstract UDT syntax. In this case, BER is used as transfer syntax and openUTM decodes the message.  
If you enter a value other than blanks, openUTM transfers the message to the program unit in encoded format (i.e. in the transfer syntax corresponding to this abstract syntax) and the program unit itself must convert the message into the local representation. This is possible, for example, using an ASN.1 compiler.

You specify the following for the KDCS call:

6. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area into which openUTM is to read the message. You must enter the address of the message area even if you have entered the length 0 in KCLM.
7. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

8. in the specified message area, the message (segment) in its desired length, if **KCRCCC=03Z** was not set. If the message was longer than specified in **KCLA**, the remainder will be lost.  
Exception: the message originates from a socket client. In this case, the return code **02Z** is set, and the rest of the message (segment) can be obtained with the next **MGET**.
9. in the **KCRDF** field, for the first **MGET** of a job-receiving service with which communication was performed using the LU6.1 protocol, the value in the **KCDF** field of the associated **MPUT** call in the job-receiving service. In all other cases this field has the value zero.
10. in the **KCRLM** field, the actual length of the message (segment). If **KCRLM > KCLA**, the message has been truncated (**KCRCCC = 01Z**). In the case of a socket client, the rest of the message can still be read (**KCRCCC = 02Z**); otherwise, the rest of the message is lost (**KCRCCC = 01Z**).  
**KCRLM=0** for empty messages and if **KCRCCC=03Z**.

B  
B

When using **FHS**, the value returned in **KCRLM** depends on the **FHS** start parameter **KCRLM=**.

11. in the **KCRMGT** field, the data read by **MGET**:

"M" ("message)  
a message. When using **MGET** for a partner that does not communicate via the OSI TP protocol, only the value "M" can be specified.

When using **MGET** for a partner that communicates via the OSI TP protocol the following output is also possible:

"C" (confirm)  
a positive handshake confirmation.

"E" (error)  
an error message or negative handshake confirmation.

"H" (handshake)  
a handshake request.

12. in the **KCVGST/kcpcv\_state** field, the service state of the (partner) service, see [page 340](#).
13. in the **KCTAST/kcpta\_state** field, the transaction state of the partner service, see [page 341](#).
14. in the **KCRCCC** field, the KDCS return code, see [page 343](#).
15. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").

16. in the **KCRMF/kcrfn** field

- after reading an entire format: identifier of this format. This is always identical to the identifier of the last output format.
- after reading a partial format: identifier of the next partial format with input data. If there is no further partial format with input data, KCRMF/kcrfn contains the identifier of the most recently read partial format. In this case KCRMF=KCMF (kcrfn=kcfn).
- for a message in line mode: blanks or name of the edit profile
- for a message from a partner service: KCRMF/kcrfn contains the format identifier or the name of the abstract syntax of the next message (segment) which can be read by the service defined in the KCRPI field. After the MGET call for the last message (segment), this field contains the format identifier of the last message (segment).

B

17. in the **KCRPI** field

- for a message from a job-receiving service: service ID of a job-receiving service for which message segments or status information is present which has not yet been read.
- in all other cases: blanks.

**Service status in the KCVGST/kcpcv\_state field**

Entry for dialogs without distributed processing:

"O" (open)  
The local service is open.

Entries with distributed processing after a message from a partner service:

"C" (closed)  
The job-receiving service has terminated (PEND FI).

"D" (disconnected)  
Communication with a job receiver was terminated as a result of a loss of connection (only with OSI TP).

"E" (error)  
Only when communicating via LU6.1:  
The job-receiving service was terminated using PEND ER or PEND FR.

"I" (inactive)  
The job-receiving service is inactive, i.e. it could not be started since, for example, the transaction code was unknown or is locked or no association could be reserved for OSI TP.

- "O" (open)  
The partner service is open, you may send further messages to the partner service.
- "P" (pending end dialogue)  
This status can only occur in the case of heterogeneous links and for dialogs for which the Commit functionality has not been selected:  
The job-receiving service wants to end the communication. If the job-submitting service does not agree, it can continue the service using MPUT EM.
- "R" (reset)  
Only when communicating via LU6.1:  
The job-receiving service was terminated with PEND RS.
- "T" (timeout)  
The job-receiving service has been or is being terminated incorrectly, since no answer has been received from the job-receiving service within the generated wait time or it has not been possible to seize a session within the generated wait time.
- "Z" (error)  
The job-receiving service has been terminated by the system using PEND ER. (e.g. KDCS call in a job-receiving service with error  $\geq 70Z$ ). The Z service status is also set if a program unit run was terminated with PEND ER, FR or RS in an OSI TP job-receiving service.

If you read a message which originated from a job-submitting service, the service status can only have the value "O".

With the service statuses D, E, I, R, T and Z no message is passed, i.e. the KCRLM return length is 0.

### **Transaction status in the KCTAST/kcpta\_state field**

Entries for dialogs without distributed processing:

- "O" (open)  
The transaction in the local service is open.
- "C" (closed)  
Either start of service or after a synchronization point.
- "R" (reset)  
A reset message has been read.

Entries for distributed processing after a message from the partner service:

- "C" (closed)  
Only when communicating via LU6.1:  
The transaction is finished in the partner service. This situation occurs if PEND RE or PEND FI has occurred in the partner service and PEND RE is active in the local service.
- "H" (heuristic hazard)  
Only when communicating using the OSI TP protocol:  
The result of a transaction is undetermined since communication with at least one communication partner has been interrupted. The possibility that one of the communication partners involved in the last transaction has made a heuristic decision which conflicts with the actual result of the last transaction cannot be excluded.
- "I" (inactive)  
A job receiver transaction exists because, for example, the transaction code is unknown or no connection could be reserved during the generated waiting time.
- "M" (mismatch)  
It was not possible to synchronize the transaction in the remote service with the transaction in the local service. This may occur after a timeout or after termination and start of a UTM-F application.  
When communicating via the OSI TP protocol, this situation may occur if at least one of the communication partners involved in the last transaction has made a heuristic decision which conflicts with the actual result of the transaction.
- "O" (open)  
The transaction in the partner service is open.
- "P" (prepare to commit)  
The partner service has either initiated the end of transaction itself or is requesting the local service to initiate the end of transaction.
- "R" (reset)  
The transaction in the partner service has been reset.
- "U" (unknown)  
Only possible when communicating via OSI TP without global transaction logging.  
The transaction status is unknown.

When you read a message from a job-submitting service, only the values "C", "O", "P" or "U" can occur.

**KDCS return codes in the KCRCCC field**

- 000 Operation carried out.  
The message (segment) was read in its entirety.
- 01Z Length conflict: KCLA < KCRLM. The message area is too short, the message (segment) has been truncated.
- 02Z In the case of messages from a socket client:  
Length conflict KCLA < KCRLM. The message area is too small; the part of the message segment that was truncated can be read by means of another MGET call.
- 03Z For partial formats:  
KCMF/kcfn does not contain the name of the next returned partial format.  
  
For distributed processing and messages from UPIC clients:  
KCMF/kcfn does not contain the format identifier or the name of the abstract syntax of the message (segment) which is to be read next.  
  
No message (segment) is transferred to the message area; the KCRPI and KCRMF/kcfn fields contain a new proposition for the next message (segment).
- 05Z With individual formats:  
The format displayed on screen differed from the format specified in KCMF/kcfn. The message was formatted as per the format identifier of the most recent display and not as specified in KCMF/kcfn.  
  
In line mode:  
The first character in KCMF/kcfn is not a blank or the name of the edit profile is invalid.
- B**
- 10Z Message has already been completely read
- 12Z (Only possible in the job-submitting service)  
No (more) message (segments) are present from the specified service ID. However, message (segments) are still present from other job-submitting services. The content of the message area has not been changed. The KCRPI and KCMF/kcfn fields contain a new proposition about which message (segment) can be read next.
- 19Z The function key has not been generated or the allocated special function is invalid.
- 20Z...39Z  
The terminal user has pressed a function key to which a return code was assigned during generation,  
**B**  
or KDCS<sub>xx</sub> (01 ≤ xx ≤ 20) was entered (function key simulation).  
  
If a function key is triggered by an UPIC partner or a function key to which a message has been assigned is pressed then this message must be read with a subsequent MGET call.

Additional return codes can be found in the dump:

- 70Z Operation cannot be performed by the system (system or generation error); see KCR CDC.
- 71Z An MGET call was entered in the first processing step of the first program unit run of an asynchronous service, or INIT is missing in this program unit run.
- 72Z Specification in KCOM is invalid.
- 73Z Length entry in KCLA is invalid.
- 77Z The message area is missing or cannot be accessed in the specified length.
- B** 78Z The FORMAT event exit reports an error.



## Features of the MGET call

- Reaction to length conflicts

The actual message (segment) length is returned in the KCRLM field.

When length conflicts occur, note the following: If  $KCRLM < KCLA$  only  $KCRLM$  characters (bytes) are moved to the message area. The contents of the rest of the message area are undefined. Only one message (possibly consisting of two or more message segments) can be read in a program unit. If the length entry in  $KCLA$  of the parameter area is shorter than the actual message (segment), the remainder ( $KCRLM - KCLA$ ) is lost and can no longer be read with a subsequent MGET.

Exception: if, in the case of messages from a socket client, the message area is too small ( $KCLA < KCRLM$ ), the rest of the truncated message segment can be read by means of another MGET call.

### *Example*

Three message segments, each of 100 bytes, are to be read using MGET calls. The table below shows the effect of different specifications in the  $KCLA$  field.

User specification		UTM returns			Explanations
KCOP	KCLA	KCRLM	Transferred length in MA	KCRCCC	
MGET	100	100	100	000	The message segment was received successfully
MGET	50	100	50	01Z	The message segment was longer than specified in $KCLA$ , the remainder is lost.
MGET	150	100	100	000	The message segment was received successfully
MGET	100	000	000	10Z	No fourth message segment was present

- Conversion of lower case letters

openUTM does not automatically convert lowercase letters into uppercase letters when MGET is called. However, you can perform this conversion by using the appropriate format generation.

**B**

In BS2000/OSD you can also perform a conversion by using edit profiles.

- Messages of length zero

Messages of length zero are, for example, possible in the following cases:

- Only the transaction code (without any further data) was sent at the start of a service
- In a follow-up program unit a message is to be read from the preceding program unit of the same service and MPUT was specified with length 0 in the previous program unit or no MPUT was issued.
- A client program or a partner service has sent an empty message.
- The terminal user pressed a function key without assigning a message, or KDCS<sub>xx</sub> ( $01 \leq xx \leq 20$ ) was entered (function key simulation).
- The terminal user has sent an empty message (DUE function with empty screen)

B

B

- Removing a transaction code

If a transaction code was specified together with a message at service start and no function key was used, the following is removed from the message

- the transaction code, including trailing blanks, if the entry was unformatted (MAX statement parameter LEADING-SPACES)
- the first 8 characters of the message (transaction code) if the entry was formatted with \*formats
- the first 10 characters (attribute field plus TAC) if the entry was formatted with +formats
- the first 8 characters of the message (transaction code) if the entry was formatted with +formats
- the first eight (with \*formats) or ten (with +formats) characters of the first message if the entry was formatted with partial formats.

B

B

B

B

B

B

B

B

The removal of the transaction code can be prevented in an INPUT exit.

If the MGET call is used in the BADTACS event service the invalid transaction code is **not** removed from the input message. The entire message is made available. This also applies if the invalid transaction code is allocated to a function key.

- Receiving partial formats

Every partial format must be read with a separate MGET.

Following INIT, openUTM supplies, in KCRMF/kcrfn, the name of the first returned partial format in which data were entered. This name must be specified in KCMF/kcfn when issuing MGET. MGET supplies, in KCRMF/kcrfn, the name of the next partial format together with input data, which must be specified in KCMF/kcfn with the following MGET. With the final partial format with input data, KCRMF/kcrfn again has the name specified in KCMF/kcfn, see also example. You recognize the last partial format by the identical entries in KCMF/kcfn and KCRMF/kcrfn or by the return code 10Z in the next MGET.

If no data was input in any of the partial formats, the first MGET call supplies KCRCCC=10Z, KCRLM=0, KCRMF=blanks in the return area.

If the name entered in KCMF/kcfn differs from the one previously supplied in KCRMF/kcrfn then

- openUTM does not write data to the message area
- openUTM sets KCRLM=0
- openUTM sets the return code 03Z in KCRCCC
- openUTM writes the 'correct' format name again in KCRMF/kcrfn.

B  
B  
B  
B

Note that the way in which partial formats are transferred also depends on the FHS start parameters, see FHS manual. If, for instance, no entry is made in any partial format, some FHS start parameters cause openUTM to return KCRCCC=10Z and KCRMF/kcrfn=blanks after the first MGET.

*Example*

The three partial formats \*PART1, \*PART2 and \*PART3 are to be read using MGET calls; note the return information in KCMF.

User entries		UTM return info		Explanations
KCOP	KCMF/kcfn	KCRMF/kcrfn	KCRCCC	
INIT		*PART1	000	
MGET	*PART1	*PART2	000	Read 1st partial format
MGET	*PART2	*PART3	000	Read 2nd partial format
MGET	*PART3	*PART3	000	Read 3rd partial format
MGET	*PART3	*PART3	10Z	Message already completely read

You recognize the last partial format by the identical entries in KCMF/kcfn and KCRMF/kcrfn or by the return code 10Z in the next MGET.

### Features of the MGET call with distributed processing

- When communicating via the OSI TP protocol, the format identifier for the transfer of the name of the abstract syntax is used. Here, blanks represent the abstract syntax of UDT. In all cases in which the application program unit does not work with UDT, the conversion of the message from the local representation to transfer syntax or vice versa must be performed by the application itself - in accordance with the rules of the abstract syntax. This process is termed the encoding or decoding of a message. To do this, the application can use an ASN1 compiler.

openUTM carries out the encoding and decoding of messages in UDT format.

- When communicating via the LU6.1 protocol, openUTM transfers the format identifier. However, it does not format the message: the partner applications exchange only net data. When using MPUT, you can specify any name in the KCMF/kcfn field. This name is indicated to the reading program unit in the KCRMF/kcrfn field after INIT or following a preceding MGET and must be specified in the KCMF/kcfn field when calling MGET.
- The return codes for the function keys (19Z through 39Z) cannot occur with an MGET call in the job-receiving service, because the job-submitting service cannot forward any corresponding special functions to the job-receiving service.
- In the KDCS return area the MGET call provides the service status and the transaction status of the partner service.
- If you do not adhere to the bottom-up strategy (see page 137) when communicating via LU6.1, a service restart can nevertheless be initiated by sending a message from the job-submitting service to the job-receiving service. Then the follow-up program unit in the job-receiving service is started. This uses MGET to read the message from the last synchronization point and receives the service status "O" and the transaction status "C" from the job submitter. After the INIT call it recognizes from the service indicator KCKNZVG/kccv\_status in the KB header that this is a service restart.

### Particularities of the MGET call when communicating with a UPIC client program

openUTM transfers the format identifier during communication, but it does not format the message: only net data is exchanged between the UPIC client program and the application. When a message is sent, any name can be specified as the format identifier. This name is displayed in the reading program after the INIT or after a preceding MGET in the KCRMF/kcrfn field and must be specified for an MGET call in the KCRMF/kcfn field.

### **Particularities of the MGET call when communicating with a socket partner**

A message segment from a socket partner can be read by means of several MGET calls. Using the return code it is possible to determine whether a message (segment) has been read. The return code 02Z indicates that a message segment has not yet been read in its entirety. By comparing KCLA and KCRLM you can determine how large the rest of the message segment is. The return code 000 indicates that the message (segment) has been read in its entirety and that the next MGET will read a new message (segment).

### **Features of a reset message**

A reset message always originates from a program unit terminated with PEND RS. Following service restart, it is supplied to the program unit which was started as a follow-up program unit after the synchronization point. The reset message must be read with the first MGET call. It enables the program unit to react appropriately and thus avoids repeated resetting of the transaction. The program unit recognizes a service restart from the service indicator which takes the value "R". It is available in the KCKNZVG/kccv\_status field after the INIT call. The reset message is deleted after processing in the case of a loss of connection or KDCOFF.

With the MPUT call, you specify in the KDCS parameter area, whether (KCDF = KCRESTRT) or not (KCDF contains binary zero) a screen restart is performed when the service is restarted. If no screen restart is requested, openUTM resets the transaction and immediately starts the program unit run specified at the last synchronization point. The reset message can be read with MGET.

### **MGET call for reading status information from the job receiver**

Status information takes the form of messages of length 0 which are internally created by openUTM. Their only purpose is to indicate the status of the job-receiving service when errors occur in distributed processing. The status information is read unformatted using MGET calls (blanks in KCMF/kcfn).

If it was necessary to reset the distributed transaction then, when the service restarts, it is advisable to restart the program unit for which a message was present at the end of the last distributed transaction, or for which the next input from the terminal is intended. If a program unit is started in the job-submitting service, openUTM sends status information to this program, if necessary. Here you have to consider the following:

- The status information relates to the job receivers which caused the distributed transaction to be reset and were or are subsequently being terminated.
- If the input message at service restart is for the program unit, then the input message has to be read with the 1st MGET and the status information with the second MGET. If, however, the input message originates from the job-receiving service and is not sent by the job receiver, then you only receive the status information of the job receiving service.

- If the input message at service restart is for the job-receiving service and if this service cannot be started within a generated wait time (e.g. because of loss of connection), openUTM starts the follow-up program unit in the job-submitting service instead and sends the job-submitting service status information which has to be read with the first MGET.
- If, after service restart of the job-submitting service, a job-receiving service is addressed again and if an error reoccurs, then the job-submitting service can be reset several times to the same synchronization point. Since the status information is retained from the preceding reset, you may receive two or more bits of status information. In this case you obtain with each MGET the service ID of a subsequent service for which there is status information.
- There may be status information available from different job-receiving services. This status information has to be read in the order suggested by openUTM (KCRPI).
- "Substitute messages" are received with distributed processing via OSI TP, even if there is no service restart.  
If global transaction logging is not active, the transaction in the job-submitting service is **not** reset if an error occurs in the job-receiving service (e.g. timeout).  
If global transaction logging is active, the transaction in the job-submitting service is not reset only if no distributed transaction with the job receiver is open, e.g. if the timer for the association allocation has expired.

#### **B Special KDCS functions:**

- B** The KDCS interface provides "special KDCS functions" as a particular way of entering data at the terminal. The terminal user activates these functions by entering the string
- B**
- B** (KDCSxx) xx= 01,...,20
- B**
- B** when UTM expects to receive input for a follow-up program unit run. A maximum of
- B** 20 special KDCS functions are therefore possible. The special KDCS functions are
- B** intended to be replacement inputs for terminals that do not have the appropriate keys.

## MPUT Send dialog message

You use the MPUT (message put) call:

- to send a dialog message (segment) to a client
- to send a message (segment) to a subsequent program unit in the same dialog step or in a linked service
- to send a reset message for the service restart after PEND RS
- to send the last screen output of a stacked service to the terminal
- with distributed processing, in the job-submitting service, to send a message (segment) to a job-receiving service, or
- with distributed processing, in the job-receiving service, to send a message (segment) to the job-submitting service
- to request a processing confirmation from an OSI TP partner
- to send a negative processing confirmation or an error message to an OSI TP partner.
- to create an error message that is sent to a UPIC client or a socket application in the event of abnormal termination of a service (system PEND ER) initiated by openUTM

In an asynchronous service, an MPUT message can only be sent to a job-receiving service or to a follow-up program unit.

The call cannot be used in an MSGTAC program.

## Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area.

X/W  
X/W  
B  
B

Function of the call	Entries in the KDCS parameter area					
	KCOP	KCOM	KCLM	KCRN	KCMF/kcfn	KCDF
Message in format mode	"MPUT"	"NT"/ "NE"	Length	Blanks	Format identifier	Screen function
Message in line mode	"MPUT"	"NT"/ "NE"	Length	Blanks	Blanks	—
Message in line mode	"MPUT"	"NT"/ "NE"	Length	Blanks	Blanks/ edit profile	Screen function
Message to program unit	"MPUT"	"NT"/ "NE"	Length	TAC	—	—
Last screen output of stacked service	"MPUT"	"PM"	Length	Blanks	Format identifier/ Blanks	Screen function
Send reset message	"MPUT"	"RM"	Length	Reset ID	binär 0	binary 0/ KCRESTRT
Message to job-receiving conv. (LU6.1)	"MPUT"	"NT"/ "NE"	Length	Service ID	Format identifier/ Blanks	binary 0
Message to job-sub. conv. (LU6.1)	"MPUT"	"NT"/ "NE"	Length	Blanks	Format identifier/ Blanks	any value
Message to job-receiv. service (OSI TP)	"MPUT"	"NT"/ "NE"	Length	Service ID	Blanks/ abstract syntax	0
Message to job-subm. service (OSI TP)	"MPUT"	"NT"/ "NE"	Length	Blanks	Blanks/ abstract syntax	0
Request a dialog confirmation	"MPUT"	"HM"	0	Service ID/ blanks	Blanks	0
Error message or neg. confirmation	"MPUT"	"EM"	0	Service ID/ blanks	Blanks	0
Error message for UPIC client or socket application	"MPUT"	"ES"	Length	Blanks	Blanks/ Format identifier	0

NT message segment

NE last message segment or complete message.

For KCOM = HM/EM/ES/PM/RM you have to set binary zero for all the fields not used in the KDCS parameter area.



**Setting the 2nd parameter**

Here you have to supply the address of the message area from which openUTM is to read the message.

**Setting the parameters**

B  
B

	Field name in the KDCS parameter area	Contents
1.	KCOP	"MPUT"
2.	KCOM	"NT"/"NE"/"PM"/"RM"/"HM"/"EM"/"ES"
3.	KCLM	Length in bytes
4.	KCRN	Blanks/TAC/reset ID/service ID
5.	KCMF/kcfn	Format identifier/blanks/ Name of abstract syntax/ also possible in BS2000/OSD: edit profile
6.	KCDF	Screen function/binary zero
7.	Message area	Data

**KDCS call**

	1st parameter	2nd parameter
8.	KDCS message area	Message area

**C/C++ macro calls**

Macro name	Parameters
KDCS_MPUTNT / KDCS_MPUTNE	(nb,kclm,kcrn,kcfn,kcdf)
KDCS_MPUTPM	(nb,kclm,kcfn,kcdf)
KDCS_MPUTRM	(nb,kclm,kcfn)
KDCS_MPUTHM / KDCS_MPUTEM	(nb,kcrn)
KDCS_MPUTES	(nb,kclm,kcfn)

**openUTM return information**

	Field name in the KB return area	Contents
10.	KCRCCC	Return code
11.	KCRCDC	Internal return code

For the MPUT call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the "MPUT" operation code.
2. In the **KCOM** field:
  - NT for message segment.
  - NE for complete message or final message segment.
  - PM or the last screen output of a stacked service or the request for a service restart in the sign-on service.
  - RM for a reset message.
  - HM for requesting a processing confirmation from OSI partners.
  - EM for an error message or a negative processing confirmation to OSI partners.
  - ES for creating an error message to a UPIC client or a socket application.
3. In the **KCLM** field, the length of the message in the message area which is to be sent (length zero is also permitted).
4. In the **KCRN** field, depending on message recipient:
  - the transaction code of a follow-up program if this MPUT sends a message to a follow-up program in the same application (this also applies for a PEND FC call).
  - blanks if this MPUT sends a dialog message to a client.
  - the reset identification (reset ID) if a reset message is to be sent for the service restart. The reset ID must begin with "<" (see [“Features of a reset message” on page 349](#) in the description of the MGET call).
  - blanks if this MPUT creates an error message for a UPIC client.
  - blanks for a response to the job-submitting service.
  - the service ID of a job-receiving service if this MPUT call is directed to a job-receiving service.
5. In the **KCMF/kcfn** field
  - a format identifier for a message in format mode or for KCOM = PM with KCLM > 0. If a screen is to be refreshed, the specified format must be a component of the screen to be refreshed.
  - blanks for a message in line mode
  - blanks for KCOM = PM with KCLM = 0
  - blanks for messages to UPIC client programs (version < 4.0). As of version 4.0 openUTM-Client (UPIC) supports the transfer of format identifiers.

- Blanks or name of the abstract syntax  
The name of the abstract syntax of the message must be specified in the case of messages to OSI TP partners. Here, blanks represent the abstract syntax of UDT. In this case, the BER transfer syntax is used and the encoding of the message is performed by openUTM.

If you enter a value other than blanks, the message must be transferred to openUTM in encoded format, i.e. in the transfer syntax corresponding to this abstract syntax.

- in all other cases, irrelevant
- B** – edit profile for a message in line mode

6. In the **KCDF** field, a screen function, if the receiver is a terminal.

In the following cases you have to set binary zero in the field:

- KCMF/kcfn contains the name of a #format.
- The message is intended for a job-receiving service.
- The message is intended for an OSI TP job-submitting service.
- MPUT PM with KCLM = 0 is used.

- B** – KCMF/kcfn contains the name of the edit profile.

When sending a reset message (KCOM = RM), you must specify KCRESTRT or binary zero.

You enter in the message area:

7. The message you want to output.

You enter the following for the KDCS call:

8. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area from which openUTM is to read the message (or user information). You enter the address of the message area even if you have entered the length 0 in KCLM.
9. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

10. In the **KCRCCC** field, the KDCS return code.
11. In the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

**KDCS return codes at the MPUT call**

The following codes can be analyzed in the program:

- 000 Function carried out
- 41Z Too many MPUT calls:
  - another MPUT NT/NE after MPUT NE/HM
  - another MPUT after or before an MPUT PM
  - more than one MPUT RM
  - MPUT RM in the first transaction of a service or unpermitted change of entry in KCRN (if there are multiple message segments, the TAC of the follow-up program must always be the same).
  - MPUT HM issued as the first call to an OSI TP partner
  - following a CTRL call, an MPUT HM was issued to the same partner
  - following a CTRL AB call, an MPUT was issued to the same partner
  - only in an OSI TP job-receiving service: an MPUT was issued to the job submitter and KCSEND has the value N.
- 45Z The value specified in KCMF/kcfn is invalid. Possible causes:
  - the specified abstract syntax has not been generated for the OSI-LPAP partner
  - using MPUT no blanks were sent to a UPIC client program (version < 4.0).

Additional error codes can be found in the dump:

- 70Z The system cannot perform the operation (system or generation error), see KCRCDC.
- 71Z MPUT call in an MSGTAC program unit or no INIT in program unit.
- 72Z Entry in KCOM is invalid or MPUT-PM was entered in an asynchronous program unit.
- 73Z Length entry in KCLM is invalid.

- 74Z Value in KCRN is invalid, because
- in a dialog service KCRN contains a value, which is not a TAC of a dialog program unit nor a valid service ID
  - in an asynchronous service KCRN contains a value, which is not a TAC of an asynchronous program unit nor a valid service ID
  - the user is not entitled to use the program unit run
  - in an asynchronous program unit, KCRN is filled with blanks
  - with MPUT HM, the destination in KCRN is not an OSI TP partner
  - with MPUT EM, the destination in KCRN is not an OSI TP partner.
  - with MPUT ES in KCRN, no blanks have been specified.
- 75Z Value in KCMF/kcfn is invalid. Possible causes:
- Format identifier in KCMF/kcfn changes or is invalid.
  - The edit profile has not been generated or the edit profile changes for message segments (MPUT NT).
- 77Z The message area is missing or cannot be accessed in the specified length.
- 89Z Contents of fields not used in the KDCS parameter area are not equal to binary zero for KCOM PM/RM/HM/EM/ES.

B  
B

### Features of the MPUT call

- The message area is left unchanged when openUTM executes the call.
- Maximum message length for MPUT NT/NE

For messages to terminals, to transport system applications of type APPLI or to a subsequent program unit, the total message may be no longer than the value generated in the NB operand of the MAX control statement.

Otherwise, the length of a message segment is limited to 32,767 bytes and the length of the total message is unlimited.

- You can use screen functions for outputs in format mode (see [page 111ff](#)).

B  
B

If you use edit profiles, KCDF must be set to binary zero, otherwise openUTM returns 70Z.

- You can display multiple messages in a processing step, provided that the messages are issued to job-receiving services and the transaction remains open at the end of the processing step. In all other cases a maximum of **one** message may be displayed.

- A message can consist of two or more message segments, e.g. two or more MPUT NT followed by an MPUT NE with the same KCRN. You can create messages in parallel for job-receiving services, i.e. the service ID in KCRN can change. Any other change to the message destination is not permitted, because it creates an end of message for all messages not yet terminated. Following an unpermitted change of message destinations, no further MPUT is permitted. After terminating a message with MPUT NE or MPUT HM, no further MPUT is permitted with KCRN.
- With PEND KP/RE/FI/ER/FR and PGWT KP/CM, the entire message is transmitted to the communication partner (formatted, if applicable).
- With PEND PA/PR/FC/SP, the message or message segments are passed to the follow-up program whose TAC was specified in KCRN (PEND and MPUT calls). The follow-up program must read each message segment with a separate MGET.
- At the end of a processing step all messages not yet terminated are closed implicitly by openUTM.
- In a program unit run, ending with PEND PA/PR, PS, SP or FC, the MPUT call can be omitted. In a program unit run ending with PEND KP/RE or PGWT KP, at least one MPUT must be entered. Equally, in a dialog service, at least one MPUT have been entered before PEND FI/ER or FR. However, this does not apply, if the KCSEND field contains the value "N" in an OSI TP server service.  
The MPUT is not required in the sign-on service when it is to be followed by a service restart. openUTM then terminated the service that is ready to be restarted abnormally.
- In an asynchronous service, no MPUT may be issued before a PEND FI/ER or FR.
- Empty messages, i.e. KCLM=0, are permitted.

If the empty message is sent to the terminal in format mode, it causes an empty format or partial format to be output.

B  
B

Any dependencies must be taken into account by the FHS start parameters. See the FHS manual.

Empty messages are also permitted in the case of distributed processing.

- An empty message to a transport system application of the type APPLI will not be sent.

- In the case of message segments to socket partners, each message segment must be sent by means of a separate MPUT NT. A separate message segment is created for each MPUT NT/NE.

In the case of MPUT calls with KCLM=0, no messages are sent unless automatically created USP headers (see [page 197](#)) are used. In this case, the header is also sent in the case of MPUT NE/NT with a length of zero. Exception: if the program unit contains only a single MPUT NE/NT and KCLM=0, no header is sent.

- If the receiver is not a terminal, the message is sent transparently, i.e. the message may contain any bit pattern.
- An empty message to a UPIC application is sent since the permission to send has been transferred.
- Sending message segments

A message can consist of a number of message segments, e.g. multiple MPUT NT followed by an MPUT NE with the same KCRN. You can create messages in parallel for job-receiving services, i.e. the service ID in KCRN can change. Any other change to the message destination is not permitted, because it creates an end of message for all messages not yet terminated. If the last message segment is not identified by "NE" in KCOM, the message terminates automatically when PEND occurs.

If there is a change between line and formatted modes or a change of the edit profile, openUTM responds with 75Z and aborts the service.

- Sending partial formats

With a terminal in format mode, a screen can be set up from multiple partial formats. Each of these partial formats must be output with MPUT NT; the last one can be output with MPUT NE.

- Screen functions (KCDF) can only be specified with the **first** MPUT NT. For subsequent MPUT calls, KCDF must contain binary zeros. Otherwise, UTM will terminate the service with 70Z in KCRCCC and K606 in KCRCDC.

Screen functions permitted with partial formats:

KCREPL      Delete screen. This function must be specified when a screen is to be set up anew. If KCREPL is not set, any partial format existing on the screen will be overwritten by a new one. If the same partial format is output again, only the contents of the fields will be replaced (same as KCERAS).

KCALARM      Audible alarm.

KCREPR      Print on hardcopy printer.

KCERAS      Delete unprotected fields; see [section "Using multiple partial formats" on page 113](#).



- Special features of the MPUT RM call

MPUT RM is also permitted if preceded by MPUT NT/NE or MPUT PM calls. The length of the MPUT-RM message is limited to 32,767 bytes.

Only one reset message can be output in a program unit run. Other MPUT calls are permitted before or after MPUT RM. Reset messages are deleted when the user signs off. After a service restart, the field KCRPI contains blanks.

- Special features of the MPUT PM call

openUTM uses MPUT PM to output the last output message of a stacked service on the screen. The following holds:

- for  $KCLM = 0$  the output appears unchanged on the screen, for  $KCLM > 0$  it is overwritten (not exceeding the specified length), but sent in its entirety. The length of the MPUT-PM message is limited by the value generated in the NB operand of the MAX control statement.
- for output messages in line mode you must always specify  $KCLM = 0$  (and  $KCMF/kcfn = \text{blanks}$ )
- the program unit must terminate with PEND FI, otherwise UTM aborts the service with 82Z
- the sign-on service must use this variant at the end of the service if a service restart is to be executed.

- Abstract syntax with distributed processing via the OSI TP protocol

If you specify a value not equal to blanks in KCMF/kcfn when communicating with a partner via the OSI TP protocol, then you have to generate this value as the name of an abstract syntax for the partner. In this case, the application program unit has to transfer the message to openUTM in encoded form, i.e. the application must convert the message to the transfer syntax allocated to the abstract syntax. To do this, the application can use an ASN1 compiler. Abstract syntaxes with the name "CCR" or "OSITP" are not permitted.

- When communicating via a UPIC client program and the LU6.1 protocol, openUTM transfers the format identifier. However, it does not format the message: the partner applications exchange only net data. When using MPUT you can specify any name in the KCMF/kcfn field. This name is indicated to the reading program unit in the KCRMF/kcrfn field after INIT or following a preceding MGET and must be specified in the KCMF/kcfn field when calling MGET.

- MPUT call in the job-submitting service
  - An MPUT call can be used by a service in a job-submitting application to start a service in a job-receiving application, or to send a message to a job-receiving service which has already started.  
You have to specify the service ID, which was allocated to the job-receiving service when APRO DM was called, in the KCRN field as target.
  - In a program unit run of the job-submitting service you can only use MPUT to send messages either to the LTERM partner (KCRN=blanks), to the follow-up program unit (KCRN=TAC) or to one or more job-receiving services (KCRN=service ID).
  - In the job-submitting service KCDF must contain binary 0 for MPUT.
  - Each message segment output with MPUT NT to a job-receiving service has to be read there with a separate MGET.
  - In a program unit run, no messages (message segments) may be sent to a job-receiving service prior to PEND PR or PA, otherwise openUTM aborts the job-submitting service with the error code KCRCCC=82Z.
- MPUT call in the job-receiving service
  - Setting the KDCS parameter area is the same as for the MPUT call for a terminal.
  - When communicating via LU6 protocol, the KCDF field may contain any value which has been passed to the job-submitting service in MGET as screen function. For message segments, only the KCDF value of the first message segment is transferred.  
You have to set KCDF to binary zero when communicating via the OSI TP protocol.
  - Each message segment output with MPUT NT to a partner service has to be read there with a separate MGET.
- Particularities of the MPUT HM call

With MPUT HM, a program run can request a processing confirmation from an OSI TP partner. The following rules apply when using MPUT HM:

  - The call can only be used, if the handshake function was selected for communication; otherwise openUTM aborts the service with 72Z. With INIT, the KB header indicates to the job-receiving service whether the handshake is permitted.
  - An MPUT HM to a partner, which does not have O or U transaction status, is rejected by openUTM with 72Z.
  - At least one MPUT NT to the partner must have been entered prior to MPUT HM.
  - MPUT HM produces an end of message for the communication partner.
  - No data can be passed to the call (KCLM = 0).

- Following MPUT HM only one PEND KP or one PGWT KP is permitted. With all other PEND variants, UTM responds with the return code 82Z.
- Following MPUT HM you may not issue a further CTRL PR/PE call to the same partner.
- Particularities of the MPUT EM call
  - Using the MPUT EM call you can report an error to an OSI TP partner. If a handshake request is confirmed negatively with the call, MPUT EM must be entered as the first MPUT to the partner that requires a processing confirmation. The call must be issued in the same transaction in which the handshake request was read. Otherwise a positive confirmation is sent.
  - No data can be passed to the call (KCLM = 0).
  - An MPUT EM to a partner, which does not have O or U transaction status, is rejected by openUTM with 72Z.
- Particularities of the MPUT ES call
  - The MPUT ES (error system) call can be used in a dialog program unit to create an error message for a UPIC client or socket application. This error message is only sent when the service is terminated abnormally by openUTM (system PEND ER).
  - An error message created with MPUT ES remains valid, provided it is not overwritten with another MPUT ES, until the service is terminated with the output of a dialog message to the UPIC client or socket application. In the case of service concatenation (PEND FC), the error message is thus also valid in the concatenated service.
  - Each subsequent MPUT ES overwrites the error message written last. An MPUT ES with a length of 0 deletes the error message.
  - The length of the MPUT ES message is limited by the value generated in the NB operand of the MAX control statement.
  - If the transaction is reset, the error message is reset to the state of the last synchronization point.

## PADM Administer printouts and printers

The PADM (printer administration) call is used to administer the printers associated with an LTERM printer control.

PADM provides you with the following functions:

- activate or deactivate confirmation mode for an LTERM printer control.  
This function applies globally to the cluster in UTM cluster applications.
- confirm or repeat a printout
- modify the assignment of a printer to an LTERM partner.  
This function is not permitted in UTM cluster applications.
- modify the printer status, i.e. lock and release printer, set up or clear connection to a printer.  
Locking and releasing apply globally to the cluster in UTM cluster applications.
- read information about a printer into the message area
- read information about printouts to be confirmed.



A sample program which you can use here is supplied with openUTM. See the section dealing with the KDCPADM program unit in the openUTM manual “Administering Applications” for further details.

### Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area						
	KCOP	KCOM	KCLA	KCRN	KCLT	KCACT	KCADRLT
Confirm printout	"PADM"	"OK"	0	Control ID	LTERM name	Binary zero	Binary zero
Repeat printout	"PADM"	"PR"	0	Control ID	LTERM name	Binary zero	Binary zero
Change confirmation mode	"PADM"	"AT"/ "AC"	0	Control ID/ blanks	LTERM name	Binary zero	Binary zero
Change printer assignment	"PADM"	"CA"; is not permitted for UTM cluster applications.	0	Control ID	LTERM name	Binary zero	LTERM name
Change printer status	"PADM"	"CS"	0	Control ID	LTERM name	ON/OFF/ CON/DIS	Binary zero
Information on printout	"PADM"	"AI"	44	Control ID/ blanks	LTERM name	Binary zero	Binary zero
Information on printer	"PADM"	"PI"	34	Control ID/ blanks	LTERM name	Binary zero	Binary zero

You must set binary zero in all the fields not used in the KDCS parameter area

### Setting the 2nd parameter

Here you supply the address of the message area to which openUTM is to read the message.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"PADM"
2.	KCOM	"OK"/"PR"/"AT"/"AC"/"CA"/"CS"/"AI"/ "PI"
3.	KCLA	Length in bytes0
4.	KCRN	Control ID/blanks
5.	KCLT	LTERM name of control terminal
6.	KCACT	"ON"/"OFF"/"CON"/"DIS"/binary zero
7.	KCADRLT	Binary zero/LTERM name

**KDCS call**

	1st parameter	2nd parameter
8.	KDCS parameter area	Message area

**C/C++ macro calls**

Macro name	Parameters
KDCS_PADMOK / KDCS_PADMPR / KDCS_PADMAT / KDCS_PADMAC	(nb,kcrn,kclt)
KDCS_PADMCA	(nb,kcrn,kclt,kcadrlt)
KDCS_PADMCS	(nb,kcrn,kclt,kcact)
KDCS_PADMAI / KDCS_PADMPI	(nb,kcla,kcrn,kclt)

openUTM return information					
Message area	Contents				
10.	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; height: 20px;"></td> <td style="width: 50%; height: 20px;">Data/-</td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;">Field name in the KB return area</td> </tr> </table>		Data/-	Field name in the KB return area	
	Data/-				
Field name in the KB return area					
11.	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; height: 20px;">KCRMLM</td> <td style="width: 50%; height: 20px;">Actual length</td> </tr> </table>	KCRMLM	Actual length		
KCRMLM	Actual length				
12.	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; height: 20px;">KCRCCC</td> <td style="width: 50%; height: 20px;">Return code</td> </tr> </table>	KCRCCC	Return code		
KCRCCC	Return code				
13.	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; height: 20px;">KCRCDC</td> <td style="width: 50%; height: 20px;">Internal return code</td> </tr> </table>	KCRCDC	Internal return code		
KCRCDC	Internal return code				
14.	<table border="1" style="width: 100%;"> <tr> <td style="width: 50%; height: 20px;">KCRMF/kcrfn</td> <td style="width: 50%; height: 20px;">Follow-up control ID/blanks</td> </tr> </table>	KCRMF/kcrfn	Follow-up control ID/blanks		
KCRMF/kcrfn	Follow-up control ID/blanks				

For the PADM call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the PADM operation code.
2. In the **KCOM** field
  - OK     confirm printout
  - PR     repeat printout
  - AC     activate acknowledgement mode.  
AC applies globally to the cluster in UTM cluster applications.
  - AT     deactivate acknowledgement mode, i.e. switch back to automatic mode.  
AT applies globally to the cluster in UTM cluster applications.
  - CA     assignment to a different LTERM.  
CA is not permitted in UTM cluster applications.
  - CS     change the state of a printer.  
CS for locking and releasing applies globally to the cluster in UTM cluster applications.
  - AI     read information about printouts to be confirmed
  - PI     read information about printer
3. In the **KCLA** field, the length of the data to be transferred to the message area.
  - if KCOM=AI: 44
  - if KCOM=PI: 34
 and 0 for all other variants.

4. In the **KCRN** field
  - for KCOM = OK/PR/CA/CS the control ID of a printer as generated in the PTERM statement with KDCDEF
  - for KCOM = AT/AC/AI/PI the control ID of a printer (as per PTERM statement) or blanks if the operation is to be valid for all printers assigned to the printer control terminal.
5. In the **KCLT** field, the LTERM name of the LTERM printer control. If your own terminal is not the printer control terminal, your user ID must have administration privileges.
6. In the **KCACT** field, if KCOM = CS the action to be performed:
  - ON the printer is unlocked (STATUS = ON);  
applies globally to the cluster in UTM cluster applications.
  - OFF the printer is locked (STATUS = OFF);  
applies globally to the cluster in UTM cluster applications.
  - CON set up virtual connection to printer
  - DIS clear virtual connection to printer.Enter binary zero for all other variants.
7. In the **KCADRLT** field with KCOM = CA, the name of the LTERM partner to which the printer is to be reassigned. The printer is identified by its control ID specified in KCRN. For all other variants binary zero is entered.

You specify the following for the KDCS call:

8. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area to which UTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLA.
9. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

10. For KCOM = AI/PI: in the specified message area the message in its actual length but not exceeding the desired length.
11. In the **KCRML** field, the actual length of the message which may differ from the length specified in KCLA of the parameter area.
12. In the **KCRCCC** field, the KDCS return code, see next page.



13. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").
14. in the **KCRMf/kcrfn** field, if KCOM = AI/PI, the control ID of the next printer which is assigned to the printer control LTERM or blanks if it is the last assigned printer.

### **KDCS error codes for the PADM call**

The following codes can be analyzed in the program:

- 000 Operation carried out (for KCOM = AI/PI) or the administration job is accepted (for KCOM = OK/PR/CA/CS/AT/AC).
- 01Z Length conflict: KCLA < KCRLM, the message is truncated.
- 40Z System cannot perform the operation (generation error or system error, no authorization for this call), see KRCDC.  
The operation is not permitted in a UTM cluster application.
- 42Z Entry in KCOM is invalid.
- 43Z Length entry in KCLA is negative or invalid.
- 44Z Control ID specified in KCRN is invalid or there is no printer with this control ID in the area of the control terminal.
- 46Z Entry in KCLT is invalid because no LTERM printer control has been defined for this name.
- 47Z Message area missing or cannot be accessed in the specified length.
- 49Z Contents of fields not used in the KDCS parameter area not equal to binary zero.
- 55Z Entry in KCACT is invalid.
- 56Z Entry in KCADRLT is invalid.

An additional error code can be found in the dump:

- 71Z INIT missing in this program.

### Return information in the message area for PADM AI

You can use a language-specific data structure to structure the message area when calling PADM AI. For COBOL this is defined in the KCPADC COPY element and for C/C++ in the *kcpad.h* include file. It has the following structure:

Byte	Field name COBOL/C/C++	Meaning
1 - 8	KCACKCID	Control ID of the printer
9 - 16	KCGENUID	UTM user ID of the job originator
17 - 24	KCDPUTID	Job ID (assigned by openUTM)
25 - 33	KCGENTIM <sup>1</sup>	Time of the FPUT/DPUT call in the form <i>dddhhmmss</i> :
25 - 27	KCGENDOY	<i>ddd</i> Day of the year (value range 001 - 366)
28 - 29	KCGENHR	<i>hh</i> Hour (value range 00 - 23)
30 - 31	KCGENMIN	<i>mm</i> Minute (value range 00 - 59)
32 - 33	KCGENSEC	<i>ss</i> Second (value range 00 - 59)
34 - 42	KCSTTIM <sup>1</sup>	For time-delayed jobs the required start time <i>dddhhmmss</i> :
34 - 36	KCSTDOY	<i>ddd</i> Day of the year (value range 001 - 366)
37 - 38	KCSTHR	<i>hh</i> Hour (value range 00 - 23)
39 - 40	KCSTMIN	<i>mm</i> Minute (value range 00 - 59)
41 - 42	KCSTSEC	<i>ss</i> Second (value range 00 - 59)
		Blanks are entered for a job without time delay.
43	KCPOSMSG	Y Positive confirmation job N No positive confirmation job
44	KCNEGMSG	Y Negative confirmation job N No negative confirmation job

<sup>1</sup> For C/C++ the summary fields KCGENTIM and KCSTTIM are not defined. However, the specific fields for day/month/year/working day/hour/minute/second are defined.

If there is no printout to be confirmed, openUTM writes blanks in the message area.

### Return information in the message area for PADM PI

You can use a language-specific data structure to structure the message area when calling PADM PI. For COBOL this is defined in the KCPADC COPY element and for C/C++ in the *kcpad.h* include file. It has the following structure:

Byte	Field name COBOL/C/C++	Meaning
1 - 8	KCPRTCID	Control ID of the printer
9 - 11	KCSTATE	Status of the printer: ON or OFF
12	KCCON	Printer connection: Y = printer connected N = printer disconnected
13 - 14	KCPRTMOD	Print mode: AT = automatic mode AC = confirmation mode
15 - 22	KCLTRMNM	LTERM name of the assigned printer
23 - 28	KCFPMSGs	Number of output jobs for this printer (without number in KCDPMSGs)
29 - 34	KCDPMSGs	Number of time-driven jobs pending for this printer whose target time has not yet been reached.

### Features of the PADM call

- PADM calls with the modification AT/AC are not executed until the end of the transaction. For the calls PADM OK/PR/CS/CA, a check is made only at the end of the transaction as to whether the call can be executed at all. Thus KCRCCC = 000 after such a call does not necessarily guarantee that the call can be successfully executed, because in the meantime there might be loss of connection to the printer, for example. You can issue a PADM AI or PADM PI call in a subsequent transaction to check whether a PADM OK/PR/CS/CA call has been successful.
- If there is no printout to confirm for a PADM OK/PR call, then PADM OK or PADM PR provides the return code 40Z.
- The printer assignment with PADM CA may only be changed if the two participating LTERMs are not connected with the application at call time, otherwise openUTM responds with 40Z.
- When the printer is locked, openUTM responds with 40Z to reject a PADM CS call designed to set up a connection (KCACT = CON).

## PEND Terminate program unit

You use the PENDING (program end) call to terminate a program unit. All UTM program units, including the event services (BADTACS, MSGTAC, SIGNON) must be exited via the PENDING call (exception: event exits). Depending on the type of program involved one of the variants of the PENDING call is used.

With distributed processing, the PENDING calls of job-submitting services and job-receiving services have to be matched, see also [chapter “Program structure in distributed processing” on page 121ff.](#)

The abbreviations in the KCOM field are derived from the following terms:

<b>PS</b>	<b>program and sign(on)</b>
<b>PA/PR</b>	<b>program</b>
<b>KP</b>	<b>keep</b>
<b>RE</b>	<b>return</b>
<b>SP</b>	<b>synchronization point</b>
<b>FI</b>	<b>finish</b>
<b>FC</b>	<b>finish and continue</b>
<b>RS</b>	<b>reset</b>
<b>ER</b>	<b>error</b>
<b>FR</b>	<b>finish and reset</b>

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area		
	KCOP	KCOM	KCRN
Continue sign-on service after authorization check in follow-up program unit	"PEND"	"PS"	TAC of the follow-up program unit
Continue processing step in the follow-up program unit	"PEND"	"PA"/ "PR"	TAC of the follow-up program unit
Terminate processing step without terminating trans.	"PEND"	"KP"	TAC of the follow-up program unit
Terminate processing step and transaction	"PEND"	"RE"	TAC of the follow-up program unit
Terminate transaction; continue processing step	"PEND"	"SP"	TAC of the follow-up program unit
Terminate processing step, transaction and service	"PEND"	"FI"	—
Terminate transaction and service; continue proc. step in concatenated service	"PEND"	"FC"	TAC of the follow-up program unit
Reset transaction	"PEND"	"RS"	Blanks
Abort service, roll back the transaction, request dump and restart the application program	"PEND"	"ER"	—
Abort service and roll back the transaction no dump and no application program restart	"PEND"	"FR"	Blanks



If KCOM = PS/FC/SP/RS, all fields not used in the KDCS parameter area must be set to binary zero.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"PEND"
2.	KCOM	"PS"/"PA"/"PR"/"KP"/"RE"/"FI"/"FC"/ "RS"/"ER"/"FR"/"SP"
3.	KCRN	Follow-up transaction code/ blank/—

**KDCS call**

	1st parameter	2nd parameter
4.	KDCS parameter area	message area

**C/C++ macro calls**

Macro name	Parameters
KDCS_PENDER / KDCS_PENDFI / KDCS_PENDFR / KDCS_PENDRS	()
KDCS_PENDFC / KDCS_PENDKP / KDCS_PENDPA / KDCS_PENDPR / KDCS_PENDPS / KDCS_PENDRE / KDCS_PENDSP	(kcrn)

**openUTM return information**

	Field name in the KB return area	Contents
6.	KCRCCC	Return code
7.	KCRCDC	Internal return code

For the PEND call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the PEND operation code.
2. In the **KCOM** field, the PEND call variants:

PS	continuation of the sign-on service in a follow-up program
PR/PA	continuation of the processing step in a follow-up program
KP	end of the dialog step without transaction end
RE	end of the dialog step with transaction end
SP	end of transaction; continuation of the processing step in a follow-up program
FI	end of service
FC	end of service with concatenation of services
RS	transaction reset (with subsequent service restart if synchronization point exists)
ER	end of service because of error; the service is terminated abnormally, the transaction reset and a dump is written
FR	end of service because of error. The service is terminated abnormally, the transaction reset (without dump).
3. In the **KCRN** field, according to variant
  - for PEND KP/RE:  
the TAC of the follow-up program unit in which processing should be continued after receiving the next input message.
  - for PEND PA/PR/PS/SP:  
the TAC of the follow-up program unit in which processing of the same step should be continued.
  - for PEND RS/FR:  
blanks
  - for PEND ER/FI: irrelevant.

You specify the following for the KDCS call:

4. 1st parameter: the address of the KDCS parameter area.
5. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

6. in the **KCRCCC** field: the KDCS return code, see below.
7. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### **KDCS return codes in the KCRCCC field**

These error codes are only to be taken from the dump:

- 000 Operation carried out (with requested PEND ER).
- 70Z Operation PEND cannot be performed (system or generation error, deadlock, timeout), see KCRCDC.
- 71Z No INIT entered in this program unit run.
- 72Z KCOM contains an invalid operation modifier in a call
  - in the MSGTAC program
  - in the sign-on service
  - outside of the sign-on service
  - in the server service
  - in the asynchronous serviceor the operation modifier conflicts with
  - the database transaction
  - the communication protocol used
  - the status of the sign-on service
  - waiting for a DGET message
- 74Z
  - the KCRN field contains a value that is not a TAC or a follow-up TAC
  - the user is not authorized to use the TAC
  - a switch between a dialog TAC and an asynchronous TAC is pending
  - a switch between program units with KDCS API and X/OPEN API is pending
  - KCRN does not contain SPACES in a PEND RS,FR
  - when waiting for a DGET message, the TAC specified in KCRN is not in a TAC class
- 81Z KCRN entry in PEND PA/PR/FC/SP/PS (TAC of the follow-up program) conflicts with KCRN in MPUT (e.g. MPUT TAC 1 and PEND xx TAC2).



- 82Z KCOM or KCRN entry in PEND conflicts with entries in the preceding MPUT:
  - MPUT with KCRN=blanks for PEND PA/PR/PS/FC
  - MPUT with KCRN=TAC for PEND KP/RE/FI/ER/FR
  - MPUT with KCRN=RESET ID and no PEND RS
  - MPUT with KCRN=service ID for PEND PA/PR/FI/ER/FR.
  
- 83Z
  - dialog program did not issue an MPUT prior to a PEND KP/RE/FI/ER/FR
  - no reset message was sent with MPUT RM prior to a PEND RS in a follow-up transaction
  - no MPUT PM issued prior to a program unit terminated with PEND FI in a sign-on service with service restart.
  
- 86Z
  - job complex not terminated at the time of PEND call
  - no job associated with DPUT NI call generated
  - no asynchronous job sent to a service addressed with APRO AM.
  
- 87Z PEND call conflicts with transaction or service status of a remote service.
  
- 89Z Contents of fields not used in the KDCS parameter area are not equal to binary zero (only if KCOM = PS/FC/RS/SP).

The table below shows the PEND variants and the associated actions.

Variants	Meaning	openUTM actions
PS	Sign-on service to be continued in follow-up program  No synchronization point	<ul style="list-style-type: none"> <li>– Checks authorization data</li> <li>– Possibly executes intermediate dialog to query password/ID card information.</li> <li>– Provides this program unit's MPUT for the follow-up program unit or saves it in the page pool if an intermediate dialog is required.</li> <li>– Takes over the follow-up program unit's TAC from the KCRN field and starts the follow-up program either immediately or on termination of the intermediate dialog.</li> </ul>
PA or PR	Processing step to be continued in the follow-up program The transaction remains open No synchronization point	<ul style="list-style-type: none"> <li>– Provides MPUT of this program unit to the follow-up program</li> <li>– Takes TAC of the follow-up program from the KCRN field. The follow-up program unit is started either immediately or after a delay depending on the TAC classes or TAC priority control.</li> </ul> If a DGET message is to be waited for, openUTM does not start the follow-up program unit until a message arrives for this queue or is placed back in the queue (redelivery) or if the maximum wait time has expired or if the queue was deleted (see DGET call).
SP	Termination of transaction. However, processing step should be continued in the next program unit.  Synchronization point!	<ul style="list-style-type: none"> <li>– Closes DB transaction</li> <li>– Executes DPUT, FPUT, LPUT, PTDA, SPUT and SREL of the transaction.</li> <li>– Deletes the FGET message in the first transaction of an asynchronous service.</li> <li>– Deletes messages processed with DGET from their queues.</li> <li>– Executes MPUT of this program unit</li> <li>– Takes TAC of the follow-up program from KCRN field. The follow-up program unit is started either immediately or after a delay depending on the TAC classes or TAC priority control.</li> </ul>
KP	End of the processing step without terminating the transaction. It is to be continued in the next processing step.  No synchronization point	<ul style="list-style-type: none"> <li>– Executes MPUT of this program unit (if necessary formats message).</li> <li>– If necessary, takes data in KB and transfers it to the follow-up program as soon as it is initialized.</li> <li>– Takes TAC of the follow-up program from KCRN field and starts follow-up program as soon as the next message arrives.</li> </ul>

<b>Variants</b>	<b>Meaning</b>	<b>openUTM actions</b>
RE	End of a processing step and simultaneous end of the transaction. The service is to be continued in a follow-up program.  Synchronization point	<ul style="list-style-type: none"> <li>– Closes DB transaction.</li> <li>– Executes DPUT, FPUT, LPUT, PTDA, SPUT and SREL of the transaction.</li> <li>– Deletes the FGET message in the first transaction of an asynchronous service.</li> <li>– Deletes messages processed with DGET from their queues.</li> <li>– Executes MPUT of this program unit (formats message if necessary).</li> <li>– Releases resources.</li> <li>– Takes TAC of the follow-up program unit from KCRN field and starts follow-up program unit as soon as the next message arrives.</li> <li>– If necessary, fetches data from KB and transfers it to the follow-up program unit as soon as it is initialized.</li> </ul>
FI	End of service and end of the transaction.  Synchronization point!	<ul style="list-style-type: none"> <li>– Closes DB transaction.</li> <li>– Executes DPUT, FPUT, LPUT, PTDA, SPUT and SREL (for GSSB) of the transaction.</li> <li>– Deletes the FGET message in the first transaction of an asynchronous service.</li> <li>– Deletes messages processed with DGET from their queues.</li> <li>– Releases LSSBs of the service.</li> <li>– Executes MPUT of this program unit (formats message if necessary).</li> <li>– Releases resources</li> </ul>
FC	End of service and end of the transaction, the processing step is to be continued in the concatenated service  Synchronization point!	<ul style="list-style-type: none"> <li>– Closes DB transaction.</li> <li>– Executes DPUT, FPUT, LPUT, PTDA, SPUT and SREL (for GSSB) of the transaction.</li> <li>– Deletes messages processed with DGET from their queues.</li> <li>– Releases LSSBs of the service.</li> <li>– Transfers MPUT of this program unit to the first program unit of the concatenated service.</li> <li>– Releases resources</li> </ul>

Variants	Meaning	openUTM actions
RS	Reset a transaction to the last synchronization point	<p>In the first transaction of a service:</p> <ul style="list-style-type: none"> <li>– Resets UTM and DB transaction</li> <li>– Terminates service (with message K034 to the client).</li> <li>– only for terminals and TS partner: Outputs last output message of the preceding service (if available).</li> <li>– Restart asynchronous services and follow-up services (if chained services are used) following a reset.</li> </ul> <p>In a follow-up transaction of service:</p> <ul style="list-style-type: none"> <li>– Resets UTM and DB transaction to the last synchronization point and, if necessary, restarts screen with message K034.</li> <li>– Starts the program unit addressed at the last synchronization point.</li> <li>– Transfers reset message to this program unit.</li> </ul> <p>For all the transactions in a service:</p> <ul style="list-style-type: none"> <li>– Messages processed with DGET can be processed again if the maximum number of redeliveries specified in the generation has not yet been reached. Otherwise they are deleted or, in the case of TAC queue messages, may be saved in the dead letter queue.</li> </ul>
ER (ERror)	<p>End of service with dump and restart of the application program</p> <p>Reset to last synchronization point (exception: MPUT)</p>	<ul style="list-style-type: none"> <li>– Writes the UTM dump.</li> <li>– Resets the UTM and DB transaction.</li> <li>– Messages processed with DGET can be processed again if the maximum number of redeliveries specified in the generation has not yet been reached. Otherwise they are deleted or, in the case of TAC queue messages, may be saved in the dead letter queue.</li> <li>– In the first transaction of an asynchronous service: The FGET message can be processed again if the maximum number of redeliveries specified in the generation has not yet been reached. Otherwise it is deleted or may be saved in the dead letter queue.</li> <li>– Causes dump of the application program.</li> <li>– Executes MPUT of this program unit (formats message if necessary)</li> <li>– Restarts application program.</li> <li>– Releases resources.</li> </ul>

Variants	Meaning	openUTM actions
FR	<p>End of service, no dump, the application program remains loaded</p> <p>Reset to last synchronization point (exception: MPUT)</p>	<ul style="list-style-type: none"> <li>– Resets the UTM and DB transaction.</li> <li>– Messages processed with DGET can be processed again if the maximum number of redeliveries specified in the generation has not yet been reached. Otherwise they are deleted or, in the case of TAC queue messages, may be saved in the dead letter queue.</li> <li>– In the first transaction of an asynchronous service: The FGET message can be processed again if the maximum number of redeliveries specified in the generation has not yet been reached. Otherwise it is deleted or may be saved in the dead letter queue.</li> <li>– Executes MPUT of this program unit (formats message if necessary)</li> <li>– Releases resources.</li> </ul>

### Features of the PEND call

- Following a PEND, no branch is made back to the calling program. Thus, the KDCS return code cannot be evaluated there.
- In case of error, if PEND executes abnormally, openUTM calls PEND ER internally.
  - With dialog processing, an error message is sent to the client. The transaction is reset and the service is terminated. A new service can be started.
  - An asynchronous service is aborted and a message is written to the system log file SYSLOG.
- If another call led to a KDCS return code  $\geq 70Z$ , openUTM calls PEND ER internally.
- If a program unit calls PEND ER/FR in a dialog service, you must first use MPUT to send a message to the client or the job-submitting service - with one exception: This does not apply to a OSI TP job-submitting service for which KCSEND contains the value "N". If openUTM calls PEND ER internally ( $KCRCC \geq 70Z$ ), a default message is issued. However, if a separate error message was created by means of MPUT ES in communication with the UPIC client, this error message is sent to the UPIC client instead.
- A reset message has to be sent with MPUT RM prior to a PEND RS call in a follow-up transaction. Thus a PEND RS should not be issued in the first transaction of a service otherwise no reset message can be read. Unlike PEND ER, PEND RS does not cause a dump to be written.
- If the PEND RS call is used for communication with a UPIC client program, you must note the following:

If the preceding transaction was terminated with PEND SP or PEND FC, then the local transaction is rolled back and the service with the follow-up program unit specified in PEND SP or PEND FC continues execution when PEND RS is called.(local service restart)

If the preceding transaction was **not** terminated with PEND SP/FC and the service is running under a user ID **with** the restart property, then the service is rolled back to the last synchronization point and the dialog with the UPIC client is terminated. A new OSI TP dialog can be started under the user ID and the service restart requested. In all other cases, openUTM terminates the service with PEND FR.

- If PEND RS is used for distributed processing via the OSI TP protocol without commit functionality, you must note the following:
  - if called in a job-receiving service:

If the preceding transaction was terminated with PEND SP, then the local transaction is rolled back and the service with the follow-up program unit specified in PEND SP continues execution when PEND RS is called (local service restart). If the preceding transaction was **not** terminated with PEND SP and the service is running under a user ID **with** the restart property, then the service is rolled back to the last synchronization point and the dialog with the UPIC client is terminated. A new OSI TP dialog can be started under the user ID and the service restart requested.

In all other cases, openUTM terminates the service with PEND FR.
  - if called in a job-submitting service, all job-receiving services of this service are terminated using PEND FR.
- If a PEND RS call is issued in a transaction previously terminated with PGWT CM, the service is terminated with PEND FR.
- Any messages or message segments which openUTM maintained for the program following INIT, but which were not read in the program with MGET or FGET, are lost with PEND (likewise with PEND PA or PR).

The same applies when not all of a DGET message is read: the remaining parts of the message that were not read are lost.
- If a DB transaction was terminated prior to a PEND RE/SP/FI/FC, execution is delayed until PEND RE/SP/FI/FC.
- PEND KP locks resources (LSSBs, GSSBs, TLS, ULS and, if applicable, database areas) beyond a dialog step.

Except in the case of distributed processing, it is therefore advisable

  - to use the PEND KP sparingly in order to keep global resource occupancy times short
  - not to reserve the global resources until the final dialog step when using PEND KP.
- Any MPUT output to be executed by a PEND KP is lost if the system crashes and is then restarted. The service is reset to the last synchronization point. The dialog output at the synchronization point is available to the client following the restart.

Exception: Automatic screen restart was omitted for this client with the generation operand RESTART=NO (see the openUTM manual "Generating Applications", LTERM or USER statement).
- PEND SP is only permitted in distributed processing via LU6.1 if no partner services with open transactions are available.

- A PEND PA/PR or SP can cause a task switch in a dialog or asynchronous service if the follow-up program unit is in a different TAC class than the program unit run calling PEND (see the openUTM manual "Generating Applications", TAC classes), or if the application was generated with the TAC-PRIORITIES statement.
- A PEND PS call may only be specified in the sign-on service and only if the status of the sign-on service allows this.
- If the PEND PA/PR or PS call is executed without calling MPUT beforehand in the sign-on service for a UPIC client after receiving a message from the client, then the follow-up program unit can still read unread message (segments) from the UPIC client.
- PEND FC terminates the UTM service, but not the UPIC conversation. If the PEND PA/PR or PS call is executed without calling MPUT beforehand in the sign-on service for a UPIC client after receiving a message from the client, then the follow-up program unit can still read unread message (segments) from the UPIC client. In this case the first program unit of the chained service receives the value F (first), and not C (chained) as its service indicator in the KBKOPF because it contains a message from the client.
- If an asynchronous service or a follow-up service concatenated with PEND FC is interrupted during the first transaction, then openUTM restarts the service.
- Program unit runs in asynchronous services can only use PEND KP and RE if they have previously supplied a message for a job-receiving service with MPUT.
- Note that, if PEND ER/FR is called in the job-receiving service, you cannot send a message to the job-submitting service (as with PEND ER/FR to the terminal). Nevertheless, you have to issue an MPUT call before a PEND ER/FR as otherwise openUTM terminates the service. The job-submitting service then receives status information with service status Z (instead of E).
- When using distributed processing, you have to take account of the service and transaction status of the partner when calling PEND. For further information see [chapter "Program structure in distributed processing" on page 121](#).
- If a DGET message is to be waited for, only PEND PA/PR (with waiting) or PEND ER/FR/RS (without waiting) are permitted.



## PGWT Set wait point in program without terminating program unit

The PGWT call (program wait) sets the program unit to an internal wait point **without** terminating the unit. You can

- terminate the processing step without terminating the transaction.
- neither terminate the processing step nor the transaction in order to wait for a message at a message queue (no previous MPUT permitted).
- terminate the transaction. If an MPUT message was sent previously, the processing step is terminated; otherwise, it is continued.
- reset the transaction and continue the processing step.

The program unit is always continued by the same process (process ID remains the same). The PGWT call is comparable to a PEND call followed by an implicit INIT call.

If you enter a value greater than 0 for the parameter KCLI in the KDCS parameter area, then openUTM supplies information about the application, the system and the communication partner. The PGWT call is then comparable to a PEND KP call followed by an implicit INIT PU call.

The following table shows the meaning of the PGWT call and the associated actions:

Variant	Meaning	openUTM actions
PGWT KP	End of processing step without program unit or transaction end; if necessary request information (KCLI > 0)	<ul style="list-style-type: none"> <li>– Send MPUT message</li> <li>– Continue program as soon as response received</li> <li>– For KCLI &gt; 0: Make requested information available in message area</li> </ul>
PGWT PR	Wait without terminating processing step, program unit or transaction; if necessary request information (KCLI > 0)	<ul style="list-style-type: none"> <li>– Continue program as soon as there is a message at the queue</li> <li>– For KCLI &gt; 0: Make requested information available in message area</li> </ul>
PGWT CM	Terminate transaction without terminating the program unit; if necessary request information (KCLI > 0)	<ul style="list-style-type: none"> <li>– If MPUT was issued: Send MPUT message and continue program as soon as response received.</li> <li>– Without MPUT: Continue processing without waiting, exception for OSI TP see <a href="#">page 394</a>.</li> <li>– For KCLI &gt; 0: Make requested information available in message area</li> </ul>
PGWT RB	Reset transaction; if necessary request information (KCLI > 0)	<ul style="list-style-type: none"> <li>– Reset transaction</li> <li>– Continue processing, exception for OSI TP see <a href="#">page 394</a>.</li> <li>– For KCLI &gt; 0: Make requested information available in message area</li> </ul>



The PGWT KP/RP call locks resources after the processing step is terminated. It should therefore be used sparingly.

### Setting the KDCS parameter area (1st parameter)

Function of the call	Entries in the KDCS parameter area		
	KCOP	KCOM	KCLI
Terminate processing step without end of program unit or transaction	"PGWT"	"KP"	0 / Length of the message area
Wait without terminating processing step and transaction	"PGWT"	"PR"	0 / Length of the message area
Terminate transaction without end of program unit	"PGWT"	"CM"	0 / Length of the message area
Reset transaction and continue processing step	"PGWT"	"RB"	0 / Length of the message area

### Setting the 2nd parameter (only necessary if KCLI > 0)

Here you enter the address of the message area into which openUTM is to write the requested information. To structure the message area you can use the same data structure as for the INIT PU call, i.e. the KCINIC COPY element for COBOL, the *kcini.h*. include file for C/C++.

In the header of the data structure, you specify the version number of the structure and select which information openUTM is to supply.

For a detailed description of the data structure, see the INIT PU call, [page 315ff.](#)

### Setting the parameters

	Field name in the KDCS parameter area	Contents
1.	KCOP	"PGWT"
2.	KCOM	"KP"/"PR"/"CM"/"RB"
3.	KCLI	0/Length in bytes

### Setting the header of the message area (only necessary if KCLI > 0)

	Field name in message area	Contents
4.	KCVER/if_vers	Version number (3)
5.	KCDATE/dattim_info	Request date and time (Y/N)
6.	KCAPPL/appl_info	Request application information (Y/N)
7.	KCLOCALE/locale_info	Request locale information (Y/N)
8.	KCOSITP/ositp_info	Request OSI TP information (Y/N)
9.	KCENCR/encr_info	Request encryption information (Y/N)
10.	KCMISC/misc_info	Request miscellaneous information (Y / N)

### KDCS call

	1st parameter	2nd parameter
11.	KDCS parameter area	Message area (only necessary if KCLI > 0)

### 12. C/C++ macro calls

Macro name	Parameters
KDCS_PGWTKP	( )
KDCS_PGWTKP_PU / KDCS_PGWTTPR/ KDCS_PGWTM / KDCS_PGWTRB	(nb,kcli)

**openUTM return information**

	Message area	Contents
13.	<input type="text"/> Field name in KB return area	Data (only if KCLI > 0)
14.	KCRMLM	Length of transferred data (only if KCLI > 0)
15.	KCRCCC	Return code
16.	KCRCDC	Internal return code
17.	KCRMf/kcrfn	Format identifier/blanks
18.	KCRPI	Service ID/reset ID/blanks

For the PGWT call you make the following entries in the KDCS parameter area:

- In the **KCOP** field, enter the "PGWT" operation name.
- In the **KCOM** field, the variant of the PGWT call:
  - KP End of processing step without transaction end
  - PR Wait without terminating the processing step and transaction
  - CM End of the transaction
  - RB Rollback of the transaction
- in the **KCLI** field, enter the length of the message area to which openUTM is to transfer the information. Enter the length in bytes. The value entered in KCLI specifies the maximum number of bytes of information that openUTM transfers to the message area.  
  
If KCLI is greater than zero, then you must specify the address of the message area in the 2nd parameter when you call PGWT. The information is equivalent to that for the INIT PU call.

All unused fields of the parameter area must contain binary zero.

Setting the header of the message area (only necessary if  $KCLI > 0$ ):

4. In the **KCVER/if\_ver** field, enter the version number of the data structure. The current version is version 3.
5. Enter Y in the **KCDATE/dattim\_info** field if you want information on the date and time of the start of the application and the program unit run, otherwise enter N.
6. Enter Y in the **KCAPPL** field if you want to request information about the application, system and communication partner, otherwise enter N.
7. Enter Y in the **KCDATE/dattim\_info** field if you to want to request information about the language environment of the LTERM partner, otherwise enter N.
8. Enter Y in the **KCOSITP/ositp\_info** field if you require OSI TP-specific information, otherwise enter N.
9. Enter Y in the **KCENCR/encr\_info** field if you require information about the encryption methods used between the client and the UTM application, otherwise enter N. (The encryption mechanism can be coordinated. See the openUTM manual "Generating Applications".)
10. Enter Y in the **KCMISC/misc\_info** field if you require miscellaneous information (e.g. number of queued messages in the user's queue, password validity, time of last sign-on), otherwise enter N.

You specify the following for the KDCS call:

11. 1st parameter: the address of the KDCS parameter area.  
2nd parameter (only necessary if  $KCLI$  is not equal to 0):  
the address of the message area to which openUTM is to write information. For return information you can use the KCINIC data structure in COBOL and the *kcini.h* include file in C/C++ (description see [page 316](#)).
12. The use of C/C++ macro calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

13. only if  $KCLI > 0$ :  
in the **message area**, the information in its actual length, up to a maximum length of the value specified in  $KCLI$ . The data supplied by the PGWT call corresponds to the return information of the INIT PU call and is described on [page 316](#).
14. only if  $KCLI > 0$ :  
in the **KCRLM** field, the length of the information actually transferred by openUTM, provided that  $KCRCCC = 000$  or  $07Z$ . If  $KCRCCC \geq 40Z$  then  $KCRLM = 0$ .

15. in the **KCRCCC** field, the KDCS return code (length 3 bytes).  
For possible return codes and their meaning, see below.
16. in the **KCRCDC** field, the internal return code of openUTM (see openUTM manual “Messages, Debugging and Diagnostics”).
17. only for PGWT KP and PGWT CM  
in the **KCRMF/kcrfn** field (similar to the INIT call):

- for a message from a terminal:

Blanks (in line mode) or the format name (in format mode) of the last screen output, i.e. the name specified in the KCMF/kcrfn field with the MPUT of the last dialog step. If the last output consisted of multiple partial formats, KCRMF/kcrfn contains the name of the first partial format into which data was entered. If no data was entered in any of the partial formats, KCRMF/kcrfn contains the name of the first partial format.

B  
B

If an edit profile was used in the last screen output, KCRMF/kcrfn contains this edit profile.

- for distributed processing via LU6.1:

in the job-submitting service:

the format identifier of the first message segment sent by the job-receiving service to the job-submitting service, or blanks if a status flag exists for the service ID specified in KCRP;

in the job-receiving service:

the format identifier of the first message segment from the job-submitting service that can be read by the job-receiving service with MGET.

- for distributed processing via OSI TP:

If the program unit run was started because of a distributed dialog, KCRMF/kcrfn in the job-submitting service contains the name of the abstract syntax which was allocated to the message by the job submitter; if the field contains blanks, the UTD syntax is selected as the abstract syntax.

In the job-receiving service, KCRMF/kcrfn contains the name of the abstract syntax which was allocated to the message by the job-receiving service described in KCRPI. If the field contains blanks, the UTD syntax is selected as the abstract syntax or an error message from the partner is present.

18. only for PGWT KP and PGWT CM  
in the **KCRPI** field (similar to the INIT call):
- For a message from a terminal: blanks.
  - In the job-submitting service with distributed processing:  
the service ID of the job-receiving service if a message from the job receiver exists.
  - In the job-receiving service with distributed processing: blanks

### **KDCS return codes in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Operation carried out successfully. If a message area was specified for the call (KCLI > 0), the requested information was transferred to the message area in its full length.
- 07Z Function was executed, the available message area is too short (Length in KCLI insufficient). No or incomplete information was returned.
- 40Z For PGWT CM: The function was not executed. An error that does not permit continuation of the current transaction has occurred. The transaction is rolled back implicitly with PGWT RB.
- 48Z Only when KCLI is greater than zero:  
Invalid data structure version.

These return codes can only be found in the dump:

- 70Z The PGWT operation could not be performed (system or generation error, deadlock, timeout).
- 71Z In this program no INIT has yet been issued or the call was issued by an MSGTAC program.
- 72Z Entry in KCOM is invalid or KCOM = CM/RB was specified and a partner communicating with the LU6.1 protocol is involved in the current distributed transaction.
- 77Z The address of the message area specified when the call was issued is invalid.
- 82Z An MPUT was issued to a program unit run before a PGWT KP.
- 83Z The program unit did not issue an MPUT before a PGWT KP or an MPUT was issued before a PGWT PR.
- 87Z The PGWT call conflicts with the transaction or service status.
- 88Z Interface version of the data structure (for KCLI > 0) is invalid.
- 89Z When the function was called, unused parameters were not set to binary zero.

### Features of the PGWT call

- Any messages or message segments which are held by openUTM for the program following INIT and which are not read by the program using MGET are lost.
- PGWT KP and PR
  - A PGWT KP call corresponds to a PEND KP with a following INIT/INIT PU. PGWT KP is always allowed, if a PEND KP call is allowed.
  - A PGWT PR call corresponds to a PEND PA/PR with a following INIT/INIT PU. PGWT PR is meaningful only if preceded by a DGET call with wait and if it is necessary to wait for a message for the specified queue. After PGWT PR the program waits until a message arrives for this queue.
  - PGWT KP and PR does not produce a synchronization point! Dialog messages initiated by MPUT for PGWT KP calls are output. The remaining output operations LPUT, FPUT, SPUT, PTDA and the release action SREL remain stored until the next synchronization point. Locked resources remain locked after the call.
  - If a DB transaction began prior to a PGWT KP or PR, this transaction is not terminated! The transaction is only terminated if a PEND RE, SP, FI, FC or a PGWT CM call is issued. RSET, PEND, RS FR, ER or PGWT RB reset the transaction.
  - Any MPUT output to be executed by a PGWT KP is lost if the system crashes and is then restarted. The service is reset to the last synchronization point. The client can output the dialog as recorded at the last synchronization point.  
Exception: Automatic screen restart was omitted for this client with the generation operand RESTART=NO (see the openUTM manual "Generating Applications", LTERM or USER statement).
- PGWT CM and RB
  - PGWT CM with an MPUT message is always allowed whenever a PEND RE is allowed. PGWT CM then corresponds to PEND RE with a following INIT/INIT PU.
  - PGWT CM without an MPUT message is always allowed whenever a PEND SP is allowed. PGWT CM then corresponds to PEND SP with a following INIT/INIT PU.
  - PGWT CM sets a synchronization point while preserving the program context. With regard to database transactions PGWT CM behaves in the same way as PEND SP or PEND RE.
  - PGWT CM and PGWT RB are not allowed if the LU6.1 protocol is used for communication with a partner in the current transaction.
  - PGWT RB rolls back the transaction. In contrast to RSET or PEND RS the program context is preserved. The program context includes, for example, the KB program area, SPAB and local data areas.



- No reset message may be generated before PGWT RB.
- No service restart is possible at a synchronization point set using PGWT CM. The follow-up transaction can therefore be reset only with PGWT RB without terminating the abnormal service.
- If the PGWT call cannot be executed successfully, openUTM calls PEND ER internally.
- Continuation of processing
  - After a PGWT KP call or a PGWT CM call with MPUT a return is made to the calling program as soon as all responses are available.
  - Processing is continued immediately after a PGWT CM without MPUT and after a PGWT RB, exception see distributed processing with OSI TP on [page 394](#).
  - After a PGWT PR call processing is continued as soon as a message arrives in the queue. A DGET call must be used to read such messages.
- PGWT KP and PGWT PR lock resources (LSSBs, GSSBs, TLS, ULS, and possibly database areas) outside the processing dialog step. Calls from other transactions (SGET, SPUT, SREL, PTDA or GTDA) that wish to access these resources are rejected with a return code (40Z and KCR CDC code). Therefore, it is recommended when using PGWT KP/PR to first allocate the global resources before use, and to release them again immediately afterwards.



Use this call sparingly, because it occupies the task and no other job can be processed during the wait period. This applies especially if the PGWT call is waiting for an entry from a terminal. In this case, the task is blocked until an entry is input via the keyboard!

### Particularities of the PGWT call in the job-submitting service

When using a PGWT call in the job-submitting service a message can be sent to a job-receiving service with MPUT. If a job-receiving service is to be started with this message, a session is allocated to this job-receiving application in PGWT management. The program unit with the PGWT call (the job-submitting service) only continues when the result of the job-receiving service is received.

If the job-receiving service cannot return a result, the transaction is reset in the job-submitting service and the program unit is restarted from the last synchronization point. In this program unit run, the input message can be read again with MGET. The status information of the job-receiving service can also be read with further MGET calls, where the cause of the errors is displayed. This means that the first program unit of a transaction must always handle errors in the job-receiving service, which may occur during communication with job-receiving services.

If the job-submitting service has been reset to a synchronization point, at which an entry is expected from a terminal, openUTM carries out a screen restart from the synchronization point, outputs a message, and requests the entry again.

### **PGWT call in a distributed OSI TP service with Commit**

If the Commit functionality was selected with distributed processing via OSI TP, PGWT CM without MPUT and PGWT RB **always** result in a wait point, i.e.:

- for PGWT CM without MPUT, processing is not continued until the transaction end has been confirmed by all partners or the transaction has been reset, e.g. due to an error.
- for PGWT RB, processing is not continued until reset has been confirmed by all partners.

A return is also made to the program unit if

- for an PGWT KP call, a situation is detected which makes it impossible to set the transaction forward.
- for a PGWT CM or RB call, the current transaction was set forward or reset and the present situation does not allow the current transaction or the follow-up transaction to be set forward.

This situation is indicated to the program via the KCTARB field in the KB header:

**KCTARB**      In an TP service this indicates whether a situation has occurred which makes it necessary to reset the transaction.

Blanks

No situation has occurred which makes it necessary to reset the transaction.

Y

A situation has occurred which makes it impossible to set the transaction forward. Communication with partner services is still permitted. A call to set the transaction forward results in an abnormal end of service.

## PTDA Write to TLS

You use the PTDA (put data) call to write a block from a specified storage area to a terminal-specific long-term storage area (TLS) of an LTERM/ LPAP/ OSI-LPAP partner.

A program unit run of a dialog service can only write to blocks of its “own” TLS, i.e. blocks of the LTERM/ LPAP/ OSI-LPAP partner, via which the service was started.

A program unit run of an asynchronous service can write to the blocks of any LTERM/ LPAP/ OSI-LPAP partner of the UTM application.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area			
	KCOP	KCLA	KCRN	KCLT
Write to a TLS block (in the dialog program)	"PTDA"	Length	Block name	—
Write to TLS (in the asynchronous program)	"PTDA"	Length	Block name	LTERM/ LPAP/ (MASTER-)OSI-LPAP Name

### Setting the 2nd parameter

Here you have to supply the address of the message area which contains the message to be written.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"PTDA"
2.	KCLA	Length in bytes
3.	KCRN	Block name
4.	KCLT	Name of LTERM/ LPAP/ OSI-LPAP partner
	Message area	
5.		Data

**KDCS call**

	1st parameter	2n parameter
6.	KDCS parameter area	Message area

7. **C/C++ macro call**

Macro name	Parameters
KDCS_PTDA	(nb,kcla,kcrn,kclt)

**openUTM return information**

	Field name in the KB return area	Contents
8.	KCRCCC	Return code
9.	KCRCDC	Internal return code

For the PTDA call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the "PTDA" operation code.
2. In the **KCLA** field, the length of the data which openUTM is to write to the TLS. The length specified here becomes the new length of the TLS block.
3. In the **KCRN** field, the name of the TLS block to which openUTM is to write the data.
4. only for asynchronous programs:  
in the **KCLT** field, the name of the LTERM/ LPAP/ OSI-LPAP partner containing the TLS to which openUTM is to write data (this field is not evaluated by dialog programs).

In the message area you enter:

5. the message which you want to write to the TLS

You specify the following for the KDCS call:

6. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area from which openUTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLA.
7. The use of C/C++ calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

8. in the **KCRCCC** field: the KDCS return code, see below.
9. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").

### KDCS return codes in the KCRCCC field

The following codes can be analyzed in the program:

- 000 Function carried out.
- 40Z System cannot perform the operation (generation error or system error, deadlock, timeout), see KCRCDC.
- 41Z Call issued in the first segment of the sign-on service, although this is not allowed by the generation.
- 43Z Length entry in KCLA is negative or invalid.
- 44Z Name of the block in KCRN is unknown or invalid.
- 46Z LTERM/ LPAP/ (MASTER-)OSI-LPAP name in KCLT is invalid (only for asynchronous programs).
- 47Z Message area missing or cannot be accessed in the specified length.

An additional return code can be found in the dump:

- 71Z No INIT issued in this program.

### Features of the PTDA calls

- At the end of transaction (PEND RE/FI/FC/SP), the changes made to the TLS block are carried out and the block is unlocked. Other transactions can then make use of it again. With PEND RS/ER/FR or RSET the changes made to the TLS blocks are cancelled and the blocks are unlocked.
- The lock may apply for a longer period in the following cases:
  - PEND KP and PGWT KP
  - PEND PA/PR with a task change due to TAC class control
  - PEND PA/PR with waiting for a DGET message
- A PTDA call locks access to a TLS block until the next synchronization point. No other TLS blocks of the addressed LTERM/ LPAP/ OSI-LPAP partner are locked.

Note that the current length of a TLS block is the length in which it was written with the last PTDA call.

How UTM reacts when the desired TLS block is locked is described in the [section “Action with locked storage areas \(TLS, ULS and GSSB\)” on page 88](#).

## QCRE Create temporary queue

The QCRE (Queue CREate) call is used to create a temporary queue dynamically.

The prerequisite for a successful QCRE call is that enough table spaces for QUEUE objects must have been reserved at generation by means of the QUEUE statement.

In the QCRE call you can either assign a name for the queue to be created or specify that openUTM assigns a name automatically, which is then entered in the KCRQN (queue name) return field.

openUTM creates queue names that follow on from each other from printable digits. If the queue names are assigned by openUTM, the same queue name is not used again for 100 million QCRE calls. This ensures that long-running services for communication do not inadvertently use a temporary queue whose name has been reassigned after being deleted.

The format of the QCRE call is described in detail below. You will find more information on the subject of “message queuing” in [section “Message Queuing \(asynchronous processing\)” on page 50](#).

### Setting the KDCS parameter area (1st parameter)

The following table shows the entries required in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area					
	KCOP	KCOM	KCRN	KCLA	KCMF/kcfn	KCQMODE
Create queue without name	"QCRE"	"NN"	Blanks	Queue level	Blanks	"S"/"W"/ binary zero
Create queue with name	"QCRE"	"WN"	Queue name	Queue level	Blanks	"S"/"W"/ binary zero

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"QCRE"
2.	KCOM	"NN"/ "WN"
3.	KCRN	Blanks/name of the queue
4.	KCLA	Queue level of the queue
5.	KCMF/kcfn	Blanks
6.	KCQMODE	"S"/"W"/binary zero

**KDCS call**

	1st parameter	
7.	KDCS parameter area	

8. **C/C++ macro call**

Macro name	Parameters
KDCS_QCRENN	(kcla,qmode)
KDCS_QCREWN	(kcla,kcrn,qmode)

**openUTM return information**

	Field name in the KB return area	
9.	KCRQN	Name assigned by openUTM
10.	KCRCCC	Return code



In the KDCS parameter area you make the following entries for the QCRE call:

1. In the **KCOP** field, enter the QCRE operation code.
2. In the **KCOM** field:
  - NN (no name) if openUTM is to create the name of the queue automatically
  - WN (with name) if you assign the name yourself
3. In the **KCRN** field, enter the name of the queue (KCOM=WN) or blanks (KCOM=NN). A name you assign must not begin with a digit and must adhere to the conventions for generatable names. In other words, it can consist only of the characters A...Z, a...z, 0...9, \$, #, @. If necessary, it must be filled with blanks.
4. In the **KCLA** field, enter the queue level. In other words, you enter the maximum number of messages that can be stored in this queue.  
If you specify zero, openUTM uses the value or default value of the QLEV parameter from the QUEUE statement of the generation.
5. The **KCMF/kcfn** field must be supplied with blanks.
6. In the **KCQMODE** field:
  - S (standard) if further messages are to be rejected when the queue level is reached
  - W (wrap) if a new message overwrites the oldest existing message when the queue level is reached
  - Binary zero  
openUTM uses the value or default value of the QMODE parameter from the QUEUE statement of the generation.

In the KDCS call, specify:

7. as the 1st parameter: the address of the KDCS parameter area.
8. How to use macro calls for C/C++ is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

9. in the **KCRQN** field the name generated automatically (when KCOM=NN).
10. in the **KCRCCC** field the KDCS return code (see next page).

**KDCS return codes for the QCRE call**

The following can be evaluated in the program:

000 The operation was executed.

16Z KCOM=WN: The queue name already exists.

KCOM=NN: openUTM could not find a free name. In this case, you can try to find a free queue name with another QCRE NN call.

*Note on name assignment:*

openUTM assigns names consisting only of digits and remembers the last name it assigned. When QCRE is specified with KCOM=NN, openUTM searches the next 100 names consisting of digits for a free entry. If these names are all occupied, openUTM terminates the search with 16Z. At the next QCRE NN call, the next 100 names are searched.

40Z The operation cannot be executed:

- because there is no free table space left for temporary queues (what to do: increase the value for NUMBER in the QUEUE statement and regenerate, or use QREL to delete temporary queues that are no longer required).
- because there is no free table space left in the process-specific buffer for restart data (what to do: increase the value of MAX RECBUF=(..., *length*) and regenerate).

42Z The value in KCOM is invalid.

43Z The value in KCLA (queue level) is negative or invalid.

44Z The queue name begins with a digit (KCOM=WN), or KCRN does not contain any blanks (KCOM=NN).

45Z KCMF/kcfn was not supplied with blanks.

46Z The value in KCQMODE is invalid.

49Z Unused fields have a value other than binary zero.

71Z An INIT has not yet been called in the program unit run.

**Features of the QCRE call**

- No administration authorization is required to create a temporary queue.
- If the queue names are assigned by openUTM, they are not used again for 100 million QCRE calls.
- If a temporary queue is created with QCRE, messages can be written in this queue in the same transaction. However, these messages cannot be read and administered until the transaction is successfully completed.
- In the case of UTM-S, temporary queues and their messages are preserved after the end of the application run until they are deleted explicitly by means of a QREL call. In the case of UTM-F, temporary queues are deleted automatically at the end of the application run. All the messages still stored in the queue are lost.

## QREL Delete temporary queue

The QREL (Queue RELEASE) call is used to delete a temporary queue dynamically. All the messages in the queue are deleted, and the name and the table space of the queue are made available.

Services that wait for DGET messages of this queue are continued.

The format of the QREL call is described in detail in the following. You will find more information on the subject of message queuing in [section "Message Queuing \(asynchronous processing\)" on page 50](#).

### Setting the KDCS parameter area (1st parameter)

The following table shows the entries required in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area			
	KCOP	KCOM	KCRN	KCMF/kcfn
Delete temporary queue	"QREL"	"RL"	Name of the queue	Blanks

### Setting the parameters

	Field name in the KDCS parameter area	Contents
1.	KCOP	"QREL"
2.	KCOM	"RL"
3.	KCRN	Name of the queue
4.	KCMF/kcfn	Blanks

### KDCS call

	1st parameter	
5.	KDCS parameter area	

### C/C++ macro call

	Macro name	Parameters
6.	KDCS_QRELRL	(kcrn)

### openUTM return information

	Field name in the KB return area	
7.	KCRCCC	Return code

In the KDCS parameter area, make the following entries for the QREL call:

1. In the **KCOP** field, enter the QREL operation code.
2. In the **KCOM** field, enter the RL modifier.
3. In the **KCRN** field, enter the name of the queue to be deleted.
4. The **KCMF/kcfn** field must be supplied with blanks.

For the KDCS call, specify:

5. as the 1st parameter: the address of the KDCS parameter area.
6. How to use macro calls for C/C++ is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

7. in the **KCRCCC** field the KDCS return code (see below).

### **KDCS return codes for the QREL call**

The following can be evaluated in the program:

- 000 The operation was executed.
- 40Z There is no more space in the process-specific buffer for restart data. Bottlenecks can result because openUTM executes a separate DADM call for each message read by means of DGET in a transaction that is not yet completed and has to write a processing item to the buffer.  
What to do: Increase the value of MAX RECBUF=(..., *length*) and regenerate.
- 42Z The value in KCOM is invalid.
- 44Z There is no temporary queue with the name specified in KCRN.
- 45Z KCMF/kcfn was not supplied with blanks.
- 49Z Unused fields (except KCMF) have a value other than binary zero.
- 71Z An INIT has not yet been called in the program unit run.

### **Features of the QREL call**

- No administration authorization is required to delete a temporary queue.
- After the QREL call, messages in the deleted queue can no longer be read or administered. New messages cannot be created for this queue.
- After a QREL call and the successful conclusion of the transaction, a new temporary queue can be created with the same name.

## **RSET Reset transaction**

You use the RSET (reset transaction) call to reset changes and operations of local transactions. Open database transactions are also reset. All output operations since the last local synchronization point are cancelled. Control is returned to the program unit. The program unit is continued after the RSET call. Further KDCS calls and database calls are subsequently possible (except INIT).

You can use the RSET call to react to application errors with specific actions. You can reset a transaction and at the same time pass control back to the application program.

This approach is useful for errors which are not program errors (e.g. as a response to error codes  $\geq 40Z$ ). You can react specifically in the program unit, e.g. by

- sending a message to the appropriate client or to the administrator (MPUT)
- writing an item of logging information (LPUT)
- sending an output job, e.g. to a printer (FPUT/DPUT)

The RSET call may, for example, also be useful in the event that database accesses yield unexpected return codes (e.g. “data record does not exist”) and UPDATE operations have already been performed.

### **Setting the 1st parameter (KDCS parameter area)**

For the RSET call, you only have to enter the “RSET” operation code in the KCOP field.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"RSET"

**KDCS call**

	1st parameter	2nd parameter
2.	KDCS parameter area	—

**C/C++ macro calls**

	Macro name	Parameters
3.	KDCS_RSET	()

**openUTM return information**

	Field name in the KB return area	Contents
4.	KCRCCC	Return code
5.	KCRCDC	Internal return code

For the REST call you make the following entries in the KDCS parameter area:

1. in the **KCOP** field, the RSET operation code.
- openUTM does not evaluate any other operands of the parameter area.

You specify the following for the KDCS call:

2. 1st parameter: the address of the KDCS parameter area.
3. The use of C/C++ calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

4. in the **KCRCCC** field: the KDCS return code.
5. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual "Messages, Debugging and Diagnostics").



### KDCS return codes for the RSET call

The following codes can be analyzed in the program:

000 Operation carried out

Additional return codes can be found in the dump:

70Z System cannot perform the operation (generation error or system error), see KCR CDC.

71Z INIT missing in this program.

### Features of the RSET call

- All resources occupied by this transaction up to the RSET call are released.
- All service-specific data is reset to the last synchronization point. In this way, the KB program area, all LSSBs and GSSBs and TLS and ULS blocks are again made available with their original contents.
- Data in SPAB and in program-specific working storage areas is not changed.
- An open DB transaction is reset.
- Each reset of a DB transaction has the same effect as an RSET call, and thus implicitly causes the UTM transaction to be reset.
- The program unit again regains control following the RSET call and continues the program run with the next statement after the RSET call. It is then possible to issue additional KDCS (except INIT) and DB calls in the program unit.
- After this, it is no longer possible to read a dialog input message which was read prior to the RSET call, but it is possible to read a dialog input message which was not yet read prior to the RSET call.
- Input messages read with FGET or DGET can again be read following an RSET. With DGET messages this is possible only if the maximum number of redeliveries specified at generation has not been reached. For more information refer to the openUTM manual "Generating Applications", REDELIVERY operand in the MAX control statement.

If the maximum number of redeliveries has been reached then the message is either deleted or saved in the dead letter queue by UTM (only possible in the case of messages to a TAC queue), see openUTM manual "Generating Applications", DEAD-LETTER-Q operand in the TAC statement.

### Features of the RSET call with distributed processing

The behavior of openUTM following an RSET call in a program unit run belonging to a distributed transaction is governed by the RSET generation parameter of the UTMD statement (see the openUTM manual "Generating Applications"):

- If RSET=LOCAL is generated, then the RSET call has no effect on the distributed transaction.  
Here, inconsistencies may occur in the distributed databases, if some of the local transactions participating in the distributed transactions are continued and others reset. With this generation, the global data consistency is no longer guaranteed by the relevant system components, but is the responsibility of the application program units. These must decide the situations in which it is more practical to terminate the distributed transaction and the situations in which it must be reset.
- If RSET=GLOBAL is generated, then openUTM forces termination of the program unit run with a PEND variant which causes the distributed transaction to be reset (see also the PEND call, [page 372ff](#)).

## SGET Read from secondary storage area

You use the SGET (storage get) call to read data from a secondary storage area into a storage area of the program unit. The following may occur as secondary storage areas:

- the global secondary storage area (GSSB)
- the local secondary storage area (LSSB)
- the user-specific long-term storage area (ULS)

If an LSSB is no longer required, it can be deleted at the same time by entering KCOM=RL. The contents of a ULS can only be deleted by writing (SPUT) with KCLA=0.

A GSSB must be deleted with a separate call (SREL); it remains locked until the end of the transaction or service. For further information, see the description of the SREL call.

A GSSB or ULS can be unlocked explicitly with the UNLK call.

In UTM cluster applications, GSSB or ULS areas are available throughout the cluster.

### Setting the KDCS parameter area (1st parameter)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area				
	KCOP	KCOM	KCLA	KCRN	KCUS
Read from LSSB	"SGET"	"KP"	Length	Name of the LSSB	—
Read from LSSB and delete LSSB	"SGET"	"RL"	Length	Name of the LSSB	—
Read from GSSB (and lock GSSB)	"SGET"	"GB"	Length	Name of the GSSB	—
Read from ULS (and lock ULS)	"SGET"	"US"	Length	Block name	User ID/ LSES name/ association name/ blanks

For KCOM = US, all the fields not used in the KDCS parameter area are to be set with binary zero.

### Setting the 2nd parameter

Here you have to supply the address of the message area into which openUTM is to read the message.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"SGET"
2.	KCOM	"KP"/"RL"/"GB"/"US"
3.	KCLA	Length in bytes
4.	KCRN	Name of the area
5.	KCUS	User ID/blanks/ LSES name/association name

**KDCS call**

	1st parameter	2nd parameter
6.	KDCS parameter area	Message area

**C/C++ macro calls**

	Macro names	Parameters
	KDCS_SGETKP / KDCS_SGETRL / KDCS_SGETGB	(nb,kcla,kcrn)
	KDCS_SGETUS	(nb,kcla,kcrn,kcus)

**openUTM return information**

	Message area	Contents
8.		Data
	Field name in the KB return area	
9.	KCRLM	Actual block length
10.	KCRCCC	Return code
11.	KCRCDC	Internal return code

For the SGET call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the **SGET** operation code.
2. In the **KCOM** field:
  - KP (keep) to read from an LSSB - the area is retained
  - RL (release) to read from and delete an LSSB
  - GB to read from a GSSB
  - US to read a ULS block
3. In the **KCLA** field, length of data to be transferred to the message area.
4. In the **KCRN** field, name of the LSSB/GSSB or of the ULS block to be read from.
5. In the **KCUS** field, user ID if a ULS block of a foreign user ID is to be read, otherwise blanks (if blanks are specified, the ULS block of the user who started the service is read). If a foreign user ID is entered in KCUS, your own user ID must have administration privileges.

If you want to read a ULS block of a remote session/association, you have to specify its LSES name or association name.

Irrelevant for KCOM = KP/RL/GB.

You specify the following for the KDCS call:

6. 1st parameter: the address of the KDCS parameter area.
  - 2nd parameter: the address of the message area to which UTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLA.
7. The use of C/C++ calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

8. the desired data in the specified message area.
9. in **KCRLM**, the actual length of the data in the LSSB/GSSB/ULS (in bytes). This enables you to detect deviations from the KCLA entry (important if specified KCLA value is smaller). Exception: for KCLA = 0 UTM always returns KCRLM = 0.
10. **KCRCCC**: the KDCS error code, see next page.
11. **KCRCKZ** and **KCRCDC**: the identifier and the internal error code of UTM (see the openUTM manual "Messages, Debugging and Diagnostics").

**KDCS return codes for the SGET call**

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 14Z No area exists with the name specified in KCRN (only for KP, RL, GB).
- 40Z System cannot perform the operation (generation error or system error, deadlock, timeout); see KCRCDC.
- 41Z The call was issued in the first segment of the sign-on service although this is not allowed by the generation.  
For KCOM=US: The call was issued in the first segment of the sign-on service or in the sign-on service after a SIGN ON and before the PEND PS call.
- 42Z Entry in KCOM is invalid.
- 43Z Length entry in KCLA negative or invalid.
- 44Z Name in KCRN invalid. It is invalid if it consists solely of blanks or binary zero or has not been generated (in the case of ULS).
- 46Z Entry in KCUS is invalid.
- 47Z Message area missing or cannot be accessed in the specified length.
- 49Z Contents of fields not used in the KDCS parameter area not equal to binary zero (only for KCOM = US).

An additional error code can be found in the dump:

- 71Z INIT missing in this program.

### Features of the SGET call

- The area is transferred in its actual length, but at the most in the length specified in KCLA. The actual length of the data in the GSSB, LSSB or ULS is returned in the KCRLM field.
  - If the length specified in KCLA is smaller than the actual length of the record to be read, the data is truncated at the right-hand side. You can capture this situation in a program unit (KCLA < KCRM).
  - If the length specified in KCLA is greater than the actual length of the record to be read (KCLA > KCRLM), the surplus part of the message area is undefined after the SGET call.
- If an attempt is made in a program unit to perform an SGET read operation on a non-existent storage area, the program unit will receive the error code 14Z ("No area with this name exists").
- If SGET is used to access a GSSB or ULS, the following applies:
  - An SGET call for a GSSB or ULS locks the GSSB or ULS until the next synchronization point or reset point, i.e. until PEND SP/RE/FI/FC/RS/ER/FR or RSET. If, following SGET, the processing step is terminated in a program unit with a PEND KP, PGWT KP, PGWT PR or with a PEND PA/PR call with a task change due to TAC class control or a wait for a DGET message, the access lock remains until the next synchronization point, unless the lock is cancelled beforehand with an UNLK call.
  - Reading from a non-existent GSSB has the same effect as creating a GSSB with simultaneous deletion (SPUT, SREL sequence).
    - The name of this GSSB remains locked until the next synchronization point or reset point.
    - If the generated maximum number of GSSBs is already reached, the program unit receives the return code 40Z with KCRCDC K804.

In a UTM cluster application, reading from a non-existent GSSB requires four additional file accesses (to increment and decrement the GSSB counter). Consequently, in UTM cluster applications, it is advisable for performance reasons not to use any empty GSSBs but instead, for instance, a GSSB of length 1 for the serialization of program units, for example.

In [section “Action with locked storage areas \(TLS, ULS and GSSB\)” on page 88](#) there is a description of how openUTM reacts when the desired GSSB or ULS block is locked.
- When an SGET is used to access an LSSB, the following applies:
  - SGET KP causes the LSSB to remain available in the follow-up transaction of the service, i.e. after the subsequent PEND RE/SP.

- SGET RL reads the LSSB and deletes it at the end of transaction (i.e. with PEND RE/SP/FI/FC/RS/ER/FR). Attempts to access in the meantime are rejected with 14Z.  
You should always use this variant if the LSSB is no longer needed after reading in the current service.



## SIGN Control sign-on and sign-off, check authorization data, change passwords

You use the SIGN (sign on) call to

- query the status of the sign-on service in the sign-on service or transfer the authorization data to openUTM
- change the password for the current user ID
- have the authorization data checked
- initiate in the program the effect of the commands KDCOFF and KDCOFF BUT.

The SIGN call is only allowed in dialog program units (exception: SIGN CK).

**B** In B2000/OSD you can additionally use the SIGN CL (Change Locale) call to change the location of the current user ID. This call is described as of [page 428](#).

### Setting the KDCS parameter area (1st parameter)

The table below shows the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area			
	KCOP	KCOM	KCLA	KCUS
Query status of the sign-on service	"SIGN"	"ST"	48	Binary zero
Transfer authorization data to openUTM	"SIGN"	"ON"	8	User ID
Change password	"SIGN"	"CP"	16	Binary zero
Check authorization data (without sign-on)	"SIGN"	"CK"	8	User ID
Initiate effect of the KDCOFF command	"SIGN"	"OF"	0	Binary zero
Initiate effect of the KDCOFF BUT command	"SIGN"	"OB"	0	Binary zero

All the fields not used in the KDCS parameter area have to be set to binary zero.

### Setting the 2nd parameter

Here you have to supply the address of the message area from which openUTM is to read the data.

### Setting the parameters

	Field name in the KDCS parameter area	Contents
1.	KCOP	"SIGN"
2.	KCOM	"ST"/"ON"/"CP"/"CK"/"OF"/"OB"
3.	KCLA	48/8/16/0
4.	KCUS	User ID/binary 0
	Message area	
5.		Data/ -

### KDCS call

	1st parameter	2nd parameter
6.	KDCS parameter area	Message area

### 7. C/C++ macro calls

Macro names	Parameters
KDCS_SIGNST / KDCS_SIGNOF / KDCS_SIGNOB	(nb)
KDCS_SIGNON / KDCS_SIGNCK	(nb,kcla,kcus)
KDCS_SIGNSTLA / KDCS_SIGNCP	(nb,kcla)

### openUTM return information

	Field name in the KB return area	Contents
8.	KCRSIGN1	Sign-on status
8.	KCRSIGN2	Additional information
9.	KCRUS	Name of user ID
10.	KCRCCC	Return code
11.	KCRCDC	Internal return code
12.	KCRMF/kcrfn	Format identifier of start format/blanks
13.	KCRLM	Validity period of password

For the SIGN call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the SIGN operation code.
2. In the **KCOM** field
  - ST query status of the sign-on service
  - ON check authorization data (with sign-on)
  - CP change password
  - CK check authorization data (without sign-on)
  - OF initiate effect of the KDCOFF command
  - OB initiate effect of the KDCOFF BUT command.
3. In the **KCLA** field
  - 48 for KCOM=ST:  
This is the length of the message area into which openUTM transfers the information. openUTM provides a data structure for structuring of the message area, see description on [page 425](#). Specify the version number of the structure in the data structure header.
  - 8 for KCOM = ON/CK
  - 16 for KCOM = CP
  - 0 for KCOM = OF/OB
4. In the **KCUS** field, the user ID if, with KCOM = ON/CK, authorization data is to be transferred to UTM. Enter binary zero for all other variants.

In the message area you have to enter:

5. the data you wish to transfer to openUTM or receive from openUTM.  
The password (8 characters) is made available for KCOM = ON/CK, the old and the new password (16 characters) are made available for KCOM = CP.  
For KCOM = ST, the following information is exchanged when KCLA > 0 (maximum of 48 characters):
  - the desired version of the data structure is passed (2 characters)
  - data for the sign-on service (e.g. validity period of the password) is returned.

You specify the following for the KDCS call:

6. 1st parameter: the address of the KDCS parameter area  
2nd parameter: the address of the message area from which UTM is to read the data. You enter the address of the message area even if you have entered the length 0 in KCLA.
7. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

8. Additional information in the **KCRSIGN1** and **KCRSIGN2** fields (nothing is entered unless KCRCCC = 000):
  - for KCOM = ST, the current status of the sign-on procedure, see table on [page 421](#).
  - for KCOM = CK, the result of the check, see table on [page 423](#).
9. in the **KCRUS** field, the name of the user ID if:
  - KCRSIGN1=U, i.e. it was not possible to sign-on the user ID successfully, or
  - KCRSIGN1=I, i.e. sign-on was not terminated successfully, an intermediate dialog must be performed for the user
10. in the **KCRCCC**, the KDCS return code, see [page 424](#).
11. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).
12. in the **KCRMF/kcrfn** field, for KCOM = ST the format identifier of the start format or blanks if no start format was generated. When used with USER and if the sign-on was successful, the identifier of the user-specific start formats is returned. If sign-on has not yet been completed successfully or if the application is generated without USER, then KCRMF/kcrfn contains the identifier of the LTERM-specific start format.
13. in the **KCRMLM** field, for KCOM = ST:
  - for KCRCCC < 40Z: KCRMLM contains the length of the information actually available in openUTM.
  - for KCRCCC >= 40Z, 0 is returned.

**SIGN ST call return information in the fields KCRSIGN1 and KCRSIGN2**

For SIGN ST, openUTM delivers the following information about the current status of the sign-on procedure in fields KCRSIGN1 and KCRSIGN2:

KCRSIGN1	KCRSIGN2	Meaning
C	01	Connection established (as K002)
	02	KDCOFF BUT command issued (as K018)
	03	KDCOFF BUT issued from program

KCRSIGN1	KCRSIGN2	Meaning
U	01	Specified USER not generated (as K004)
	02	Specified USER locked (as K005)
	03	Someone already signed on with this USER (as K007)
	04	Specified old password is invalid (as K006)
	05	Entries for new password unusable
	06	Card information is invalid (as K031)
	07	Sign-on not possible at present because of resource bottleneck or no
	08	further users can sign on at present, since the maximum possible number of simultaneous users has been reached, or it was not possible to change the password, since an inverse KDCDEF is currently running
	09	Sign-on not possible because of missing Kerberos support (as K110).
	10	The current LTERM is not authorized to continue the service (as K123)
	11	The period of validity for the password has been exceeded (as K120)
	12	The new password does not fulfill the requirements of the complexity level generated (as K097)
	13	The new password is too short (as K097)
	14	The password transferred by KDCUPD does not fulfill the requirements of the complexity level generated (as K125)
	15	A transaction restart is required for the specified user ID (as K145)
	16	The open service cannot be continued from within this LTERM partner (like K123)
	17	The administrator issued a SHUT WARN; Normal users cannot sign on any more to the application (as K016); administrators can still sign on.
	18	The encryption mechanism required for the continuation of the open service is not available in the connection (as K123)
	20	Error in the Kerberos authentication (as K108)
	21	Invalid Kerberos principal (as K109)
	22	The specified USER does not exist in the cluster user file (as K004).
	23	Somebody has already signed onto a different node application under this USER (as K007).
	24	It is currently not possible to sign on because the cluster user file could not be locked in the generated time (CLUSTER statement, parameter FILE-LOCK-TIMER-SEC, parameter FILE-LOCK-RETRY) (as K091).
	25	It is not possible to sign on at this node application because the user has a service that is bound to another node application and which may not be terminated (as K189).
	26	Sign-on rejected because the user's open service has a transaction in PTC state but no service restart has been requested.
	I	01

B

B  
B

KCRSIGN1	KCRSIGN2	Meaning
A	01	Sign-on successful because generated without USER (as K001)
	02	Sign-on successful (as K008)
	03	Sign-on successful (via distributor, <b>only in BS2000/OSD</b> )
	04	Sign-on successful (by the user of the connection, only for TS or UPIC clients). A real user can sign on by calling SIGN ON.
	05	Sign-on successful, password changed via intermediate dialog
	06	Sign-on successful (via distributor, <b>only in BS2000/OSD</b> ), the password was changed via distributor
R	01	as A01, but after service restart
	02	as A02, but after service restart
	03	as A03, but after service restart
	04	as A04, but after service restart
	05	as A05, but after service restart
	06	as A06, but after service restart

KCRSIGN1 provides a rough classification

- C Connected but not signed on. Status following connection setup or KDCOFF BUT
- U Signon Unsuccessful. A preceding attempt to sign on was rejected.
- I Signon Incomplete. Intermediate dialog is required to obtain additional data (password, ID, chipcard)
- A Signon Accepted. Without subsequent service restart
- R Signon accepted + Restart. With subsequent service restart.

### Return information of the SIGN CK call in the KCRSIGN1 and KCRSIGN2 fields

openUTM returns the following information for SIGN CK in the KCRSIGN1 and KCRSIGN2 fields:

KCRSIGN1	KCRSIGN2	Meaning
A	02	Authorization data is correct and complete
U	01	The specified USER does not exist
	02	The specified USER is locked
	04	The specified old password is incorrect
	07	The card information is not available
	11	The validity period of the password has expired
	14	The password transferred by KDCUPD does not satisfy the complexity level requirement or is too short

**KDCS return codes for the SIGN call**

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 01Z The function was executed, the message area provided is too short, though (the value in KCLA is too small). No information or incomplete information was returned.
- 40Z System cannot perform the operation (generation error or system error).
- 41Z Call is not allowed at this point:
  - SIGN OF/OB call already issued, or
  - SIGN call in an asynchronous service and call is not SIGN CK , or
  - SIGN ST/ON call outside a sign-on service, or
  - SIGN CP/CK call prior to successful sign-on, or
  - SIGN ON/CP/CK call in an application which was generated without user IDs, or
  - SIGN OB/OF in a job-receiving service (with distributed transaction processing).
- 42Z Entry in KCOM is invalid.
- 43Z Length entry in KCLA is negative or invalid.
- 44Z For KCOM=CP: entry for old password incorrect, password not changed
- 45Z For KCOM=CP: entry for new password incorrect, password not changed. The more precise cause provides KCRCDC.
- 47Z Message area missing or cannot be accessed in the specified length.
- 48Z For KCOM=ST: invalid interface version
- 49Z Contents of fields not used in the KDCS parameter area not equal to binary zero.

Two additional return codes can be found in the dump:

- 71Z INIT call still missing in the program unit run.



## Features of the SIGN call

- Message area for SIGN ST with KCLA > 0:

Field name COBOL	Field name C/C++	Length in bytes	Description
Call information:			
KCVER	if_version	2	Version number of the data structure ( 3 )
Return information:			
KCRPWVAL	rpwval	2	Validity period of the password
KCRPWMIN	rminpw	2	Minimum validity period of the password
KCRUSER	ruser	8	User ID
KCRTAC	rtac	8	Transaction code from the UPIC protocol
KCRPSWRD	rpsword	8	Password from the UPIC protocol
KCLSTSGN	rlstsgn	14	Date/time of the last sign-on
KCDSPMSG	rdispmsg	1	Message present (Y/N)
KCTAPTC	rtainptc	1	Transaction in PTC state (Y/N)
KCCLNODE	rclusternode	8	Node to which the open service is bound
KCSGRES	reserved	10	Reserved for future extensions

The following mean:

- KCVER** Version number of the data structure. The number 2 is to be entered here for this version of openUTM.
- KCRPWVAL** If the sign-on was successful (KCRSIGN1 = A/R) or has not yet been successful (KCRSIGN1 = I), then this field contains the number of days that the password for this user is still valid.  
 The value -1 means that there was no validity period generated for the password.  
 The value 0 means that the password will become invalid within the next 24 hours.  
 The value -2 means that validity of the password has run out. The password must be changed if the sign-on service is to terminate successfully. This values can only be returned if grace sign-ons are permitted (SIGNON statement in the generation, parameter GRACE = YES).  
 In the case of KCRSIGN1 = I, the value -3 means that the complexity level or minimum length of the password has been increased and that the password transferred with the KDCUPD tool may possibly no longer meet the requirements.

Otherwise the value -3 means that the password passed by the KDCUPD tool does not meet the complexity level requirements or is too short. The password must be changed with SIGN CP if the sign-on service is to terminate successfully.

This values can only be returned if grace sign-ons are permitted (SIGNON statement in the generation, parameter GRACE = YES).

KCRPADMIN	<p>If the sign-on is successful (KCRSIGN1 = A/R) or has not yet been successful (KCRSIGN1 = I), then this field contains the number of days that the password for this user may not be changed using a SIGN CP call.</p> <p>The value 0 means that the password may be changed.</p>				
KCRUSER	<p>After an unsuccessful or not yet completed sign-on (KCRSIGN1 = U/I), this field contains the name of the user that was rejected (KCRSIGN1 = U) or for whom an intermediate dialog must be executed first (KCRSIGN1 = I), otherwise it contains blanks.</p>				
KCRPTAC	<p>This field contains the name of the transaction code (TP_Name) passed in the UPIC protocol in the sign-on service for the UPIC partner, otherwise it contains blanks. openUTM does not check if the transaction code is valid or not.</p>				
KCRPSWRD	<p>This field contains the password of a user generated without a password that was passed in the UPIC protocol in the sign-on service for the UPIC partner after a successful sign-on (KCRSIGN1 = A/R), otherwise it contains blanks.</p>				
KCLSTSGN	<p>This field contains the date and time of the last successful sign-on of this user to the application after the user has successfully signed on (KCRSIGN1 = A/R) or has not yet signed on successfully (KCRSIGN1 = I). The date and time are passed in the format YYYYMMDDHHMMSS. Printable nulls are returned after the first successful sign-on after a regeneration.</p>				
KCDSPMSG	<p>After the successful sign-on (KCRSIGN1 = A/R) of a user, the field contains the following value:</p> <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 10px;">Y</td> <td>If there is an open dialog service for the user (KCRSIGN1=R) or a dialog message that can be output with MPUT PM</td> </tr> <tr> <td>N</td> <td>in all other cases</td> </tr> </table>	Y	If there is an open dialog service for the user (KCRSIGN1=R) or a dialog message that can be output with MPUT PM	N	in all other cases
Y	If there is an open dialog service for the user (KCRSIGN1=R) or a dialog message that can be output with MPUT PM				
N	in all other cases				
KCTAPTC	<p>When the user has successfully signed on with a subsequent service restart (KCRSIGN1 = R), the field has the following value:</p> <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 10px;">Y</td> <td>If there is a transaction in the state P(repare) T(o) C(ommit) for the user. In this case, the service restart cannot be prevented.</td> </tr> </table>	Y	If there is a transaction in the state P(repare) T(o) C(ommit) for the user. In this case, the service restart cannot be prevented.		
Y	If there is a transaction in the state P(repare) T(o) C(ommit) for the user. In this case, the service restart cannot be prevented.				

N in all other cases.

**KCCLNODE** If the sign-on was not successful (KCRSIGN1 = U) due to the existence of a service bound to another node application (KCRSIGN2 = 25), the field contains the host name of the node to which the open service is bound.

- Entries in the message area for SIGN ON and SIGN CP
  - For SIGN ON, write an 8 byte password to the message area. Blanks mean "user ID without password".
  - For SIGN CP, write the old and new passwords at 8 byte length to the message area as follows:

old password <sup>1</sup>	new password <sup>1</sup>
---------------------------	---------------------------

<sup>1</sup> Blanks mean "user ID without password"

If the call has the correct syntax, then openUTM overwrites the data area with blanks.

You do not need administration authorization for this call.

- With SIGN ON openUTM checks whether the user is able to sign on from this client at this time.
- With SIGN CK openUTM checks whether the authorization data is adequate for successful sign-on from this client but does not check whether sign-on is possible at this time.
- SIGN OF and SIGN OB may only be issued in program units terminated with PEND RE or PEND RE or PEND FI and which issue the dialog message to the terminal, the UPIC client program or the transport system client. Otherwise, openUTM aborts the service with PEND ER.

SIGN OB to UPIC or transport system clients has the same effect as SIGN OF. SIGN OF and SIGN OB only take effect at the next input from the terminal. In other words, the user is not signed off until after the next input (SIGN OB), or the connection to the terminal is not cleared until after the first input (SIGN OF). In the case of UPIC or transport system clients, the connection cleardown is initiated immediately.

- SIGN ST and SIGN ON are only allowed in the sign-on service.



SGN is used as symbolic name instead of the operation code SIGN in the COBOL data structure because SIGN is a reserved COBOL word.

## SIGN CL - Change locale of user ID

- B SIGN CL (Change Locale) allows a terminal user or a UPIC client program (with security
- B function) to change the user-specific locale of the current user ID, i.e. to reset the language
- B identifier, the territory identifier and the name of the employed character set.
  
- B SIGN CL is only permitted in dialog program units of a UTM application which is configured
- B using user IDs. Administration privileges is not necessary.
  
- B If the call is successful, then the new locale is valid as of the next end of transaction. All
- B messages in the current transaction are processed using the old character set name. It is
- B therefore advisable to terminate the program unit at the end of transaction, but without
- B issuing a dialog message to the terminal if the character set name of the user has changed.
  
- B If you only want to change specific components of the location, you have to set binary zero
- B for the remaining components.
  
- B Modifications performed with SIGN CL remain valid after a KDCUPD run. KDCUPD
- B implicitly transfers the current values of a user location to the new KDCFILE.
  
- B SIGN CL is an upwards compatible extension of DIN 66265.

### B Setting the KDCS parameter area (1st parameter) with SIGN CL

Function of the call	Entries in the KDCS parameter area				
	KCOP	KCOM	KCLANGID	KCTERRID	KCCSNAME
Change locale of user ID	SIGN	CL	New language identifier of user ID	New territory identifier of user ID	CCS name of new character set for user ID

- B
- B
- B
- B
- B
- B
- B
- B
- B
- B

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"SIGN"
2.	KCOM	"CL"
3.	KCLANGID	Language identifier of user/ binary zero
4.	KCTERRID	Territory identifier of user/ binary zero
5.	KCCSNAME	Character set name of user/ binary zero

- B
- B
- B

**KDCS call**

	1st parameter	2nd parameter
6.	KDCS parameter area	—

- B
- B
- B

**C/C++ macro call**

	Macro name	Parameters
7.	KDCS_SIGNCL	(nb,kclangid,kcterrid,kcccsname)

- B
- B
- B
- B

**openUTM return information**

	Field name in KB return area	Contents
8.	KCRCCC	Return code
9.	KCRCDC	Internal return code

- B For the SIGN call with CL operation modifier you make the following entries in the KDCS parameter area:  
B
- B 1. In the **KCOP** field, the SIGN operation name.
  - B 2. In the **KCOM** field, the operation modifier  
B CL Change locale of user ID.
  - B 3. In the **KCLANGID** field, the new language identifier which is to be assigned to the user ID from which the service was started. The length of the language identifier is 2 bytes. Enter binary zero in KCLANGID if you do not want to change the language identifier.  
B  
B
  - B 4. In the **KCTERRID** field, the new territory identifier which is to be assigned to the user ID from which the service was started. The length of the territory identifier is 2 bytes. Enter binary zero in KCTERRID if you do not want to change the territory identifier.  
B  
B
  - B 5. In the **KCCSNAME** field, the CCS name of the new character set to be assigned to the user ID. The length of the CCS name is 8 bytes maximum. Enter binary zero in KCCSNAME if you do not want to assign a new character set.  
B  
B
- B You specify the following for the KDCS call:
- B 6. 1st parameter: the address of the KDCS parameter area.
  - B 7. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).  
B
- B openUTM returns:
- B 8. in the **KCRCCC** field, the KDCS return code (length 3 bytes). See below for possible return codes and their meaning.  
B
  - B 9. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).  
B

<b>B</b>	<b>KDCS return codes in the KCRCCC field</b>
<b>B</b> <b>B</b>	000 Operation carried out, the specified component(s) of the location have been changed.
<b>B</b>	40Z The system was not able to perform the operation (generation error).
<b>B</b> <b>B</b> <b>B</b>	41Z SIGN CL is not allowed at this point: <ul style="list-style-type: none"><li>– the call was issued before a successful sign-on</li><li>– SIGN CL was called in an application without user ID.</li></ul>
<b>B</b> <b>B</b>	46Z Specifications for the new locale are incorrect. The location of the user ID is not changed. openUTM returns the precise cause in the KCRCDC field.
<b>B</b> <b>B</b>	49Z When the function was called, unused fields in the KDCS parameter area were not set to binary zero.
<b>B</b>	71Z No INIT call issued in the program unit.

## SPUT Write to secondary storage area

You use the SPUT (storage put) call to write data from a specified area to a

- global secondary storage area (GSSB) or a
- local secondary storage area (LSSB) or a
- user-specific long-term storage area (ULS).

Note that the name of a ULS block is defined at generation (ULS statement for KDCDEF), whereas you can select the names of GSSBs and LSSBs arbitrarily when you call SPUT.

In UTM cluster applications, GSSB or ULS areas are available throughout the cluster.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the calls	Entries in the KDCS parameter area				
	KCOP	KCOM	KCLA	KCRN	KCUS
Write to LSSB	"SPUT"	"DL"/ "MS"/ "ES" (all have same effect)	Length	Name of LSSB	—
Write to GSSB	"SPUT"	"GB"	Length	Name of GSSB	—
Write to ULS	"SPUT"	"US"	Length	Block name	User ID/ LSES name/ association name/ blanks

For KCOM = US all the fields not used in the KDCS parameter area are to be set to binary zero.

### Setting the 2nd parameter

Here you have to supply the address of the message area which contains the message to be written.



**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"SPUT"
2.	KCOM	"GB"/"DL"/"MS"/"ES"/"US" ("DL", "MS" and "ES" have the same effect)
3.	KCLA	Length in bytes
4.	KCRN	Name of the area
5.	KCUS	User ID/LSES name/association name/blanks
	Message area	
6.		Data

**KDCS call**

	1st parameter	2nd parameter
7.	KDCS parameter area	Message area

8. **C/C++ macro calls**

Macro names	Parameters
KDCS_SPUTGB / KDCS_SPUTDL / KDCS_SPUTMS / KDCS_SPUTES	(nb,kcla,kcrn)
KDCS_SPUTUS	(nb,kcla,kcrn,kcus)

**openUTM return information**

	Field name in the KB return area	Contents
9.	KCRCCC	Return code
10.	KCRCDC	Internal return code

For the SPUT call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the SPUT operation code.
2. In the **KCOM** field, the entry indicating
  - whether writing should be to an LSSB ("DL" or "MS" or "ES"), or
  - whether writing should be to a GSSB ("GB"), or
  - whether writing should be to a ULS block ("US").

The entries "MS" and "ES" have the same effect as "DL" for openUTM.

3. In the **KCLA** field, the length of the data which you make available in the message area. The length is not written to the LSSB/GSSB/ULS.
4. In the **KCRN** field, the name of the LSSB/GSSB or ULS block to be initialized or to which the data is to be written. Blanks and binary zero are invalid entries.
5. In the **KCUS** field, the user ID (for KCOM = US), if a ULS block of a foreign user ID is to be written, otherwise blanks. If you enter a foreign user ID in KCUS, your own user ID must have administration privileges.

If you want to write to a ULS block of a remote session/association, you have to specify its LSES name or association name.

For KCOM = DL/MS/ES/GB: irrelevant.

In the message area you have to enter:

6. The message which you want to output.

You specify the following for the KDCS call:

7. 1st parameter: the address of the KDCS parameter area.  
2nd parameter: the address of the message area from which openUTM is to read the message. You enter the address of the message area even if you have entered the length 0 in KCLA.
8. The use of C/C++ calls is described in detail in the [section "C/C++ macro interface" on page 497](#).

openUTM returns:

9. in the **KCRCCC** field: the KDCS return code, see next page.
10. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### **KDCS return codes in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 40Z System cannot perform the operation (generation error or system error, deadlock, timeout), see KCRCDC.
- 41Z Call issued in the first segment of the sign-on service, although this is not allowed by the generation.  
For KCOM=US: the call was issued in the first segment of the sign-on service or after a SIGN ON and before the PEND PS call.
- 42Z Entry in KCOM is invalid.
- 43Z Length entry in KCLA is negative or invalid.
- 44Z Name in KCRN invalid if it consists solely of blanks or binary zero or has not been generated (in the case of ULS).
- 46Z Entry in KCUS is invalid.
- 47Z Message area missing or cannot be accessed in the specified length.
- 49Z Contents of fields not used in the KDCS parameter area not equal to binary zero (only for KCOM = US).

An additional error code can be found in the dump:

- 71Z INIT missing in this program

The following table describes the effect of the call sequences SPUT ... RSET and SPUT ... PEND/PGWT on GSSBs, ULSs and LSSBs.

Call	Effect on GSSBs/ULSs	Effect on LSSBs
SPUT	locks: GSSBs are created if not already present, existing GSSBs are replaced	creates or replaces
... PEND KP ... PGWT KP/PR	leaves resettable and locked	leaves resettable
... PEND RE/SP/ PGWT CM	sets valid (they are then no longer resettable) and unlocks (i.e. other transactions can use them)	sets valid (they are then no longer resettable)
... PEND FI/FC		deletes (they are no longer available)
... RSET/ PEND RS/ PGWT RB	cancels changes and unlocks; a GSSB is deleted if created in this transaction	cancels changes
... PEND ER/FR		deletes

Please note the following features of GSSBs, ULSs and LSSBs:

- A GSSB is available to all program units in an application, i.e. it can be overwritten by all program units. To avoid unintentional overwriting of GSSBs by other program units, you must ensure that their names are unique.
- In UTM cluster applications, GSSB and ULS areas are available throughout the cluster. I.e. any GSSB or ULS that you create/write with SPUT exists in all node applications where it can be read using SGET.
- An LSSB is assigned uniquely to a service.
- GSSBs, LSSBs and ULS blocks always take the length of the last SPUT called. This length cannot exceed 32767 bytes.
- The name of a ULS block is defined at generation (as a TLS block).
- The maximum number of GSSBs or LSSBs is defined at generation

### Features of the SPUT call

- The SPUT call for a GSSB/ULS locks this GSSB or ULS block until the end of the transaction, i.e. until PEND RE, SP, FI, FC, RS or ER/FR.

Following an SPUT call, a GSSB/ULS block remains locked by this transaction; a subsequent PEND KP or PGWT KP call or a PEND PA/PR or PGWT PR (in the case of a wait for a DGET message) retains the lock beyond the end of the processing/dialog step.

Another transaction that wants to process this GSSB/ULS block with SGET, SPUT or SREL will be rejected in the following cases.

- PEND KP and PGWT KP
- PEND PA/PR with a task change due to TAC class control
- PEND PA/PR or PGWT PR with waiting for a DGET message

At the end of a transaction or when there is a reset operation, all locked GSSBs and ULS blocks are released.

In the [section “Action with locked storage areas \(TLS, ULS and GSSB\)” on page 88](#) there is a description of how openUTM reacts when the desired GSSB or ULS block is locked.

- GSSB areas are permitted with length 0 in KCLA. These GSSBs can be used for communication between application programs; all that is evaluated is whether the GSSB is locked or not. However, openUTM deletes a GSSB with length 0 the next time the application is started. KDCUPD likewise does not transfer GSSBs with length 0 to a new KDCFILE (see the openUTM manual “Generating Applications“, changing KDCFILE).
- You can also use a SPUT call with KCLA=0 to delete the contents of ULS blocks.

## SREL Delete secondary storage area

You use the SREL (storage release) call to delete a secondary storage area. A secondary storage area can be

- the global secondary storage area (GSSB)
- the local secondary storage area (LSSB)

Blocks of a ULS (user-specific long-term storage) **cannot** be deleted using SREL, since their names are specified when the application is generated. If you want to delete the contents of a ULS block, you have to overwrite the block with length zero.

In UTM cluster applications, a GSSB is valid throughout the cluster. Its deletion with SREL therefore also applies throughout the cluster.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area		
	KCOP	KCOM	KCRN
Delete LSSB	"SREL"	"LB"	Name of the LSSB
Delete GSSB	"SREL"	"GB"	Name of the GSSB

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"SREL"
2.	KCOM	"LB"/"GB"
3.	KCRN	Name of the area

**KDCS call**

	1st parameter	2nd parameter
4.	KDCS parameter area	Message area

5. **C/C++ macro call**

Macro names	Parameters
KDCS_SRELGB / KDCS_SRELLB	(kcrn)

**openUTM return information**

	Field name in the KB return are	Contents
6.	KCRCCC	Return code
7.	KCRCDC	Internal return code

For the SREL call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, the SREL operation code.
2. In the **KCOM** field
  - LB to delete an LSSB, or
  - GB to delete a GSSB
3. In the **KCRN** field, the name of the LSSB/GSSB to be deleted

You specify the following for the KDCS call:

4. 1st parameter: the address of the KDCS parameter area.
5. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

6. in the **KCRCCC** field: the KDCS return code.
7. in the **KCRCDC** field: the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### **KDCS return codes in the KCRCCC field**

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 14Z No area exists with the name specified in KCRN.
- 40Z System cannot perform the operation (generation error or system error, deadlock, timeout), see KCRCDC.
- 42Z Entry in KCOM is invalid.
- 44Z Name in KCRN invalid (if it consists solely of blanks or binary zero).

An additional error code can be found in the dump:

- 71Z INIT missing in this program.



### Features of the SREL call

- openUTM does not execute SREL until the end of the current transaction.
- SREL is not executed
  - for service interrupt via PEND ER/FR, or
  - if there is a subsequent PEND RS or RSET call.
- An SREL call locks the area called until the end of the transaction or UNLK or until the next reset operation. In other words, openUTM rejects subsequent SGET calls to this area with the return code 14Z. However, if an SREL call is followed by an SPUT call with the same area name, this area (LSSB or GSSB) is set up anew.

The lock continues to apply after the end of a processing/dialog step when the program unit run is terminated or interrupted with:

- PEND KP and PGWT KP
- PEND PA/PR with a task change due to TAC class control
- PEND PA/PR or PGWT PR with waiting for a DGET message

If this area is a GSSB, then other services cannot access this GSSB either with SGET or with SPUT until the end of the transaction. The [section “Action with locked storage areas \(TLS, ULS and GSSB\)” on page 88](#) describes how openUTM reacts in this case

- At the end of a service (PEND FI/FC) or service interrupt (PEND ER/FR, possibly also PEND RS), openUTM automatically deletes all LSSBs. SREL calls to LSSBs are therefore superfluous in transactions which terminate services.
- In UTM cluster applications, an SREL call deletes a GSSB throughout the entire cluster, i.e. as soon as the SREL call takes effect, the GSSB can no longer be read in any of the node applications.

The following table describes the effect of the call sequences SREL ... RSET and SREL ... PEND on GSSBs and LSSBs.

Call	Effect on GSSB	Effect on LSSB
SREL	deletes and locks (operation remains resettable)	deletes (operation remains resettable)
... RSET/ PEND RS PGWT RB	resets and unlocks	resets
... PEND KP PGWT KP	leaves the operation resettable	leaves the operation resettable
... PEND RE PEND SP PGWT CM	deletes and unlocks (operation not resettable)	deletes (operation not resettable)
... PEND FI PEND FC		deletes all LSSBs
... PEND ER PEND FR	resets and unlocks	deletes all LSSBs
... PEND PA / PEND PR <sup>1</sup> / PGWT PR	lock remains, operation resettable	lock remains, operation resettable

<sup>1</sup> With a task change due to TAC class control or with waiting for a DGET message

## UNLK Unlock TLS, ULS or GSSB

You use the UNLK (unlock) call to unlock one of the following storage areas:

- the global secondary storage area (GSSB).
- a block of the terminal-specific long-term storage area (TLS).
- a block of the user-specific long-term storage area (ULS).

The area is not unlocked unless it was only read in the current transaction

In UTM cluster applications, GSSB and ULS areas are valid throughout the cluster. As a result, unlocking a GSSB or ULS with UNLK is effective throughout the cluster.

### Setting the 1st parameter (KDCS parameter area)

The table below shows the various options and the necessary entries in the KDCS parameter area.

Function of the call	Entries in the KDCS parameter area			
	KCOP	KCOM	KCRN	KCLT or KCUS
Unlock TLS (in dialog programs)	"UNLK"	"DA"	Block name	—
Unlock TLS (in asynchronous programs)	"UNLK"	"DA"	Block name	LTERM name/ LPAP name
Unlock GSSB	"UNLK"	"GB"	Name of GSSB	—
Unlock ULS	"UNLK"	"US"	Block name	User ID/ blanks/ LSES name/ association name

For KCOM = US, all the fields not used in the KDCS parameter area are to be set to binary zero.

**Setting the parameters**

	Field name in the KDCS parameter area	Contents
1.	KCOP	"UNLK"
2.	KCOM	"GB"/"DA"/"US"
3.	KCRN	Name of the area/block name
4.	KCLT or. KCUS	LTERM name/LPAP name or user ID/LSES name/association name/blanks

**KDCS call**

	1st parameter	2nd parameter
5.	KDCS parameter area	Message area

6. **C/C++ macro calls**

Macro name	Parameters
KDCS_UNLKGB	(kcrn)
KDCS_UNLKDA	(kcrn,kclt)
KDCS_UNLKUS	(kcrn,kcus)

**openUTM return information**

	Field name in the KB return area	Contents
7.	KCRCCC	Return code
8.	KCRCDC	Internal return code

For the UNLK call you make the following entries in the KDCS parameter area:

1. In the **KCOP** field, UNLK operation code.
2. In the **KCOM** field, the type of storage to be unlocked:
  - GB for a global secondary storage area (GSSB).
  - DA for terminal-specific long-term storage area (TLS).
  - US for a user-specific long-term storage area (ULS).
3. In the **KCRN** field, the name of the storage area to be unlocked.
4. Depending on the type of storage:
  - for unlocking a TLS in an asynchronous program:  
in the **KCLT** field, the name of the LTERM or (OSI) LPAP partner, whose TLS is to be unlocked.
  - for unlocking a TLS in a dialog program:  
irrelevant, the corresponding block of the associated TLS is always accessed.
  - for unlocking a ULS block:  
in the **KCUS** field, the user ID if a ULS block of a foreign user ID is to be unlocked or blanks for a ULS block of your own user ID. If you enter a foreign user ID in KCUS, your own user ID must have administration privileges.  
  
If you want to unlock a ULS block of a remote session/association, you have to specify its name.
  - for unlocking a GSSB:  
irrelevant.

You specify the following for the KDCS call:

5. 1st parameter: the address of the KDCS parameter area.
6. The use of C/C++ calls is described in detail in the [section “C/C++ macro interface” on page 497](#).

openUTM returns:

7. in the **KCRCCC** field, the KDCS return code.
8. in the **KCRCDC** field, the internal return code of openUTM (see the openUTM manual “Messages, Debugging and Diagnostics”).

### KDCS return codes in the KCRCCC field

The following codes can be analyzed in the program:

- 000 Operation carried out.
- 14Z No GSSB/TLS exists with the name specified in KCRN.
- 16Z GSSB/TLS/ULS not locked by your own transaction or GSSB created or changed in same transaction or TLS changed in same transaction.
- 40Z System cannot perform the operation (generation error or system error, deadlock, timeout).
- 42Z Entry in KCOM is invalid.
- 44Z For GSSB: entry in KCRN is invalid (blanks or binary zero).  
For TLS and ULS: name of the block in KCRN is unknown or invalid.
- 46Z LTERM or LPAP partner in KCLT is invalid (only for asynchronous programs and TLS) or the user ID is unknown (only for ULS).
- 49Z Contents of fields not used in the KDCS parameter area not equal to binary zero (only for KCOM = US)

An additional error code can be found in the dump:

- 71Z INIT missing in this program.

### Features of the UNLK call

- An UNLK call to a TLS/ULS/GSSB is only useful after a read call (GTDA or SGET). If the area is changed in the current transaction with PTDA, SPUT or SREL (with GSSBs), openUTM rejects the call with KCRCCC = 16Z; the area remains locked until the end of transaction.
- An UNLK call is useful, for example, before entering PEND KP or PGWT KP/PR. This causes TLS/ULS blocks and GSSBs to be released prior to the next synchronization point. This makes them available to the other transactions.
- An UNLK call for an area not locked by your own transaction is rejected (return code 16Z).
- In UTM cluster applications, an UNLK call unlocks a GSSB or ULS throughout the cluster, i.e. as soon as the UNLK call takes effect, transactions in other node applications are again able to access the unlocked area.

---

## 8 Event functions

In order to be able to react to certain events in a program, openUTM permits the use of what are known as event functions. Unlike “normal” program units which are called by specifying a transaction code, openUTM starts these program units when certain events occur.

There are two different types of event functions:

- event exits, which **must not contain any KDCS calls** and
- event services, which **must contain KDCS calls**.

B Since STXIT routines such as those used in BS2000/OSD are also event-driven, a  
B description of these routines has also been included in this manual in [section “STXIT  
B routines \(BS2000/OSD\)” on page 469](#). The [section “Event handling in ILCS programs  
B \(BS2000/OSD\)” on page 470](#) deals with event handling in ILCS programs.

### Event exits:

INPUT	This event exit is called with input from a data display terminal.
START	These event exits (up to 8) are called at each start of the application program.
SHUT	These event exits (up to 8) are called at each termination of an application work process, including at PEND ER.
VORGANG	This program unit is called at the start and the end of a service, and also in the case of incorrect termination, screen restart and - if RESTART=NO was set during generation - loss of connection.
B B	FORMAT This event exit is called if a message is entered or output which you then want to format using a formatting routine of your own (-format).

### Event services:

- BADTACS** This dialog service is called whenever an invalid transaction code is entered or data protection is violated.
- MSGTAC** This asynchronous service is called whenever openUTM outputs a message with MSGTAC defined as its destination (see the openUTM manual “Messages, Debugging and Diagnostics”, the section describing the modification of output messages).
- SIGNON** This dialog service is called whenever a terminal user, a TS application or a UPIC client signs on to the application. The prerequisite for this is that a sign-on service must be generated for the transport system access point via which the client signs on; in addition, UPIC= YES must be generated in the SIGNON statement for UPIC clients.  
For each transport system access point of a UTM application (generated with MAX APPLNAME or BCAMAPPL), a separate sign-on service can be generated (see the SIGNON-TAC parameter of the BCAMAPPL statement in the openUTM manual “Generating Applications”).

Database calls are only allowed in the program units for VORGANG, BADTACS, MSGTAC and SIGNON, and not in the case of program units for INPUT, START and SHUT.

- B** Database calls are not permitted in FORMAT program units.



## 8.1 Event exits

Event exits are created as subroutines without KDCS calls. This section describes the following event exits:

INPUT  
 START  
 SHUT  
 VORGANG  
 FORMAT

**B**

The event exits INPUT, START and SHUT are defined with the KDCDEF statement EXIT. You can define the event exit VORGANG with the TAC and PROGRAM statement.

**B**

The event exit FORMAT is also defined with the KDCDEF statement EXIT.

Of these five event exits, database calls are permitted only in the VORGANG exit.

### Reading the application name

An event exit is able to read the name of the own application by calling the KDCAPLI entry, see the following example:

Assembler program:

```

                EXTRN    KDCAPLI
AKDCAPLI DC      A(KDCAPLI)

```

C program:

```
extern char KDCAPLI[8];
```

### 8.1.1 Event exit INPUT

You can use this event exit to determine the effect of terminal input.

openUTM calls the event exit INPUT - with a few exceptions - every time input is made at a terminal. The exceptions are as follows:

- input in the event service SIGNON
- input using a function key generated with
- input after a program has issued the call SIGN OB/OF.

openUTM passes the address of the parameter area, comprising an input area and an output area, to the program unit for an INPUT exit.

Programming language-specific data structures are available for declarations in the parameter area: COBOL uses the COPY member KCINPC, whereas C/C++ use the header file *kcinp.h*. For an explanation of the individual fields in the parameter area and the meanings of these fields, see pages [451 - 453](#).

**B** In BS2000/OSD, openUTM also passes a second parameter to the input exit containing additional input fields. Once again, a programming language-specific data structure is available for the structure of the second parameter as well: the COBOL structure is the COPY member KCCFC; the C/C++ structure is the header file *kccf.h*. The second parameter area is described on pages [454 - 456](#).

The program unit analyzes the values entered in the input area and sets the output area accordingly. Depending on the values entered in the output area, openUTM decides which of the following actions is to be executed:

- continue the service or
- start a new service or
- stack a service and start a new service or
- execute a user command, e.g. KDCOFF or
- send a message with error information to the terminal.

### Possible applications

The input exit provides the terminal interface with greater freedom regarding the layout of the user interface.

- Position of transaction codes or KDC commands in the message:  
transaction codes or KDC commands do not have to appear at the start of a message.
- Visibility of transaction codes on screen:  
The Transaction code has not to be visible on screen if you want to start a service. For example, a menu might offer a number of activities. The terminal user enters a text or a number to choose one of them. This input itself does not have to be a transaction code: it is only the INPUT exit which converts it to a transaction code (a service TAC). In this way transaction codes, which are an element of generation information, become independent of the dialog interface.
- Representation of command names:  
There is no need to specify “KDC..” in order to issue a KDC command. If it is felt to be useful, commands can be represented in some other form, e.g. “/” instead of “KDC”.

Parameter area KDCINPC / kdcinp.h

Input area (supplied by openUTM)	
Field name	Contents
1. KCIFCH	First 8 characters of the input
2. KCIMF/kcifn	Format identifier
3. KCICVTAC	Service transaction code
4. KCICVST	Service status
5. KCIFKEY	Value of F key: 1,...,20 / binary zero
6. KCIKKEY	Binary zero/ value of K key: 1,...,20
7. KCICFINF: "UN"/"NO" Also possible in BS2000/OSD: "ON"/"MO"	Formatting information
8. KCILTERM	Current LTERM partner
9. KCIUSER	Current user ID
Output area (supplied by the INPUT exit)	
Field name	Contents
10. KCINTAC or KCINCMD	Next service TAC or next user command
11. KCICCD	Code for the effect of the input: "ER"/"CC"/"SC"/"ST"/"CD"
12. KCICUT	Truncate TAC: "Y"/"N"
13. KCIERRCD	Error info for terminal (4 bytes)

B  
B  
B

openUTM enters the following values in the appropriate fields of the input area:

1. **KCIFCH**: the first 8 characters of the input, at most, however, up to the first blank.
2. **KCIMF/kcifn**: the format identifier specified with MPUT.

B  
B  
B  
B

As of FHS V8.0, the field **KCIMF/kcifn** can also contain the format identifier #!POPUP. #!POPUP indicates that a box “pops up” on screen. You can use the FHS service function KDCFHS with the user code INFD to display the names of all formats on the screen.

3. **KCICVTAC**: the transaction code used to start the current service (if any).
4. **KCICVST**: the service status:
 

ES	(End of dialog <b>S</b> tep)	End of the dialog step with PEND KP or PGWT KP
ET	(End of <b>T</b> ransaction)	End of the transaction with PEND RE
RS	( <b>R</b> eturn from <b>S</b> tack)	End of the inserted service; the input is intended for the stacked service.
EC	(End of <b>C</b> onversation)	The last dialog step was terminated with PEND FI; the input is intended for a new service.
5. **KCIFKEY**: the value of the F key (1 - 20, in BS2000/OSD: 1 - 24) if pressed, otherwise binary zero.
6. **KCIKKEY**: binary zero  
 or (in BS2000/OSD) the value of the K key (1 - 14), if pressed
7. **KCICFINF** Information of the formatting system:
 

UN	( <b>U</b> Nsuccessful)	Input could not be formatted. Therefore no control field specifications were possible; this entry is made, for instance, after input in a format fetched with KDCOUT
NO	( <b>N</b> O control field)	The input does not contain any control fields; either the input was in line mode or the format does not contain any control fields or one of the control fields was empty.
ON	( <b>O</b> Ne control field)	The input contains exactly one control field.
MO	( <b>M</b> Ore control fields)	The input contains several control fields.
8. **KCILTERM**: name of the LTERM partner via which the terminal is connected.
9. **KCIUSER**: the current user ID.

You input the following in the fields of the output area:

10. **KCINTAC**, if a new service is to be started:  
the TAC of the program unit which starts the next service.  
**KCINCMD**, if a user command is to be executed:  
the user command (KDC command).  
  
Blanks have to be entered in the case of an error message (KCICCD = ER) or continuation of the service (KCICCD = CC, follow-up TAC is entered with PEND!).
11. **KCICCD**, depending on the required effect of the input:  

ER	( <b>ER</b> roration indication) for an error message to the data display terminal. You can then enter an insert in the KCIERRCD field for this message.
CC	( <b>C</b> ontinue <b>C</b> onversation) for continuing the service. This must not be specified after end of service (KCICVST = EC).
SC	( <b>S</b> tart new <b>C</b> onversation) if a new service is to be started; only allowed after end of service (KCICVST = EC).
ST	( <b>S</b> Tack Conversation) if the current service is to be stacked and a new service is to be started; allowed only at end of transaction (KCICVST = ET/RS).
CD	(process <b>C</b> ommand) if openUTM is to execute a user command.
12. **KCICUT**: the value Y if the TAC is to be truncated at the start of a service (permissible only where KCICCD = SC/ST), otherwise N.
13. **KCIERRCD**: a character string of up to 4 bytes to be sent to the terminal with the UTM message K098 if KCICCD=ER. Otherwise (KCICCD ≠ ER) blank.



B In the second parameter area, openUTM supplies the following values:

B 1. **KCCFCREM** contains the remark defined for the control field when the format was  
 B generated with IFG and whose contents are included in KCCFCFLC. If no remark was  
 B created when the format was generated, or if no entry was made in the control field, the  
 B 8 bytes in this field are filled with blanks.

B 2. **KCCFCFLD** contains the entry for a control field in the format, provided that at least one  
 B control field is defined in the format and that an entry has been made in it. KCCFCFLD  
 B contains this entry with the length permitted by the control field. The remainder of the  
 B KCCFCFLD field is padded with blanks.

B If no entry has been made in a control field, or if the format does not have any control  
 B fields, then both KCCFCREM and KCCFCFLD will contain blanks and KCCFNOCF will  
 B have the value null.

B If the format has several control fields which, in some cases, will be in different  
 B subformats, then both the remark and the contents of the control field will refer to a  
 B control field in which an entry has been made. If entries were made in more than one  
 B control field, KCCDCFLD will assume the entry made in the first control field in the  
 B uppermost subformat on the screen. KDDFCRREM will then contain the corresponding  
 B remark.

B 3. **KCCFNOCF** contains the number of control fields in the format in which entries have  
 B been made.

B 4. **KCCFS** contains an array which covers all the control fields passed by FHS, including  
 B the control field whose data is already entered in the fields KCCFCFLD and  
 B KCCFCREM. The number of valid array elements is stored in the field KCCFNOCF.  
 B Each array element consists of a structure containing the following fields:

B **KCCFFNAM**

B contains (in 8 bytes) the (sub-) format name of the format to which the control field  
 B belongs. If the name is less than 8 characters long, the rest of the field is padded  
 B with blanks.

B **KCCFREM**

B contains the remark defined for this control field when the format was generated  
 B with IFG. If no remark was created when the format was generated, or if no entry  
 B was made in the control field, the 8 bytes in this field are filled with blanks.

B **KCCFLOFL**

B length of the control field.

B **KCCFFLD**

B contains the entry for a control field in the format. KCCFFLD contains this entry with  
 B the length permitted by the control field. The remainder of the KCCFCFLD field is  
 B padded with blanks.

**B** **Input in control fields**

- B** Input can be made in a control field from a variety of sources:
- B** – Terminal users can make entries in the control field.
  - B** – The field may have been defined with the property “automatic input” during format generation with IFG.
  - B** – In the case of \*formats or +formats: The field has the property “unprotected” and the FHS start parameter ISTD=RUNP (read unprotected) is set.

**Errors with INPUT exit**

If there are errors in the INPUT exit, an open service is not terminated; however, the terminal user is informed of the error with message K098 if

- the entry in KCICCD (effect of the input) is invalid or
- the entry in KCICCD does not match the values of the other output fields.

In both cases, a UTM dump with REASON=INPERR is supplied for error diagnosis.

- B** In BS2000/OSD, a USERDUMP is also generated in such cases.

Database calls are not permitted in the INPUT exit.

- B** If database calls are nevertheless included, a USERDUMP is generated in BS2000/OSD with the error code KDCDB10.

**Generation notes**

The event exit INPUT must be defined during generation with the EXIT statement and the operand USAGE=(INPUT,...). You can also defined several INPUT exits for a variety of purposes. The following options are available:

- An application contains just one universal INPUT exit which is called with inputs in formatted mode and with inputs in line mode. This universal exit is generated with USAGE=(INPUT,ALL). Other INPUT exits are not allowed in this case.
- A special INPUT can be generated for each type of format identifier. There are two possible types; no more than one of each type may be defined:

- B** – With USAGE=(INPUT,FORMMODE) an INPUT exit is defined for "+" and "\*" formats.

**B** An INPUT exit generated in this way will also be called for the #formats (BS2000/OSD).

- B** – With USAGE=(INPUT,USERFORM), an INPUT exit is defined for the user’s own formatting routines (-formats).

- With USAGE=(INPUT,LINEMODE), an INPUT exit is defined for input in line mode.

If special INPUT exits are used in an application, then it is not allowed to define a universal type with USAGE=(INPUT,ALL).



## 8.1.2 Event exit START

START exits are called when the application program is started, when the program is reloaded after a PEND ER or when the entire application program is replaced.

The START event exits are called in the order of the corresponding EXIT statements in the KDCDEF file.

You can use START exits, for example, in order to open files in order to work with them.

### Programming notes

- You can access the KB header and SPAB in this event exit, but the KB program area and the SPAB do not contain any relevant data. openUTM enters the transaction code "STARTUP" for this program unit in the KCTACVG/kccv\_tac and KCTACAL/kcpr\_tac fields of the KB header.
- When the first process of the application is started, openUTM enters the service ID "F" in the KCKNZVG/kccv\_status field, otherwise a blank.
- If you use other shared storage areas (AREAs), you can access them in the START exit. For more information refer to the following sections: ["Other data areas \(AREAs\)" on page 489](#) and ["Extending the LINKAGE SECTION" on page 546](#).
- You are not allowed to use KDCS calls in this program unit.
- You can exit the event exit START with a return statement.
- The additional processes (if there are any) are only started after the first process of an application is started.
- If an error occurs in the START program unit (e.g. because of an attempt to open a non-existent file), the event exit START CAN ensure that the current process is terminated. An error message must, however, always be written beforehand.
- Should irrecoverable errors occur during the start exit for the first process in the UTM application and thus prevent the UTM application from running, the start exit routine can only be terminated with *exit(-1)* (with COBOL85: set RETURN-CODE to -1 and afterwards STOP RUN). The START program unit which starts the UTM application is then aborted.
- X/W – If the start exit for a follow-up process in a UTM application is terminated with *exit(-1)* (COBOL85: RETURN-CODE to -1 and STOP RUN), the UTM application loses a process. For this reason, the start exit for a follow-up process must never be terminated with *exit(-1)*. Instead, this status must be handled by other program units - if necessary by shutting the process down from the administration interface.

- B – If the application program is to be terminated and a USERDUMP is to be written you can, for example, call a short Assembler program in which you first call the CDUMP macro and then call TERM with the operand UNIT=STEP.

### Generation notes

- The program unit for the event exit START has to be defined at generation with the EXIT statement and the operand USAGE=START.
- A maximum of 8 event-driven START program units are allowed per application.

This version enables you to use several START exits and thus allows you to work better with preconfigured or purchased application components, which often have their own START and SHUT exits. These can now be processed one after the other. In addition, as an application operator, you can also add your own START exits.

## 8.1.3 Event exit SHUT

SHUT exits are called when the application program is terminated (e.g. as a result of PENDING). You can use SHUT exits, for example, to close your own files.

The SHUT event exits are called in the order of the corresponding EXIT statements in the KDCDEF file.

### Programming notes

- You can access the KB header and SPAB in this event exit, but the KB program area and the SPAB do not contain any relevant data. openUTM enters the transaction code "SHUTDOWN" for this program unit in the KCTACVG/kccv\_tac and KCTACAL/kcpr\_tac fields of the KB header.
- If you use other shared storage areas (AREAs), you can access them in the SHUT exit. For more information refer to the following sections: [“Other data areas \(AREAs\)” on page 489](#) and [“Extending the LINKAGE SECTION” on page 546](#).
- If the application terminates normally, openUTM enters the service indicator "L" in the KCKNZVG/kccv\_status field for the final process of the application, otherwise a blank.
- You are not allowed to use KDCS calls in this program unit.
- You can exit the event exit SHUT with a return statement.

- B – If an error occurs in the SHUT program unit (e.g. because of an attempt to open a non-existent file), the event exit SHUT can be terminated. Before doing so, however, you should first issue a message that you are going to terminate the program. If the application program is to be terminated and a USERDUMP is to be written you can, for example, call a short Assembler program in which you first call the CDUMP macro and then call TERM with the operand UNIT=STEP.

### Generation notes

- The program units for the SHUT event exits must be defined at generation with the EXIT statement using the operand USAGE=SHUT.
- A maximum of 8 event-driven SHUT program units are allowed per application.

This version enables you to use several SHUT exits and thus allows you to work better with preconfigured or purchased application components, which often have their own START and SHUT exits. These can now be processed one after the other. In addition, as an application operator, you can also add your own SHUT exits.

For an example of how an event exit SHUT can be used in C or COBOL, see [page 533](#) and [page 584](#).

## 8.1.4 Event exit VORGANG

openUTM calls the VORGANG event exit when a service is started or terminated, including incorrect termination and restart.

Even if a service consists of two or more transactions processed in different program units, the same VORGANG event exit which was called at the start of the service is called again at the end of the service.

If VORGANG is called at the end of a service, but an error occurs at this time, the last transaction of the service is not reset.

### Programming notes

- You are not allowed to use KDCS calls in this program unit.
- You can exit the event exit VORGANG with a return statement.
- You can access the KB header and SPAB but the KB program area and the SPAB do not contain any relevant data.

openUTM enters the service indicator for this program unit in the KCKNZVG/kccv\_status field of the KB header. The service indicator can assume one of the following values:

- F First program unit run of a dialog service
- C First program unit run of a concatenated service
- A Restart of a service
- R Errored end of service
- Z End of service
- E End of service
- D Termination of the service due to connection clear-down or loss of connection if RESTART=NO is generated for the UTM user id.

The program indicator (field KCPRIND) shows whether a program is running in a dialog service or in an asynchronous service:

- A The program unit is running in an asynchronous service.
- D The program unit is running in a dialog service.

Note that in distributed transaction processing, after the end of the service, the VORGANG event exit may be processed by a process other than the one which processed the last program unit in the service.


### Generation notes

- There can be more than one VORGANG program unit for a single application.
- Which event exit VORGANG is called for which service is defined with the TAC statement, operand EXIT=. In particular, the event-driven services BADTACS, MSGTAC and SIGNON can have an event exit VORGANG.
- A PROGRAM statement is also required for an event exit VORGANG.

## 8.1.5 Event exit FORMAT (BS2000/OSD)

B If you use the event exit FORMAT, you are implicitly doing without the output support  
 B provided by the system components FHS and VTSU. You will therefore need to write your  
 B own formatting and may need to handle screen restarts manually. openUTM recognizes the  
 B event exit FORMAT by means of the “-” prefix to the format identifier in KCMF/kcfn.

B The event exit FORMAT must be generated in the EXIT statement with the operand  
 B USAGE=FORMAT. Only one event exit FORMAT is permitted per application. Subformats  
 B must not be used.

B  The event exit FORMAT is a program which implements system software functions.  
 B The code it contains must therefore be written very carefully: no exits should be left  
 B open or paths uncoded etc., as errors can cause the service, and possibly even the  
 B application, to crash. If in doubt, terminate the event exit FORMAT with formatting  
 B errors.

B The event exit FORMAT can be written in any of the programming languages supported by  
 B openUTM. The programming languages Assembler and C/C++ are, however, particularly  
 B well suited to this purpose. The description of the FORMAT exit provided in the remainder  
 B of this section is based on the Assembler language. Nevertheless, here is a brief example  
 B of the prototype of a FORMAT exit in C:

B *Prototype of a FORMAT exit in C (ANSI)*

```
B      void FORMATEX( struct kc_ca * const pKB
B                      ,char      * const pSPAB
B                      ,char      * const pFormatName
B                      ,char      * const pDevice
B                      ,char      * const pFormatArea
B                      ,char      * const pPhysicallyInOutArea
B                      ,char      * const pRestartArea
B                      ,char      * const pFormatControlArea
B                      ,char      * const pInOutIndicator
B                      ,char      * const pSecondaryReturnCode
B                      );
```

**B Address list**

**B** When the event exit is called, openUTM provides the following address list:

<b>B</b> <b>B</b>	<b>Sequence of word addresses</b>	<b>Contents</b>
<b>B</b>	1.	Address of the KB
<b>B</b>	2.	Address of the SPAB
<b>B</b>	3.	Address of the format name
<b>B</b>	4.	Address of the terminal type
<b>B</b>	5.	Address of the formatting user area
<b>B</b>	6.	Address of the physical input/output area
<b>B</b>	7.	Address of a restart area
<b>B</b>	8.	Address of the formatting control area
<b>B</b>	9.	Address of the input/output indicator

**B** openUTM calls the formatting routine whenever one of the KDCS calls MPUT, FPUT, DPUT, MGET or FGET is used in a -format.

**B** A dialog output message (MPUT) is not formatted until after the PEND call. The contents of KB and SPAB are identical to those of the PEND call. An asynchronous output message is not formatted until it is sent. The contents of KB and SPAB are undefined in this case, i.e. they no longer relate to the program unit which called FPUT.

**B** A dialog input message is formatted during MGET handling. KB and SPAB have the contents of the MGET call. This applies only for the event exit FORMAT: normally, openUTM formats messages before the INIT.

**B** An asynchronous input message is formatted when the message is received, not when the FGET occurs. The contents of KB and SPAB are therefore undefined.

**B** Please ensure that there are no fields that precede the transaction code, since openUTM identifies the TAC without calling the format exit.

**B Format name (address in word 3)**

**B** – In the case of input formatting: The format name used for this terminal during the previous output formatting operation (calls: MPUT, FPUT, DPUT or the command KDCFOR).

**B** – In the case of output formatting: The specification made in KCMF/kcfn in the parameter area for the MPUT or FPUT call.

**B Terminal type and additional information (address in word 4)**

t	z	ba	qa	q1	q2	dc
0	1	2	4	5	6	7

**B** t is the physical device type which can be queried by means of a TSTAT macro with  
**B** TCHAR. The codes can be looked up in the DCSTA macro (see the manual  
**B** “BS2000/OSD-BC Executive Macros”).

**B** If the specified terminal type is not supported by the event exit FORMAT, a  
**B** formatting error **must** be forced.

**B** z is an item of additional information:

**B** X'00' = delete screen.

**B** X'01' = do not delete screen.

**B** ba is the screen output function:

**B** X'0001' KCRESTRT

**B** X'0001' KCREPL

**B** X'0002' KCERAS

**B** X'0004' KCALARM

**B** X'0008' KCREPR

**B** X'2000' KCEXTEND

**B** X'4000' KCCARD

**B** qa is an identifier which indicates whether confirmation must be requested - applies to  
**B** output to printers only.

**B** X'00' = request confirmation

**B** X'01' = do not request confirmation

**B** q1,q2 If qa = X'01', then these two bytes will contain the confirmation numbers in EBCDIC  
**B** code. If you are using a message header, both of these bytes must be converted to  
**B** ASCII and entered in the return bytes RB1 and RB2 (see the manuals for your  
**B** terminal).

**B** dc device characteristics; contains 8 bytes of information on the device type and  
**B** configuration, in the format that would be generated in PDN with XSTAT and  
**B** XOPCH. Refer to the DCSTA macro for information on the significance of the first  
**B** 8 bytes (see the manual “BS2000/OSD-BC Executive Macros”).

**B Formatting user area (address in word 5)**

**B** Input formatting:

**B** Once the format exit has been called, openUTM expects the logical message here in the input format (see [“Message formats” on page 466](#)). The maximum length is determined by the generated value (see MAX statement, operand NB, in the openUTM manual “Generating Applications”). openUTM enters this maximum length in the length field of the formatting user area before the call is made.

**B** If conversion of the logical message to a physical message causes it to exceed the length of this area, a formatting error **must** be forced.

**B** Output formatting:

**B** The area contains the message to be formatted in the form in which it was made available in the message area with MPUT or FPUT. The message is structured in input format. Subformats are not permitted.

**B Physical input/output area (address in word 6)**

**B** Input formatting:

**B** openUTM passes the message as it is received from the terminal in the input format.

**B** Output formatting:

**B** Once the format exit has been called, openUTM expects to find the formatted message here in output format.

**B** The maximum length is determined by the maximum message length allowed on the connection to the client. openUTM enters this maximum length in the length field of the physical input/output area before the call is made.

**B** If conversion of the logical message to a physical message causes it to exceed the length of this area, a formatting error **must** be forced or an alternative message must be generated.

**B Restart area (address in word 7)**

**B** The restart area must be used to reconstruct screen formats where necessary, which can arise in three different situations, for example:

**B** – with the KDCDISP command (display last screen)

**B** – if the terminal user wants to continue an interrupted service with the most recently entered format (e.g. after a connection has been lost or after the command KDCOFF has been issued while a service was still running).

**B** – if the screen is destroyed by an asynchronous output and is to be restored again.



B You must ensure that the restart area always contains the current logical message so that  
B this can be output if the need arises.

B If you first want to delete the old screen when formatting output, you must refresh the restart  
B area.

B If you do not want the old screen to be deleted in this case, you need only modify (overwrite)  
B those fields in the restart area that are also modified at the terminal.

B After input has been made at the terminal, the restart area must be updated accordingly.

B In the case of a PEND, the restart area is backed up to the length specified for the user area  
B for this format when the output was last formatted (KCLM for MPUT).

B In the event of a restart, output formatting takes place. The address of the formatting user  
B area is the same as that of the restart area (i.e. there is only a restart area).

### B **Formatting control area (address in word 8)**

B As of address + 1 in this area, you must store in the event exit FORMAT a (hexadecimal)  
B return code which indicates whether formatting was successful.

B The return code must be one of the following:

B X'00' Output formatting was successful.

B X'xy' User error messages were passed in the form of UTM return code "FRxy" in the field  
B KCR CDC (formatting errors). The entries X'01', X'02', X'03', X'04', X'08', X'10' and  
B X'99' are not permitted: these return codes are reserved for cooperation between  
B openUTM and FHS.

### B **Input/output indicator (address in word 9)**

B The first byte of the indicator contains the value:

B X'00' for input formatting

B X'01' for output formatting

B X'02' for a restart

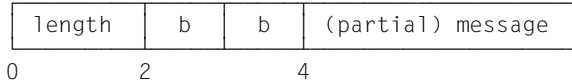
B X'03' for KDCFOR

B In the case of a restart, the addresses of the formatting and restart areas are identical.

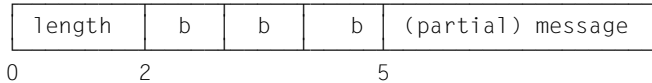
**B Message formats**

**B** Two different message formats are used in the event exit FORMAT (see previous section).  
**B** These formats do not necessarily adhere to the half-word boundary.

**B** input format



**B** output format



**B** length is the length of the entire message (binary), including the prefix, which accounts for  
**B** 4 bytes in input formatting and 5 bytes in output formatting.

**B** b is a blank (X'40')

**B** For more information on the structure of physical messages, refer to the manuals for your  
**B** terminal.

**B Example**

```

B FEXIT      CSECT
B           STM  14,12,12(13)
B           BALR 12,0
B           USING *,12
B           USING KB,2
B           USING SPAB,3
B           USING DFORMNAM,4
B           USING DFORMSDE,5
B           USING DADUSERA,6
B           USING DAREAFMI,7
B           USING DADRSRTA,8
B           USING DMDCBUSE,9
B           USING DFORMMOD,10
B           LM   2,10,0(1)
B
B *
B * INPUT OR OUTPUT FORMATTING, RESTART OR KDCFOR ?
B *
B           CLI  FORM#IND,X'00'           00=INPUT,   01=OUTPUT
B           BE  EINFORM                   02=RESTART, 03=KDCFOR
B           CLI  FORM#IND,X'01'
B           BE  AUSFORM
B           CLI  FORM#IND,X'02'
B           BE  RESTFORM
B           CLI  FORM#IND,X'03'
B           BE  FORFORM

```


```

R      *
B      * SET FORMATTING ERROR: INVALID OPCODE !
B      *
B      ***** K D C F O R
B      *
B      FORMFORM DS      OH
B      *
B      *      A) SET UP PHYSICAL MESSAGE AS PER FORMAT NAME
B      *      AND PREASSIGN WITH "STD." VALUES
B      *      B) ANALYZE CONFIRMATION REQUEST ('FORMQA')
B      *      AND SUPPLY MESSAGE HEADER
B      *      C) SET UP RESTART AREA
B      *      D) SET RETURN CODE IN FORMATTING CONTROL AREA
B      *
B      END#EXIT LM      14,12,12(13)
B      BR      14
B      *
B      ***** INPUT FORMATTING
B      *
B      EINFORM CNOP      0,4
B      *
B      *      A) ANALYZE PHYSICAL NAME AS PER FORMAT NAME AND
B      *      SET UP FORMATTING USER AREA ('DADUSERA') AS PER
B      *      MESSAGE FORMAT FOR INPUT
B      *      B) SUPPLY RESTART AREA
B      *      C) SET RETURN CODE IN FORMATTING CONTROL AREA
B      *
B      B      END#EXIT
B      EJECT
B      ***** OUTPUT FORMATTING
B      *
B      AUSFORM CNOP      0,4
B      *
B      *      A) SET UP PHYSICAL MESSAGE
B      *      THE FOLLOWING ELEMENTS ARE ANALYZED:
B      *      - FORMATTING USER AREA
B      *      - FORMAT NAME
B      *      - STATION TYPE AND ADDITIONAL INFORMATION
B      *      WITH
B      *      1) 'FORMCLMO' (DELETE SCREEN?
B      *      I.E. "FORMAT OLD" = "FORMAT NEW" ?)
B      *      2) 'FORMBA' (SCREEN OUTPUT FUNCTION)
B      *      3) 'FORMQA' AND GGF. 'FORMRB'
B      *      B) SUPPLY RESTART AREA
B      *      C) SET RETURN CODE IN FORMATTING CONTROL AREA
B      *
B      B      END#EXIT

```



## 8.2 STXIT routines (BS2000/OSD)

- B In BS2000/OSD it is possible to define STXIT routines for certain events (e.g. address errors, program end). These routines are activated by the operating system (not by openUTM) if one of the specified events occurs (see the manual "BS200/OSD-BC Executive Macros").
- B openUTM provides STXIT routines of its own which are opened before the START exit is called. Exception: TIMER/RTIMER, which are opened immediately after the START exit and are used by openUTM to check the program unit runtime (see the TIME parameter for the TAC statement).
- B You can define your own parallel STXIT routines which are then activated in addition to the ones defined by openUTM. If you specify the start parameter STXIT=OFF, only those STXIT routines that you have defined are activated (UTM STXIT routines are not activated). The latter is only possible if the application was started in the dialog.
- B The STXIT routines that you have created are always activated before the UTM STXIT routines (with the exception of RUNOUT).
- B  If programs are loaded dynamically with associated runtime systems after the start phase, and if these programs initiate their own STXIT routines, the sequence in which the STXIT routines are activated may differ from the one described here.
- B The STXIT routines that you have created must be terminated with EXIT CONTINU=YES. Otherwise, openUTM cannot guarantee that errors will be handled correctly (e.g. PEND ER in certain situations). The openUTM STXIT routines are terminated with EXIT CONTINU=NO.

### 8.3 Event handling in ILCS programs (BS2000/OSD)

B In an ILCS application, all interruptions are first rerouted to the ILCS event routines.  
 B Whether and how an event is handled depends on:

- B – the event itself
- B – the programming language in which the most recent active program is written
- B – the event handling routines in the programs involved
- B – the call sequence, if this contains different ILCS programming languages

**B Event handling variants:**

- B 1. The program unit (i.e. a program within the call sequence) or a runtime system incorporated in its call sequence handles the event and continues processing.  
 B
- B 2. Within the framework of the program unit as described above, the event is not handled  
 B but is passed on to openUTM together with its context and event code.

B Two distinctions must be drawn in this case:

- B a) It is an event of the class PROCHK or ERROR:

B openUTM prepares the message K102 together with the context and IW passed to  
 B it. In this case, the IW is additional control element for the following PEND handling.

- B b) It is any OTHER EVENT:

B If the event 'Other Event' occurs, openUTM can only offer a stereotypical response  
 B and has to prevent the task from being terminated abnormally. The event code that  
 B is passed is not an IW in the sense used in STXIT - its value is of no significance to  
 B openUTM. In particular, it is important to prevent this event code, in its role as a  
 B control element in UTM PEND handling, from leading to incorrect interpretations.  
 B To enable UTM error handling to run correctly, the event code is therefore set to the  
 B value X'FF'. During continued processing, openUTM interprets this value as a  
 B (simulated) IW. In other words, if an OTHER EVENT is passed to openUTM, then  
 B openUTM behaves as though an STXIT event had occurred with IW=X'FF'.

B In such cases, diagnosis has to be based on the message that was output when the  
 B OTHER EVENT occurred by the corresponding language, the program unit or ILCS.  
 B If no such message is output, the STXIT event X'FF' is displayed together with the  
 B UTM message K102...KDCIWFF.

B The resulting user-dump contains the code KDCIWFF.

B 3. Within the framework of the program unit as described above, the event is interpreted,  
B a message is output (if necessary) and the ILCS entry IT0TERM is called. This entry  
B activates openUTM and simulates the event handling of a TERM UNIT=PROG. In this  
B case, the event data for the primary interruption could only be derived from the UTM  
B message issued by the program unit or from the corresponding language. openUTM  
B issues the message K102... KDCIW90 (TERM), together with all register states up to  
B the time of the IT0TERM call.  
B The resulting user-dump contains the code KDCIW90.

B 4. An ILCS-internal error occurs:

B ILCS branches to a defined UTM entry; openUTM sets an internal event code,  
B IW=X'00', which can have one of two meanings:  
B – an ILCS-internal error occurred  
B – the ILCS stacking chain has been interrupted.

B openUTM outputs this IW=X'00' in the message K102. In its function as a control  
B element, this IW=X'00' is interpreted by openUTM during further execution of the  
B program as an IW=X'88', i.e. the task is terminated with TERM UNIT=STEP.

#### B **STXIT events - how ILCS behaves if an error occurs**

B After STXIT events which ILCS or the EHL routines that were called are unable to process,  
B ILCS closes its own STXIT and terminates the routine with EXIT CONTINUE=NO. In most  
B cases, the same error will occur again immediately, whereupon other STXIT routines (e.g.  
B those controlled by UTM) can handle the error. In isolated cases, such as an exponential  
B overflow, the program environment (register states etc.) at the start of the ILCS STXIT  
B routine will already have been modified in such a way that it is now no longer possible to  
B reconstruct the error environment. Before the STXIT is terminated, ILCS uses the IW to  
B check whether it is possible for the error to recur. If not, ILCS calls its program termination  
B routine IT0TERM.

## 8.4 Event services

Event services are dialog or asynchronous services which are started as a result of a certain event, i.e. they are event-driven services. The program units in these services must contain KDCS calls and adhere to the rules described in [chapter “Structure and use of UTM programs” on page 29](#).

Event services are generated with a privileged transaction code which openUTM uses internally. The following event-driven services are possible:

BADTACS with the TAC **KDCBADTC**

MSGTAC with the TAC **KDCMSGTC**

SIGNON with the TAC **KDCSGNTC** or specified as **SIGNON-TAC** in the BCAMAPPL statement

The transaction codes for these event-driven services are defined with the KDCDEF statement TAC.

Database calls are permitted, but (in the case of SIGNON) only if this was permitted explicitly during generation (SIGNON ...,RESTRICTED=NO).

### 8.4.1 Dialog service BADTACS

If it has been generated, the dialog service BADTACS is started whenever a client has entered an invalid transaction code. This can have any of the following reasons:

- The transaction code has not been generated.
- The transaction code is not assigned to any program unit.
- The transaction code is an administrator TAC but the user did not sign on with administrator privileges.
- A lock code is assigned to the transaction code but the user or LTERM partner does not have the corresponding keycode.

#### Programming notes

- The dialog service BADTACS must issue an MPUT call in accordance with the rules described in the chapter “Structure and use of UTM programs”. This MPUT call then replaces error message K009, which openUTM would generate in the absence of a BADTACS service.
- At the start of a service, after the INIT call, the invalid TAC is located in the KB header, in both the KCTACAL/kcpr\_tac and KCTACVKG/kccv\_tac fields.



- If the BADTACS service is called in line mode, it will receive the full message including the first 8 bytes. If it is called in format mode, the first 8 bytes (for \*formats) or 10 bytes (for +formats) are removed from the message.
- Outside of a service, BADTACS can also be started with a function key if one has been generated with a return code (20Z-39Z) but no TAC (SFUNC statement, operand RET=), or if the function key has not been generated (return code 19Z). The return code is then passed with the first MGET; a message (including the first 8 bytes) can be read with the second MGET. No entry is made in the KCTACAL/kcpr\_tac and KCTACVg/kccv\_tac fields.

### Generation notes

- There must be no more than one BADTACS service per application.
- The first program unit in the dialog service BADTACS must be defined during generation with the TAC statement `TAC KDCBADTC, PROGRAM=...`

Examples of BADTACS event services in C and COBOL are given on [page 531](#) and [page 588](#) respectively.

## 8.4.2 Asynchronous service MSGTAC

The asynchronous program unit MSGTAC is called:

- if openUTM outputs a *Knnn* or *Pnnn* message
- if MSGTAC is entered as the destination for this message.

How you enter MSGTAC as the message destination and how the messages that are passed are structured is described in the openUTM manual “Messages, Debugging and Diagnostics”.

The MSGTAC service is given administration authorization and all keys of the application by openUTM, i.e. with the maximum authorization allowed.

If the MSGTAC program unit aborts with `KCRCCC ≥ 70Z`, openUTM locks it for the remainder of the application run (`STATUS=OFF`). It then needs to be released explicitly by the administrator (`STATUS=ON`). This rule does not apply if a program is aborted by a programmed `PEND ER/FR`.

openUTM writes messages for the message destination MSGTAC to the page pool. As long as warning level 2 for this page pool is not exceeded, you can be sure that none of these UTM messages will be lost.

Exception: The messages indicating that the warning level has been exceeded or not reached cannot always be sent to the MSGTAC program unit.

### Programming notes

- The asynchronous program MSGTAC reads the message to the message area for the program unit with an FGET call. In this context, FGET calls should be repeated as often as necessary until the return code 10Z is set, to ensure that all pending UTM messages are read in a single program unit run.
- For each UTM message, there is a data structure available which can be used in the program unit to interpret the message contents. For C/C++, these structures are contained in the header file *kcmsg.h*; for COBOL they are contained in the COPY member KCMMSGC.
- The MSGTAC program unit runs as an asynchronous service under the internal UTM user ID KDCMSGUS with KSET=MASTER and PERMIT=ADMIN.
- The MSGTAC service can only consist of one program unit run.
- In a program unit run, at least one message must be read with FGET as otherwise the program unit will be terminated abnormally.
- The service is started internally with the administrator TAC KDCMSGTC.

openUTM supplies the KB header as follows:

Fields in the KB header		Entries for MSGTAC
COBOL	C/C++	
KCBENID	kcuserid	KDCMSGUS
KCTACVG	kccv_tac	KDCMSGTC
KCTACAL	kcpr_tac	KDCMSGTC
KCLOGTER	kclogter	KDCMSGLT
KCTERMN	kctermn	MT

- The MSGTAC program unit can use the administration program interface and it can issue administration commands. In particular, the MSGTAC program unit can issue the KDCS calls DADM and PADM to administer DPUT messages and printers; see the openUTM manual “Administering Applications”.

### Generation notes

- There can be only one MSGTAC program unit per application.
- The MSGTAC program unit must be defined in the TAC statement with  
TAC KDCMSGTC,PROGRAM=...

### Example of a MSGTAC program unit

The MSGTAC program unit NOHACK is designed to prevent unauthorized users from gaining access to a UTM application. If more than three invalid attempts are made to sign on via an LTERM partner (with an invalid user ID, an incorrect password or the wrong ID card), the connection to the terminal is to be aborted. This will require additional preparatory measures (see also the openUTM manual “Messages, Debugging and Diagnostics”).

- Preparations:
  1. Call the UTM tool KDCMMOD.
  2. Issue the GEN command, specifying the name of the message module.
  3. Use MODMSG commands to define MSGTAC as an additional destination for the messages K008, K033 and K094.
  4. Compile the source program written so far and link it to the application.
  5. Define the KDCDEF statement MESSAGE in the message module.
  6. Define KDCPTRMA in the TAC statement.

A more elegant solution would be to write the specifications for points 2 and 3 to a file and then to use this as an input file for point 1.

- Implementing the MSGTAC program unit:

The MSGTAC program unit NOHACK counts the number of incorrect attempts in a TLS. If openUTM accepts a sign-on attempt to the application (message K008 or K033), this TLS is deleted again.

If three consecutive incorrect attempts are followed by a fourth incorrect attempt, the corresponding terminal is to be released by means of "asynchronous administration". This happens with an FPUT call with KCRN = "KDCPTRMA" and a message area with the contents PTERM=pterm, ACT=DIS (see also the openUTM manual “Administering Applications”).

The administration command is then written with LPUT to the user log file and the TLS is deleted. Examples in C and COBOL are given on pages [519](#) and [572](#) respectively.

Each K message is read with FGET by the MSGTAC program unit. After one K message has been “processed”, FGET immediately reads the next K message within the same program unit run. A list of all the K messages is provided in the openUTM manual “Messages, Debugging and Diagnostics”.

### 8.4.3 The SIGNON service

A separate sign-on service can be generated for each transport system access point of a UTM application, see the SIGNON-TAC parameter of the BCAMAPPL statement and the TAC statement with the KDCSGNTC transaction code in the openUTM manual "Generating Applications".

The SIGNON sign-on service is a dialog service that is started

- after the connection to a terminal or a transport system client has been established by means of a transport system access point, provided a sign-on service has been generated for this access point, or
- before the start of every conversation initiated by the UPIC client, provided a sign-on service has been generated for this access point and sign-on services were released for UPIC clients at generation.

#### Generation notes

- The sign-on service for a transport system access point defined with MAX APPLNAME is generated with `TAC KDCSGNTC,PROGRAM=...`. This sign-on service thus becomes the default for all the transport system access points of the application.
- The SIGNON-TAC parameter of the BCAMAPPL statement can be used to generate another sign-on service for a transport system access point. If a sign-on service is not to be used for a transport system access point, the value \*NONE must be specified for the SIGNON-TAC parameter.
- Sign-on services are enabled for UPIC clients as follows:  
`SIGNON ...,UPIC=YES.`
- In the SIGNON statement in the generation you can specify that a user can still temporarily sign on to the UTM application if the validity period of his or her password has run out (operand GRACE = YES). In this case, the password of the user must be changed if the sign-on service is to end successfully.
- The maximum number of failed sign-on attempts from a client can be monitored as follows:  
`SIGNON ,SILENT-ALARM=number`

When this value is reached, message K094 is output to SYSLOG. This message can also be processed by an MSGTAC program (see [page 473](#)).

You will find a detailed description in the openUTM manual "Generating Applications".

### 8.4.3.1 Programming notes

The SIGNON service controls the sign-on procedure by means of a program. The sign-on service is controlled, above all, by the following KDCS calls (see the chapter “KDCS calls”):

SIGN ST	Queries the status of the sign-on service. The call also returns the result of the previous SIGN ON.
SIGN ON	Transfers authorization data to openUTM for checking. The call returns only whether the call was syntactically correct, not whether the sign-on was successful.
PEND PS	Specifies the point at which openUTM checks the authorization data and at which it should insert the intermediate dialog, if appropriate. Not until this call does a provisional sign-on take place provided the data transferred with SIGN ON were correct.

openUTM is shipped with an appropriate sample program. The openUTM manual “Using openUTM Applications” contains a description of this sample program and additional information on the concept behind the sign-on service and typical applications.

#### Notes

- In the first part of the sign-on procedure the KCBENID/kcuserid field in the KB header only contains blanks.
- If the sign-on is not completed successfully before PEND FI is executed, then openUTM clears the connection to the terminal or TS client, or it ends the UPIC conversation. In this manner, you can handle several unsuccessful attempts to sign on, e.g. because the user is not authorized, in a simple manner using PEND FI after KCRSIGN1 = U.
- If the sign-on service violates the rules applicable to it, then openUTM aborts the service with PEND ER. The connection to the terminal or to the TS application is cleared, and the connection to the UPIC client remains.
- If the UTM application is generated without user IDs (i.e. without USERS), the sign-on service can terminate immediately because the sign-on has been successful. The sign-on service receives the corresponding information at the SIGN ST call. However, an application-specific authorization check can also be carried out in the sign-on service (using a database with authorization data, for example).
- Restrictions in the sign-on service
  - The KDCS calls PEND RE/RS/SP are prohibited.
  - FPUT/DPUT calls and accesses to a ULS are not allowed before the sign-on is successful. Database calls and accesses to GSSBs and TLSs are only allowed before the sign-on is successful if the SIGNON statement explicitly allow this. You should therefore query the sign-on status (returned in KCRSIGN1) with SIGN ST to see if it contains the value A or R before you execute the accesses stated above.
  - You are not allowed to initiate calls for distributed processing.

### 8.4.3.2 Sign-on service for terminals

The sign-on service for terminals consists of two parts:

1st part: Read authorization from terminal and transfer to openUTM.

2nd part: Send confirmation (message and USER-specific start format for the terminal, for example) in the event of a correct sign-on.

Between the first and the second part, openUTM may conduct an intermediate dialog with the terminal in order to query further authorization data such as identification information or the password.

openUTM returns the status of the sign-on procedure in the KCRSIGN1 field for SIGN ST. The program unit decides on how to proceed after this point based on this status indicator. The following values are possible:

- KCRSIGN1 = C (connected)  
The terminal is connected to the application, but there is no user signed on yet. The user on the terminal is requested to enter his or her authorization data via an MPUT call. The program unit terminates itself with PEND KP. The follow-up program unit reads the authorization data with MGET, passes the data with SIGN ON to openUTM and terminates itself with PEND PS.
- KCRSIGN1 = I (incomplete)  
openUTM already knows the user ID, but still requires additional information (password, ID card). This information is queried in an intermediate dialog. The program unit must terminate itself with PEND PS (specify the follow-up TAC!), and then openUTM executes the intermediate dialog.
- KCRSIGN1 = A (accepted)  
The sign-on is correct. The user ID is entered in the KB header. If so desired, additional dialog steps can be inserted before the end of the service. The sign-on service is terminated with PEND FI or PEND FC. The final message is created by the service itself and is output using MPUT. If the message is directed to a terminal, then it can be created from the user-specific start format. The name of the start format is returned by openUTM when SIGN ST is called in the KCRMF/kcrfn field.  
When MPUT PM is called, and then PEND FI, the last dialog message of the last service is output, if such a message is available.

- KCRSIGN1 = R (restart)  
The sign-on is correct and a service restart is pending. The user ID is entered in the KB header.  
If desired, additional dialog steps can be inserted before the end of the service. The sign-on service must terminate itself with PEND FI. The service restart is initiated by calling MPUT PM, KCLM=0, KCMF/kcfn=blanks. In this case, openUTM outputs the last saved message of the interrupted service (screen restart on the terminal), or it starts the follow-up program unit or follow-up service when there is a local synchronization point after PEND SP/FC. The service that is open can be terminated abnormally by terminating it without using an MPUT call. A K017 message is sent to a TS or terminal partner.
- KCRSIGN1 = U (unsuccessful)  
openUTM did not accept the authorization data. A terminal sign-on service is still in the 1st part and must request the user to re-enter the authorization data. If the sign-on service terminates in this state, the connection to the terminal is cleared.

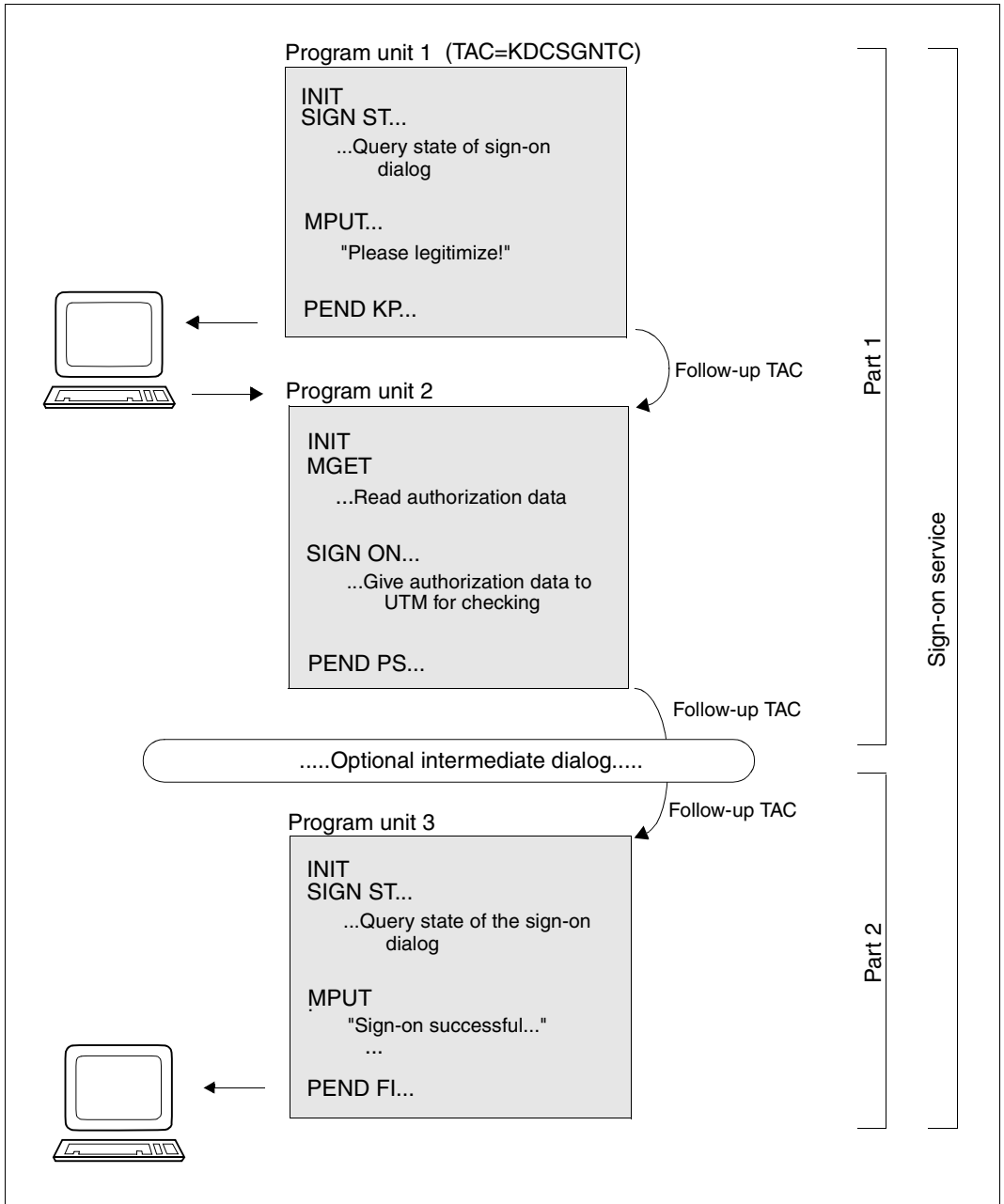
*LTERM partner with automatic KDCSIGN*

At the SIGN ST call, the sign-on service receives the information that the user ID is already known. Depending on the generation, an intermediate dialog can be conducted to request a magnetic strip card or a password.

**B** *Sign-on via router*

**B** If the router itself carries out a sign-on procedure, it forwards the authorization data to  
**B** openUTM by means of the PUTMUX protocol for checking. If the data is correct,  
**B** openUTM starts the sign-on service, which receives corresponding information with the  
**B** SIGN ST call. If the data is not correct, the sign-on service is not started and the router must  
**B** inform the terminal at the terminal.

The following diagram shows the sequence of execution of a sign-on service at the terminal:



Processing of a sign-on service for terminals



### 8.4.3.3 Sign-on service for UPIC clients or transport system clients

If the sign-on service is running for a UPIC client or a transport system client, then the sign-on service will **never** be located in the **first part** because, at a minimum, the connection user ID is assigned as the user ID. The following applies:

- In the case of TS applications, the user is signed on under the connection user ID or a user ID passed by means of a SIGN ON call. If the user is signed on under the connection user ID, the sign-on service can still sign the user on under a real user ID by means of the SIGN ON call when there is no service open for the connection user ID (KCRSIGN1=R).
- In the case of UPIC partners, the user is signed on under the connection user ID, the user ID passed in the UPIC protocol or the user ID passed by means of a SIGN ON call. If the user is signed on under the connection user ID, the sign-on service can still pass a real user ID in the SIGN ON call.

*The second part of the sign-on service*

You can use the SIGN ST call to query the status.

#### 1. KCRSIGN1= A or R

The sign-on was successful. The service is now assigned a user ID.

When a sign-on service starts for a transport system client, the connection user ID is signed on.

When a sign-on service starts for a UPIC client, either the connection user ID or a real user ID passed by the client in the UPIC protocol is signed on.

If the client is signed on under the connection user ID, the sign-on service can now pass a real user ID by means of the SIGN ON call.

For KCRSIGN1 = A

The sign-on service can terminate with PEND FI or PEND FC. The final message is created by the service itself and is output with MPUT.

If the sign-on service terminates with MPUT PM, KCLM=0, KCMF=SPACE and PEND FI, then openUTM outputs the last dialog message and terminates the conversation. If no message is available, then a "NULL message" is output and the conversation is terminated abnormally.

For KCRSIGN1 = R

The sign-on is correct, and there is no service restart. If desired, other dialog steps can be inserted before the end of the service. The sign-on service must terminate with PEND FI. The service restart is initiated by the call MPUT PM, KCLM=0, KCMF/kcfn=blanks. In this case, openUTM outputs the last saved message of the interrupted service (screen restart) or, in the case of a local synchronization point, starts the follow-up program unit or the follow-up service after PEND SP/FC.

If terminated without an MPUT call, the open service can be terminated abnormally. A UPIC client receives a CM\_DEALLOCATED\_ABEND, and message K017 is sent to a TS partner.

## 2. KCRSIGN1= U

The sign-on attempt was not successful, i.e. openUTM has not accepted the authorization data. If a sign-on service for a UPIC partner terminates in this state, then the conversation is terminated. If a sign-on service for a transport system client terminates itself in this state, then the connection is cleared.

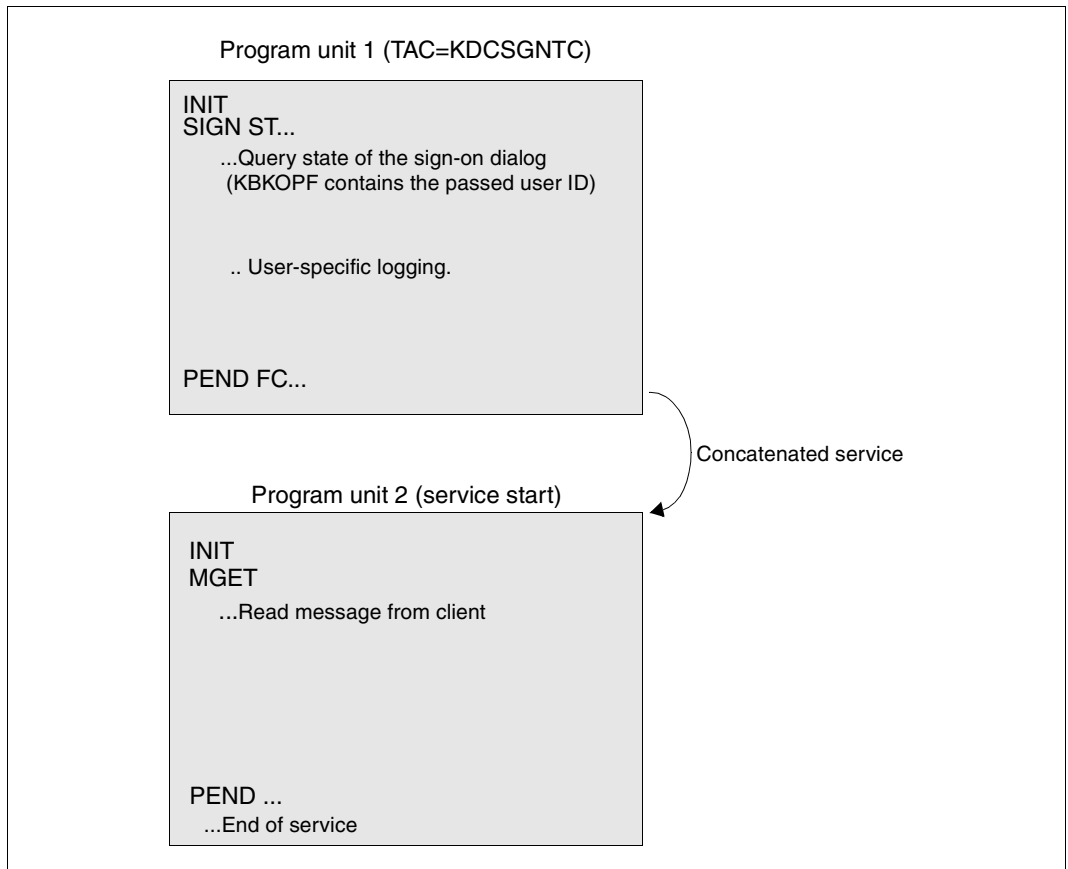
### **Particularities of the sign-on service for UPIC clients**

The sign-on service is started before the beginning of every conversation.

A PEND FI in the sign-on service after a successful sign-on terminates the sign-on service but not the conversation.

If a program unit of the sign-on service terminates after receiving a message from the UPIC client with PEND PA/PR, PS or FC without a preceding MPUT, the follow-up program unit specified in the KCRN field can read messages or message segments that have not yet been read. If the sign-on service is terminated with PEND FC without a preceding MPUT, the first program unit of the concatenated service receives the value F (first) rather than C (chained) as the service identifier in KBKOPF because it receives a message from the client.

The following diagram shows an example of the processing of a successful sign-on service via a UPIC client that passes the authorization data of a real user idea in the protocol field.



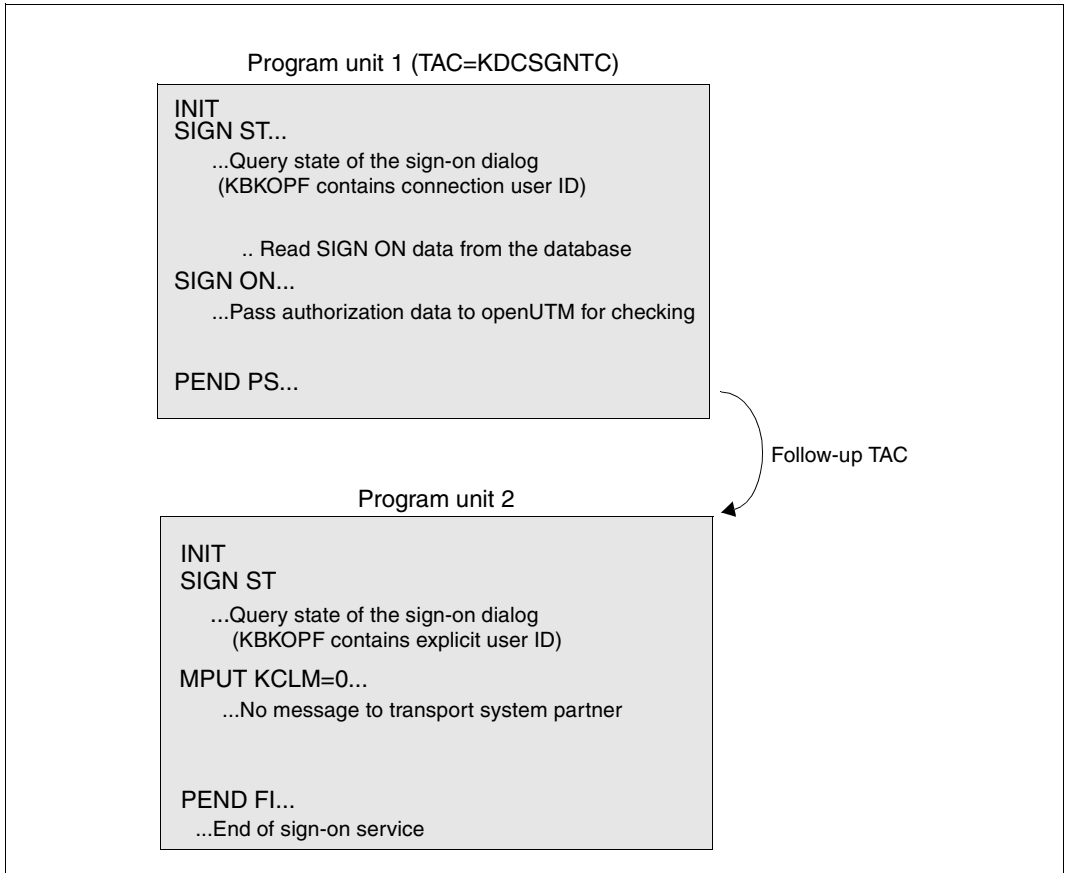
Processing of a sign-on service for UPIC clients

The sign-on service is terminated with **PEND FC** in program unit 1. The transaction code passed with **SIGN ST** is taken from the UPIC protocol as a follow-up TAC.

The concatenated service can then read the message from the client in program unit 2. In this way, UPIC clients can use the sign-on service without any need for reprogramming.

### Example of a sign-on service for TS applications

The following diagram shows the processing of a sign-on service via a TS application when the sign-on is successful under a real user ID:



Processing of a sign-on service for TS applications

### Sample programs for sign-on service

openUTM is shipped with program units as source programs that implement a complete sign-on service with a formatted interface to the terminal. This sign-on service is suitable for all generation variants. The format used contains English text.

The user can adapt this template to suit his or her requirements. A sign-on procedure with a formatted interface to the user is thus easily obtained. It is thus not necessary to start programming from scratch.

---

## 9 Additional information for C/C++

This chapter contains programming language-specific information as a supplement to the general information in chapters 1 to 8. You will need the information provided here when writing C or C++ program units:

In the first section you will learn about the structure of C/C++ program units. The second section contains sample programs. The third section contains a list of the data structures in *kcca.h*, *kcmac.h* and *kcpa.h*.

### 9.1 Program structure for C/C++ program units

In this section you will learn:

- how to create a UTM program unit as a subroutine
- how you must declare the data
- what the command should look like and how a KDCS call is programmed
- what kind of special platform-specific characteristics exist (for example, format system or compiler-specific dependencies).

#### 9.1.1 C/C++ program units as subroutines

UTM program units and event exits are subroutines of the UTM main routine. This has the following consequences:

- The program name defines the start address.
- A C/C++ program unit is defined as a function of type *void*.
- All formal parameters must be declared explicitly.
- The program unit is terminated dynamically with a PEND call; the event exits are an exception to this rule. They can be exited using the *return* statement. The *exit* statement must never be used.

A set of header files is provided for compatibility and to ensure that you are working with correct data structures. You use the same header files for C and C++ program units. The use of these header files is described in [section “Data structures for C/C++ program units” on page 494](#).

The include files *kcca.h*, *kcmac.h* and *kcpa.h* are located:

- B** – in BS2000/OSD, in *SYSLIB.UTM.061.C*
- X/W** – in Windows and Unix systems, in the *include* subdirectory of the UTM *utmpath* directory



If you cannot compile C programs because the first KDCS call parameter has the type *struct kc\_pa \** instead of the type *union kc\_paa \** then you should set the UTM\_OLDANSI option for the C precompiler in order to prevent checking.

### Program unit names

The name of a C program unit is also its start address.

This name is freely definable. It must be unique within an application program. Some names have already been reserved and for that reason may not be used.

You should observe the following points when choosing a name:

- B** ● For BS2000/OSD:
  - B** – All names beginning with KDC, KC and I are reserved and should be avoided.
- X/W** ● For Unix systems and Windows systems:
  - X/W** – All names beginning with KDC, KC, x or ITS are reserved.
  - X/W** – Names beginning with t\_ are reserved for PCMX.
  - X/W** – Names beginning with a\_, o\_ and s\_ are reserved for OSS.
- The name must conform to the C/C++ conventions.

You must also specify the program names (start address names) when generating the UTM application in the PROGRAM KDCDEF application for each name (see the openUTM manual “Generating Applications”).

## 9.1.2 Parameters of a C/C++ program unit

A C/C++ program unit contains at least one (but usually several) parameters that are passed in the form of addresses as follows.

```
[extern "C"] void cprog (kb [,spab] [,param_1] ...[,param_n])
```

<code>extern "C"</code>	is only necessary for C++ program units: You must identify C++ program units as external "C" links in your source code for openUTM, otherwise errors will occur during linking.
<code>void</code>	A C/C++ program unit is defined as a function of type <i>void</i> .
<code>cprog</code>	The name of the program unit. The name must be specified during the generation in the PROGRAM statement; see the openUTM manual "Generating Applications".
<code>kb</code>	The name of the communication area (KB). You may choose any name, but the communication area must then be declared under this name. The header file <code>kcca.h</code> is provided for use with the communication area.
<code>spab</code>	The name of the standard primary working area. You may choose any name, but the standard primary working area must then be declared under this name. The header file <code>kcpa.h</code> is provided for use with the KDCS parameter area.
<code>param_1 ... param_n</code>	These are the names of additional objects (AREAs) that must also be declared. In particular, these objects could be storage areas that serve to expand the standard primary working area. If these objects are not used, then this data does not have to be specified.

The `kcca.h`, `kcpa.h`, `kcapro.h` and `kcdf.h` header files are included implicitly when the `kcmac.h` header file is used. This means that you no longer have to explicitly specify these files as header files in the actual program. The definitions contained in these files are available at all times to the programmer.

The value `KDCS_SPACES` is defined in the include file `kcmac.h` for the transfer of blanks to Char arrays. The value `KDCS_NULL` is available for the transfer of the value "binary zero" to Char parameters.

### 9.1.3 Declaring data

You must explicitly declare all formal parameters. Please note the points described below when doing so.

#### 9.1.3.1 Communication area

Each program unit, including the event exits (exception: INPUT exit), must contain a data structure that describes the KDCS communication area. Use the *kcca.h* header file for this purpose.

#### 9.1.3.2 Standard primary working area

A C/C++ program unit generally also contains a data structure for the standard primary working area. If the standard primary working area is used by the program unit, then it should include the KDCS parameter area (*kcpa.h* header file). You should also store the KDCS message areas and other variable data in the standard primary working area.

If you do not store variable data in the standard primary working area, then you must make sure that the program unit is reentrant by storing it in the automatic area of the function, for example.

You must define the KDCS message areas yourself. For calls that request information from openUTM (KDCS\_INFO, KDCS\_INITPU, for example) there are specific data structures provided in the header files. If you are working with a formatting system, you can use automatically generated address assistants to format the KDCS message area (see the formatting system manual).



*Example 1*

The communication area also possesses a KB program area for data transfer to follow-up program units. The message area is located in the SPAB.

```
#include <kcmac.h>                                /* UTM data structures      */
#include <forma3a.h>                               /* Structure of the addressing */
                                                /* assistant for +format forma3 */

struct ca_area
{ struct ca_hdr ca_head;                          /* KB header                */
  struct ca_rti ca_return;                        /* KB return area           */
  struct ca_prog_area
  { char ca_info[22];                             /* Application-specific      */
    char ca_start[2];                             /* declaration of the KB    */
    char ca_dest[2];                              /* program area             */
    char ca_fl_day[5];
    char ca_fl_nr1[5];
    char ca_fl_nr2[5];
  } ca_prg;
};

struct work
{ union kc_paa param;                             /* KDCS parameter area     */
  struct msg_area
  { forma3a_std_mask;                             /* Declaration +format forma3 */
    ...
  } msg_a;                                         /* KDCS message area       */
};

void cprog (struct ca_area *ca, struct work *spab)
{.../* Start of the function area of the program unit */ ...
```

**9.1.3.3 Other data areas (AREAs)**

In addition to the communication area and SPAB, you can also pass other areas as parameters (see [page 87](#)). You create such an area in C/C++ as a source code file that only contains data definitions but no executable statements.

The following example will demonstrate how you define such areas in C/C++ and how to use them in your C/C++ programs.

**X/W Example with AREAs (Unix systems and Windows systems)**

X/W In the following, two areas are generated, defined in a C source and passed to a program unit. The following are defined:

- X/W – the area *area* for direct access (i.e. the data area is passed directly to the program unit)
- X/W – the area *areaind* for indirect access

**X/W *KDCDEF generation***

X/W AREA *area*,ACCESS=DIRECT  
 X/W AREA *areaind*,ACCESS=INDIRECT

**X/W *Creating a C/C++ source to supply the additional data areas***

X/W You define the data structures belonging to the areas in a C/C++ source as follows. In this example, the address of the area is set for *areaind* (i.e. for the area with indirect access) at compilation time.

```
X/W char area[20] = "Area direct ";
X/W static char area_ind[30] = "Area indirect ";
X/W char *areaind = &area_ind[0];
```

X/W You have to compile the C/C++ source with the C/C++ compiler and link the created object module to the program units.

X/W If you want to set the address of the area *areaind* during the application run, you have to define *areaind* in the C/C++ source as follows:

```
X/W char *areaind:
```

X/W During the application run (typically in the START event exit) you then have to supply *areaind* with the address of the area that you want to pass to the program units as a parameter – by means of the following statement, for example:

```
X/W static char area_ind[30] = "Area indirect ";
X/W areaind = &area_ind[0];
```

X/W This makes it possible, for example, to open a shared memory in the START event exit and store the address of the shared memory in the pointer variable *areaind*. The program units are thus able to access the shared memory.

**X/W *Access to the areas in the program units in Unix systems and Windows systems***

X/W A program unit in which you use the areas or one of the areas must be as below. Note that the order of the areas in the parameter list has to be the same as the order of the AREA statements at KDCDEF generation.

```

X/W ● Program unit in C:
X/W void areaprg (
X/W struct spab *spab ,
X/W struct kc_ca *kb ,
X/W char area1 [20] ,
X/W char area2 [30] )
X/W
X/W {
X/W printf ( BUFFER
X/W , "Hello world from UTM (Lterm = %.8s)\n"
X/W "Area is '%s' \n"
X/W "Area (indirect) is '%s' \n"
X/W , kb -> kopf.kclogter
X/W , area1
X/W , area2
X/W );
X/W .
X/W .
X/W .

```

### Alternatives to AREAs

If program units that use AREAs are to be transferred from one application to another application, then the use of AREAs can lead to problems due to possible differences in the parameter lists.

The following alternatives are available for C/C++ programs. You must determine whether the data area is stored in memory local to the process or in shared memory under Unix systems or in the memory mapped file under Windows systems.

- Data areas in local process memory  
You only need to provide a global C/C++ data structure for each data area that other modules can access using the same definition and the 'external' storage class attribute.

```

X/W – For Unix systems in shared memory and for Windows systems in the memory mapped
X/W file:

```

```

X/W In the start exit you must provide a data structure (see example 3):

```

```

X/W static DataAreas struct {
X/W     struct table *TABLE1;
X/W     struct table *TABLE2;
X/W     struct table *TABLE3;
X/W };
X/W :
X/W struct DataAreas ExternDataAreas;

```

X/W The shared memory area will then be requested in the start exit and the addresses will  
 X/W be set in the ExternDataAreas data structure. Other program units will access the data  
 X/W structure with:

X/W extern struct DataAreas ExternDataAreas;

B – BS2000

B You must provide a data area (see the example) which has been written in Assembler,  
 B for example:

```
B TABLE1 CSECT PUBLIC
B         DS CL64
B         END
```

B This CSECT is compiled and may be linked with other modules. The module (without  
 B an additional module that has code linked to it) should be named MTABLE1; you should  
 B then generate it as a load module:

```
B LOAD-MODULE MTABLE1
B             ,LOAD-MODE = ( POOL , poolname , NO-PRIVATE-SLICE ) -
B             , ...
```

B You then make it possible for a program unit or event exit to access the data area with:

B extern struct table TABLE1;

B This program unit or event exit must then be linked dynamically afterwards.

*Example 2*

The areas TABLE1, TABLE2 and TABLE3 have been defined in this order with the AREA statement. TABLE3 is required in a program unit. TABLE1, TABLE2 and TABLE3 have the same structure and are defined as follows:

```
struct TABLE
{ int no;
  int tag;
  char name[20];
  char company[20];
  int order_no;
  int quantity;
  float price;
  int discount;
};
```

The addresses of these areas are passed as follows:

```
.
.
#include <kcmac.h>                               /* UTM data structures */
.
  struct ca_area
  {....};
  struct work
  {....};
struct TABLE
  {... };
.
.
void cprog (struct ca_area *ca, struct work *spab, struct TABLE *TABLE1,
           struct TABLE *TABLE2, struct TABLE *TABLE3)
{ ...
.
.
```

### 9.1.4 Data structures for C/C++ program units

In order to structure the data areas, the following header files containing predefined data structures are supplied with openUTM.

**B** The data structures are present in the library SYSLIB.UTM.061.C.

**X/W** The data structures are present in the *include* directory in the UTM directory *utmpath*.

**B**  
**B**  
**B**

**B**  
**B**  
**B**

Name	Contents and meaning
kcapro.h	Optional second parameter area for the APRO call: This area is used to select special combinations of OSI TP functions and the type of security. <i>kcapro.h</i> is included by <i>kcmac.h</i> .
kcat.h	KDCS attribute functions: When using +formats, you can change the attribute field formats with the symbolic names for attribute functions.
kcca.h	The data structure for the KDCS communication area; this area contains: – current service and program data, – return data from a UTM call and – if desired, the communication area program area for passing data between programs in a service. You must also define the fields of the communication area program area. <i>kcca.h</i> is included by <i>kcmac.h</i> .
kccf.h	Defines the second parameter passed by openUTM during the INPUT event exit. In this parameter openUTM passes the contents of the control fields of screen formats to the program unit. This second parameter is also called the control field area for this reason.
kcdad.h	Data structure for the DADM call: You should place this data structure in the KDCS message area during a DADM RQ KDCS call.
kcdf.h	KDCS screen functions: Using these symbolic names, you can influence the screen output by placing the name of the desired function in the KCDF field of the KDCS parameter area. <i>kcdf.h</i> is included by <i>kcmac.h</i> .
kcinf.h	Data structure for the INFO call: You should place this data structure in the KDCS message area during a INFO DT/SI/PC KDCS call.
kcini.h	Defines a second parameter area for the INIT call (only required for INIT PU): openUTM returns the information queried with the INIT PU call in this parameter area.
kcinp.h	Data structure for the INPUT exit: This data structure contains the input and output parameters of the INPUT exit.

Name	Contents and meaning
kcmac.h	KDCS macro interface for C/C++: This file contains all macros in the C/C++ macro interface as well as the include statements for the <i>kcapro.h</i> , <i>kcca.h</i> , <i>kcd.f.h</i> and <i>kcpa.h</i> header files.
kcmshg.h	Data structure for the UTM messages: You will need this data structure if you handle UTM messages in an MSGTAC routine or when you want to evaluate the SYSLOG file using your own program.
kcpa.h	Data structure for the KDCS parameter area: This area contains the parameters of a KDCS call. <i>kcpa.h</i> is included by <i>kcmac.h</i> .
kcpad.h	Data structure for the PADM call: You should place this data structure in the KDCS message area during a PADM AI/PI KDCS call.
kcsgst.h	Data structure for the SIGN call: You should place this data structure over the message area when issuing the KDCS call SIGN ST with KCLA > 0.

Insert the data structures you will use with *#include* before the call to the program unit. You must explicitly declare the corresponding areas (communication area, KDCS parameter area,...) in the program unit.

### Example 3

```

/* insert constants and data structures */

#include <kcmac.h>                /* UTM data structures      */
#include <kcinf.h>

    struct ca_area {...};

    struct work
    { union kc_paa param;
      struct msg_area
      { struct kc_dttm  info_time;    /* area for INFO DT      */
        struct kc_sysinf info_sys;    /* area for INFO SI      */
        char text[200];
      } msg_a;
    };
void cprog (struct ca_area *ca, struct work *spab)

```

## 9.1.5 Command section of a C/C++ program unit

You can design the command section of a C/C++ program unit any way you want. You merely need to abide by a few rules pertaining to transaction processing as described in detail in [chapter "Structure and use of UTM programs" on page 29](#):

- the program units are subroutines of the UTM main routine KDCROOT
- the program units must be reentrant
- dialog program units must observe the strict rules of the dialog.

KDCROOT is used to designate the UTM main routine. The source program for KDCROOT is created with the KDCDEF generation tool; see the openUTM manual "Generating Applications".

There are special rules for event exits that will be described in [section "Event exits" on page 502](#).

### Local classes in C++ program units

If a local class is declared in a C++ program, then the destructor for this class may only be executed if the class is in a block whose block terminator "}" is reached before the PEND call. This restriction does not apply under LINUX.

Recommendation: Local classes should be used within their own "inner" block.

#### *Example*

```
//extract of cpphello.C in sample Application
extern "C" void cpphello (struct kc_ca *kb, struct work *spab)
{
    {
        Demo Autoclass('A');
        // further code using Autoclass
    }
    // reached after destructure call for Autoclass
:
:
    /* PEND-FI - Call */
    KDCS_PENDFI();
}
```

### KDCS calls in C/C++ program units

KDCS allows you to use the using the C/C++ macro interface, which makes it easy to supply the parameters with data (see [page 497](#)) when calling UTM functions.



### 9.1.6 C/C++ macro interface

To make the passing of parameters through the KDCS interface easier, macros are provided in the *kcmac.h* header file for each of the KDCS calls. These macros contain all the data specifications required for a KDCS call as macro parameters. These macros execute the desired call and then provide a return code (see also [page 503](#)).

The names and characteristics of the various macros and how the macro names were created will be explained in the following section. The preparations you must make in order to utilize the *kcmac.h* header file are discussed. You will find an example of a macro call and a program listing of an executable KDCS program that will clarify the use of the *kcmac.h* header file at the end of the section.

#### Using KDCS\_SET to prepare for a call

In each compilation unit in which you want to call a KDCS macro, you must execute the following two actions before the first macro call:

- Copy the include file *kcmac.h* into the compilation unit
- Copy the *kcmac.h* header file into the program unit

The four header files *kcca.h*, *kcpa.h*, *kcapro.h* and *kcdf.h* are used implicitly when *kcmac.h* is used, i.e. the definitions therein are available to the programmer at all times.

- Call the KDCS\_SET(pb,hdr,rti) macro for initialize the areas
  - pb     pointer to the KDCS parameter area.
  - hdr    pointer to the header area of the communication area (ca\_hdr data structure).
  - rti    pointer to the return area of the communication area (ca\_rti data structure).

The KDCS\_SET call establishes the connection between the KDCS macros and the programmer's definitions for the parameter area and the communication area. You can call this macro more than once. It is therefore possible to use several different parameter areas, for example.

## Macro names

The names of the macros are created according to the following rules:

A KDCS macro name always begins with the prefix "KDCS\_". The operation code of the desired KDCS call follows in upper case letters. If necessary, the operation modifier is appended to the operation code (also in uppercase letters).

Examples:     KDCS\_INIT, KDCS\_LPUT, KDCS\_MGET  
                   KDCS\_MPUTNT, KDCS\_PENDFI, KDCS\_SPUTMS

The DPUT calls using a "+T", "-T", "+I" or "-I" as the kcom parameter represent an exception to these rules. These macros are named: KDCS\_DPUTPT, KDCS\_DPUTMT, KDCS\_DPUTPI and KDCS\_DPUTMI.

Other exceptions are the APRO calls used to address OSI TP partners:  
 KDCS\_APRODM\_OSI / KDCS\_APROAM\_OSI/  
 KDCS\_APRODM\_OSI\_O / KDCS\_APROAM\_OSI\_O

## Macro parameters

The macro parameters are each named after the KDCS parameters for which they will contain values (*kern, kclt, kchour, kcpj,...*). The KDCS message area is named *nb* and, if it is needed, is always specified as the first parameter.

There are three different types of KDCS parameters: character arrays, letters and numbers:

- Character arrays

In the KDCS interface these parameters are either two, three or eight characters long. In the C/C++ macro interface these parameters are specified as pointers to a C string of any length.

Internally, these parameters are converted to arrays of characters of the required length by either appending any eventually missing spaces to the end of the C string or by ignoring all superfluous characters at the end of the C string. In this manner, for example, for an array of length 8, "\_\_\_\_" and "" are converted to the same string, namely eight spaces: "\_\_\_\_\_".

A string such as "xyz" is converted to "xyz\_\_\_\_\_". The conversion is performed without adding an end of string character ("\0").

Parameters of this type are: *kern, kcfj, kclt, kcpa, kcus, kcadrlt, kcacl, kcpj, kcpo, kcneg, kccomid, kclangid, kcterrid, kccsname*.

- Letters

In the KDCS interface these parameters are of type character.

In the C/C++ macro interface these parameters are also specified as characters (per value). The parameters are specified in the form: ' ', 'A', 'C', etc.

Parameters of this type are: *kcmof*, *kcof*.

- Numbers

There are different types of numbers in the KDCS interface: short, unsigned short and numbers that are expected to be in the form of printable text.

In the C/C++ macro interface these parameters are always specified as numbers (per value). These numbers are converted internally to the desired format. Short and unsigned short numbers are passed directly. Numbers that must be provided as text are converted.

Parameters of this type are: *kcla*, *kclm*, *kcdf*, *kclcapa*, *kclspa*, *kcdlay*, *kchour*, *kcmmin*, *kcsec*, *kcli*.

### Simplified parameter passing

The macros know the required length of every parameter and ensure that they are passed correctly (short strings are padded with spaces up to the required length). A parameter only needs to be specified in the macro parameter list and does not have to be passed anymore using *memcpy*. The macros also process all data that was previously processed with *memcpy*. The assignments of the parameters to the appropriate KDCS parameter fields is done implicitly by the C/C++ macros. The parameter types are determined by the C/C++ macro definitions.

For comparison, here is an example of supplying the *kcrn* parameter with data:

– In a direct call:

```
memcpy(pb.kcrn,"rnam",8) /* only kcrn is supplied with data*/
```

– When using a macro:

```
macroname(...,"rnam",...) /* complete KDCS call*/
```

Any parameter fields not used are implicitly set to binary zero. You specify the constant *KDCS\_SPACES* for the fields that are to be set to spaces in a call.

Some of the KDCS parameters that were previously passed as an array of characters are usually stored as integers in C. Such parameters are, for example, the parameters used to specify the time and date: *kcday*, *kchour*, *kcmín*, *kcsec*. Such parameters are expected to be passed as numbers (integers) in the new macro calls. The macros ensure that the data is passed correctly to the interface.

Here is an example of supplying the KDCS call with a time specification for comparison:

- In a direct call:

```
memcpy(pb.kcext.kcdput.kcday,"003",3);
memcpy(pb.kcext.kcdput.kchour,"11",2);
memcpy(pb.kcext.kcdput.kcmín,"55",2);
memcpy(pb.kcext.kcdput.kcsec,"00",2);
```

- When using a macros:

```
macroname(...,3,11,55,0)
```

### Format of the KDCS call through the C/C++ macro interface

KDCS calls using the C/C++ macro interface have the following format:

```
KDCS_operation_code[operation_modifier](parameter_list)
```

If a KDCS message area is required, then its address is always specified as the first parameter. The parameter list can also be empty.

There are a few exceptions to the macro name format (see [page 498](#)).

Examples: `KDCS_INIT (length_of_communication_area_program_area, length_of_standard_primary_working_area);`  
`KDCS_SGETRL (pointer_to_message_area, message_length, name_of_local_secondary_storage_area)`  
`KDCS_PGWTKP()`

### Example: KDCS\_MPUTNT macro call

In the following the use of a KDCS macro is demonstrated with an example of an MPUT NT call. First, a description of the macros:

`KDCS_MPUT NT(nb,kclm,kcrn,kcfn,kcdf)`

<code>char *nb</code>	pointer to the KDCS message area
<code>short kclm</code>	length
<code>char kcrn[8]</code>	space/TAC/service ID
<code>char kcfn[8]</code>	format/space/edit profile
<code>unsigned short kcdf</code>	screen fill character / binary zero / ---

The `KDCS_MPUTNT` macro requires five parameters, one of which is a pointer to the KDCS message area *nb*. This parameter is always passed as the first parameter. The other parameters of the macro originate in the KDCS parameter area. The required type and meaning of each parameter is specified (for example, *kclm* is of type *short* and specifies the length of the area in which the message is to be passed). In the following example, *NB* is a pointer to the KDCS message area used and *kc\_pa* is the KDCS parameter area.

A typical macro call looks like the following call, for example:

```
KDCS_MPUTNT(NB,10,KDCS_SPACES,KDCS_SPACES,KCNODF);
```

The same call without using a macro looks like (direct KDCS interface call):

```
memcpy(PB.kcop,MPUT,4);
memcpy(PB.kcom,NT,2);
kc_pa.kclm=10;
memcpy(kc_pa.kcrn,"",8);
memcpy(kc_pa.kcfn,"",8);
kc_pa.kcdf=0;
KDCS(&kc_pa,NB);
```

## DEBUG function

You can enable a logging function during run time for the KDCS calls triggered by the macros. The calling module, the source code line and the `KCOP`, `KCOM`, `KCRCCC` and `KCDCDC` KDCS fields are recorded.

The logging function is switched on as follows:

X/W *Unix systems and Windows systems:*  
 X/W by setting the `KDCS_C_DEBUG` environment variable  
 X/W The information logged is written to *stdout*.

B *BS2000/OSD:*  
 B by setting the `*KDCSCDB JOB VARIABLE LINK`  
 B The information logged is written to `SYSOUT`.

B Please note that the job variable link must point to a non-empty job variable.

B *Example:*  
 B `/CREATE-JV JV-NAME = KDCSCDB`  
 B `/MODIFY-JV JV-CONTENTS = KDCSCDB, SET-VALUE = 'YES'`  
 B `/SET-JV-LINK LINK-NAME = *KDCSCDB, JV-NAME = KDCSCDB`

You will find additional information on this subject in the openUTM manual “Messages, Debugging and Diagnostics”.

**Note: Macros as statement follow-ups**

The KDCS macros are not simply individual C/C++ functions, rather they are a series of several statements. This means that a single macro must be handled exactly as would be required for a series of several statements. This peculiarity is normally not relevant, and the macros can be used like normal C/C++ functions. There are, however, program structures that require single statements, for example within an if statement:

```
if (condition) STATEMENT else STATEMENT;
```

Because the macros consist of several individual statements, it is not possible simply to use a macro as if it were one (single) STATEMENT. You must therefore first designate the macro as a unified block. This is done by using curly brackets: {macro;}. Instead of incorrectly writing

```
if (condition) macro; else macro;          /* W R O N G !!!! */
```

you should write:

```
if (condition) {macro;} else {macro;}     /* C O R R E C T ! */
```

**9.1.7 Event exits**

The INPUT, START, SHUT and VORGANG event exits may not contain any KDCS calls. They should be written as subroutines and must end with the *return* statement.

For START, SHUT and VORGANG the addresses of the communication area (KB) and standard primary working area (SPAB) are passed as parameters; these areas must be declared accordingly (just like for program units containing KDCS calls). On [page 533ff](#) you will find an example of a combined START/SHUT exit.

openUTM passes an address for the INPUT exit. This address specifies the INPUT parameter area. The header file *kcinp.h* contains the structure of the INPUT parameter area; the name of the data structure is *kc\_inp*. On [page 516ff](#) you will find an example of a INPUT exit.

There can be a maximum of eight START and eight SHUT event exits per application. There can only be one INPUT and one VORGANG.

### 9.1.8 Programming the KDCS error handling routines

The *krccc* return code in the return area of the communication area is returned as a three-character field in the C/C++ KDCS interface. If the C/C++ macros are not used, then the values must be compared using the *strncmp* function.

The request for the return code is simpler when the macro interface is used:

The *kmac.h* header file defines a return code in the static variable *long* KCRCC that contains the integer value of the *krccc* field after each macro call. Determining if an error occurred during the call is then limited to checking KCRCC.

For comparison, here are two examples of determining if an error has occurred after a KDCS call:

- for a direct call:

```
if(strncmp(kb->rti.krccc,"000",3)!=0) ...
```

- when using a macro:

```
if (KCRCC!=0) ...
```

To ensure a better diagnosis of a problem, all unused parameters are automatically set to binary zero before a KDCS call when the macros are used.

In this manner, the entire parameter area contains a precisely defined set of data. If an error occurs, the parameter area can be specifically checked and any deviation from the expected contents can be detected.

### 9.1.9 Modifying KDCS attributes

If you use `+formats` or `#formats`, you can change the attributes of format fields in the program.

The KDCS attribute combinations are contained in the header file `kcat.h` and are copied into the program unit via include statements.

The `kcat.h` header file also contains the `KDCATTR` macro with which you can set the KDCS attribute combinations. `KDCATTR` is called as follows:

```
KDCATTR (attribute_field, attribute_value);
```

The names and characteristics of the possible KDCS attribute combinations are listed in your formatting system manual.

#### Example

The "name" field is to be output to the screen as a protected field; `a_name` is the attribute field corresponding to this.

```
KDCATTR (spab->std_mask.a_name, KCPROT);
```

The `+format "FORMAT5"` contains the "FELD1" to "FELD5" fields. FELD4 should be sent to the screen blinking, all other fields maintain the attributes from the format description.

```
#include <kcat.h>
unsigned short mput_features;
.
.
{... a_format5 mask_out; ...} *spab; 1)
.
/*MPUT call */
.
.
KDCATTR (mask_out.a_field4, KCSIGN);
KDCATTR (mask_out.a_field1, KCNOATTR);
KDCATTR (mask_out.a_field2, KCNOATTR);
KDCATTR (mask_out.a_field3, KCNOATTR);
KDCATTR (mask_out.a_field5, KCNOATTR);
.
KDCS_MPUTNT (&mask_out, sizeof(a_format5),
             KDCS_SPACES, KDCS_SPACES, mput_features);
```

<sup>1)</sup> Declares the addressing assistants for the `+format "format5"`.



## 9.1.10 Platform-specific characteristics for BS2000/OSD

### B Compiling C/C++ program units

B If a link and load module (LLM) is to be created that consists of a code CSECT and a data CSECT section at compilation of C/C++ program units, you must set MODULE-GENERATION(MODULE-FORMAT=LLM) for the COMPILER-ACTION option when calling the compiler. The LLM must be made available in a PLAM library for linking with the BINDER. Object modules can also be created for your C program units (type=R in LMS).

### B Using shareable code

B If you plan on loading C/C++ program segments as shareable code, then you must set the following option when compiling:

B `COMPILER-ACTION=MODULE-GENERATION(SHAREABLE-CODE=YES, . . .)`

B The shareable code does not have to be stored in its own object module, rather it can be placed together with the non-shareable segment in an LLM that is divided into a public and a private slice.

B The shareable program segments only need to be loaded together once for all processes (tasks) of the application(s). Then only the non-shareable segments need to be loaded into the local task memory.

B openUTM offers several ways of loading shareable objects:

- B – as a non-privileged subsystem
- B – with the ADD-SHARED-PROGRAM command into system memory
- B – into a common memory pool in user memory (class 6 memory).

B You can find additional information on compiling shareable code in the User's Manual for your compiler. The openUTM manual "Generating Applications" as well as the openUTM manual "Using openUTM Applications under BS2000/OSD" contains detailed information on linking and loading shareable code.

### B Creating formats with the IFG

B A detailed explanation of how to create formats using the IFG can be found in the IFG manual. Please follow the guidelines below when creating these formats for use with openUTM:

- B ● The format name may be at most 7 characters long.
- B ● Select "Structure of the data transfer area" in the user profile:
  - B – for #formats: separated attribute blocks and field contents
  - B – for \*formats: unformatted, without attribute fields
  - B – for +formats: unformatted, with attribute fields

- For the C/C++ programming languages the IFG always creates **one** address assistant. Fields with the "arithmetic field data type" are represented as character strings (char [...]).
- Please note when defining the addressing assistants for \*formats and +formats that openUTM deletes the transaction code from the UTM message for MGET and FGET (as long as this is not explicitly prevented in an INPUT exit). If the first field in the format contains the transaction code, then you can take this into account by reading the UTM message into the second field.

### B *Example*

```

B      struct work
B      { union kc_paa param;
B          FORM1 std_mask;           /* declare addressing aid for */
B          .                         /* the *format FORM1      */
B      } *spab;
B          .
B          .
B
B      /* MGET call */
B
B      KDCS_MGET ( spab->std_mask.FUNCTION      1)
B                  ,sizeof( FORM1 )
B                  ,"*FORM1 "                );
B          .
B      /* MPUT call */
B
B      KDCS_MPUTNT (&spab->std_mask,sizeof(FORM1)
B                  ,KDCS_SPACES,"*FORM1 " ,KCNO DF);
B
B      /* PEND FI call */
B          .

```

1) FUNCTION is the second input field of the format.

- When preparing for the application, you bring the formats into the format application field (format library). You specify these names together with the FHS start parameters.

### B **Extended line mode**

B You must define the control characters yourself when working in extended line mode if they are not available in C/C++. You will find a list of which control characters correspond to which hexadecimal values in the "TIAM User Guide", for example in the description of the VTCSET macro.

### 9.1.11 Platform-specific characteristics in Unix systems

#### X Shared objects

X You must observe the following guidelines when linking utmwork to shared objects:

X – You must always link utmwork with the *libwork.so* or *libwork.sl* (HP-PARISC) library so  
X that, for example, the "KDCS" entry can be found by the shared objects dynamically  
X loaded thereafter.

X – If COBOL program units are dynamically loaded later as shared objects, then the  
X COBOL run-time system must also be linked as a shared object in the program unit.  
X Otherwise, the program unit will not be able to find the entries of the COBOL runtime  
X system.

X – The shared object should be created with one of the following commands, depending  
X on the platform:

X `cc -shared -o shared-object.so ...(LINUX)`

X `ld -b -o shared-object.sl ... (HP-PARISC)`

X `cc -G -o shared-object.so ...(for all other Unix platforms)`

X In the case of Solaris (64-bit) it is also necessary to specify `-xcode=pic32`.

X This ensures that the C runtime system is also linked to the program unit.

X – The shared object for C++ should be generated with one of the following commands  
X depending on the platform:

X `cc -shared -o shared-object.so ...(Linux)`

X `cc -G -b -o shared-object.sl ...(HP-PARISC)`

X `cc -G -o shared-object.so Module ...-lCstd .....(SUN)`

X `cc -G -o shared-object.so ...(for all other Unix platforms)`

X This ensures that the Unix C++ runtime system is also included.

#### X Signal handling

X Program units in C have limited usage of signals in Unix systems. The SIGUSR1 and  
X SIGUSR2 signals can be trapped by the C program unit in the start exit of a work process.  
X All other signals are handled by the openUTM system code itself.

**X CMX interface in utmwork**

- X You must note the following when using CMX calls in C/C++ program units:
- X For technical reasons, a CMX simulation is used in the UTM library `libwork.a` and `libwork.so` in UTM applications with OSI TP. It is therefore not possible in this case to use CMX calls in a C/C++ program unit.

**X Calling the `fork()` function in a C/C++ program unit**

- X You must note the following when calling the `fork()` function in C/C++ program units:
- X No openUTM program interfaces may be used in a child process created using `fork()`.
- X Otherwise, the UTM application will terminate abnormally.

**9.1.12 Platform-specific characteristics in Windows systems**

- W Under Windows systems, the projects must be created with Visual Studio as of Version 2005.

- W Signals are not supported in Windows systems.

**W Dynamic loading of application programs from DLLs**

- W This functionality allows you to add application programs dynamically and replace them during operation.

- W Dynamically loaded DLLs are generated as under Unix systems with the following statement:

```
W SHARED-OBJECT dll_name, DIRECTORY= ..., LOAD= ... VERSION=...
```

- W The operands must be supplied as under Unix systems. Note that *dll\_name* must have the extension `.dll`. It is imperative that you specify `VERSION`.

- W The `%LD_LIBRARY_PATH%` and `%PATH%` environment variables are not used with the `DIRECTORY` operand.

- W You will find detailed information on generation in the openUTM manual "Generating Applications" in the section on the `SHARED-OBJECT` statement.

**W *Creating DLLs***

- W DLLs are created using Microsoft Visual Studio as of Version 2005.

- W To do this, proceed as follows:

- W – Select DLL (dynamic link library) as the project type.

W – Insert the `declspec(dllexport)` statement in all the application programs.  
 W *Example:*  
 W `void_declspec(dllexport) func(struct kc_ca *kb, struct work *spab)`

W – Link to the `%UTMPATH%\sys\libwork.lib` import library.

### W *Use of DLLs*

W DLLs are searched for under the directory specified in the SHARED-OBJECT statement.  
 W The `%PATH%` and `%LD_LIBRARY_PATH%` environment variables are not evaluated.

W When the DLL is used, openUTM generates self-explanatory K078 messages.

### W *Replacing and administering DLLs*

W Programs in DLLs can be added to an application dynamically and replaced during  
 W operation in the same way that programs in shared objects can be under Unix systems.

W The description of KCDPROG in the openUTM manual “Using openUTM Applications  
 W under Unix Systems and Windows Systems” on the replacement of shared objects also  
 W applies by analogy to DLLs.

### W **NLS - Native Language Support**

W The `%LANG%` environment variable allows you to select the language in which the UTM  
 W messages are output.

W The values “De” for German and “En” for English are supported. Any other values are  
 W treated as “En”.

### W **Changing message destinations and message texts**

W To change message destinations, as under Unix systems you have to use the `kdcmmmod`  
 W program to create a separate message module that must be compiled by the C compiler  
 W and linked to the `utmwork` process.

W Message texts cannot be changed. The message texts from the DLLs supplied with the  
 W product are always used. These are stored in the `%UTMPATH%\n1s\msg\%LANG%` directories.

## 9.2 Programming examples in C/C++

In this section you will find examples of code for individual KDCS calls executed through the C/C++ macro interface, examples of complete C programs, an INPUT exit, an MSGTAC event service and an example of a complete UTM application.

### 9.2.1 Examples of individual KDCS calls

In this section you will find examples of code for the following KDCS calls:

- MGET
- MPUT
- DPUT
- MCOM with DPUT in a job complex
- APRO with MPUT for distributed processing

Because the rest of the KDCS calls are programmed analogously, we will not explicitly present all of the calls.

In a KDCS call *&pa* designates the address of the KDCS parameter area and *&ma* the address of the KDCS message area.

**MGET call**

- An unformatted, 80 byte long dialog message is to be received. If the message received is too short due to an error, then a request for new input is sent.

```

KDCS_MGET (&ma,80,KDCS_SPACES);
if (KCRCC != 0)
    mget_error();
if ( pa.kc1a > ca->ca_return.kcr1m)
    r_mput ();

```

In the *r\_mput ()* routine a request to repeat the input is sent to the terminal with MPUT.

- The "FORM15" format was requested by a terminal. The length of the unprotected data is 500 characters in various format fields. This format should be received in the program. FORM15 was declared as a *std\_mask* in the program.

```

KDCS_MGET (&ma.std_mask,500,"*FORM15 ");
if (KCRCC == 5) /* invalid format ID */
    format_error();
if (KCRCC != 0)
    mget_error();

```

In the *format\_error* routine the format must be output again in order to continue working with the correct format.

- In a running service an input which consists of a short message, created with the F2 function key, as well as 10 characters of data can be entered. The input should trigger a special function. The F2 key was assigned the 21Z return code during generation.

```

KDCS_MGET (&ma,input_lth,dev.features);
if (KCRCC == 21 ) /* return code for F2 */
{
    KDCS_MGET (&ma,10,KDCS_SPACES);
if (KCRCC != 0 )
    mget_error();
}

```

**MPUT call**

- An unformatted, 80 byte long UTM message is to be sent to the terminal.

```
KDCS_MPUTNE (&ma,80,KDCS_SPACES,KDCS_SPACES,KCNODF);
if (KCRCC != 0 )
    mput_error();
```

- The last UTM message in a service is to be sent to a terminal in the format mode. The name of the \*format is "FORM15". The screen should be cleared beforehand.

```
KDCS_MPUTNE (&ma,500,KDCS_SPACES,"*FORM15 ",KCREPL);
if (KCRCC != 0 )
    mput_error();
```

REPLACE is executed by default during a format change. The output is performed in order to rule out the possibility of an error due to the undefined contents of a field.

- In a "FORM10" \*format that is still on the terminal according to the last input, all unprotected fields are to be erased as a result.

```
KDCS_MPUTNE (&ma,0,KDCS_SPACES,"*FORM10 ",KCERAS);
if (KCRCC != 0 )
    mput_error();
```

**DPUT call**

- A queued job with an 11 character long UTM message is to be sent on Nov.11 (= 315th day of the year) at 11:11 AM to a program unit (absolute time). The TAC name is "ALAAF".

```
KDCS_DPUTNE (&ma,11,"ALAAF ",KDCS_SPACES,0,'A',315,11,11,0);
if (KCRCC != 0 )
    dput_error(); /* A = absolute time */
```

- An 80 character long queued message is to be output on terminal 'DSS1' in 1 hour (relative time). The 'acoustic alarm' (BEL) screen function should be triggered when the message is output.

```
KDCS_DPUTNE (&ma,80,"DSS1 ",KDCS_SPACES,KCALARM,'R',0,1,0,0);
if (KCRCC != 0 )
    dput_error(); /* R = relative time */
```



**Job complex: MCOM and DPUT call**

A formatted queued message (200 bytes) is to be sent to the printer PRINTER2 on the same day at 18:00. The printer confirmation is handled in the program.

For a positive confirmation an asynchronous program receives a confirmation job using the PRINTPOS TAC with a 20 byte long message, for a negative confirmation an asynchronous program is started with the TAC PRINTNEG (without a message). For a negative confirmation an 80-byte piece of user information is logged; it can be read using DADM UI.

The job complex is framed by two MCOM calls; the target of the print job (=base job) and confirmation jobs are set in the call MCOM BC; the complex identification is "\*PRICOMP".

```

/* Begin of the complex                                     */
    KDCS_MCOMBC ("PRINTER2","PRINTPOS","PRINTNEG","*PRICOMP");
    if (KCRCC != 0 )
        mcom_error();
/* DPUT-message for printer                                 */
    KDCS_DPUTNE (&ma1,200,"*PRICOMP","*FORM1 ",KCNODF,'A',
                ca->ca_head->kccv_doy,18,0,0);
    if (KCRCC != 0 )
        dput_error();
/* acknowledgment job in positive case                     */
    KDCS_DPUTPT (&ma2,20,"*PRICOMP");
    if (KCRCC != 0 )
        dput_error();
/* User information in negative case                       */
    KDCS_DPUTMI (&ma3,80,"*PRICOMP");
    if (KCRCC != 0 )
        dput_error();
/* acknowledgment job in negative case                     */
    KDCS_DPUTMT (&ma2,0,"*PRICOMP");
    if (KCRCC != 0 )
        dput_error();
/* End of complex                                         */
    KDCS_MCOMEC ("*PRICOMP");
    if (KCRCC != 0 )
        mcom_error();

```



## 9.2.2 Example of a complete C program unit

The following is an example of an executable KDCS program unit in the C programming language.

The program outputs the text "hello world !" and the names of the logical terminals.

```
#include <kcmac.h>

struct work {
    union kc_paa call_pb;
    char  buffer[400];
};

struct kc_ca {
    struct ca_hdr kopf;
    struct ca_rti rfld;
    char  kcprg[500];
};

#define NB      spab->buffer
#define KBKOPF kb->kopf
#define KBRFLD kb->rfld
#define PB      spab->call_pb

void mhello ( struct kc_ca *kb, struct work *spab )
{
    /* KDCS interface initialization */
    KDCS_SET( &PB, &KBKOPF, &KBRFLD );

    /* INIT - Call */
    KDCS_INIT( sizeof(struct kb->kcprg), sizeof(struct work) );

    /* MPUT-NT - Call */
    strcpy ( NB, "hello world !\n\n" );
    KDCS_MPUTNT( NB, (short)strlen(NB), KDCS_SPACES,
                KDCS_SPACES, KCNODF );

    /* MPUT-NT - Call */
    sprintf ( NB, "lterm = %.8s \n", KBKOPF.kclogter );
    KDCS_MPUTNT( NB, (short)strlen(NB), KDCS_SPACES,
                KDCS_SPACES, KCNODF );

    /* PEND-FI - Call */
    KDCS_PENDFI();
}
```

### 9.2.3 Example: INPUT exit

The *forinput* INPUT exit is called when data is input in the format mode, and it reacts to the input as follows:

User commands are initiated:

- KDCOUT: Press the F1 key
- KDCDISP: Press the F2 key
- KDCOFF: the first character of the input is "/"; this is only accepted outside of a service.

If the user is also allowed to input KDCLAST and KDCFOR, then the program must be expanded accordingly.

This INPUT exit is generated with the KDCDEF statement EXIT:

```
EXIT PROGRAM=FORINPUT,USAGE=(INPUT,FORMMODE)
```

```
#include <string.h>
#include <kcinp.h>

#define KDCDISP 2
#define KDCOUT 1
#define NOKEY 0
#define FKEY 1
#define KDCOFF "KDCOFF "
#define NOTAC (strncmp (param->kcicfinf, "ON", 2) != 0)
#define CV_END (strncmp (param->kcicvst, "EC", 2) == 0)

#define TAC          param->kcicvtac[0]
#define fkey        param->kcifkey
#define cmd         param->kcintac
#define nexttac     param->kcintac
#define errcode     param->kciercd
#define contcode    param->kciccd
#define firstchar   param->kcifch[0]
#define cut         param->kcicut

static int  key( struct kc_inp * );
static void func_control( struct kc_inp * );
static void cv_continue( struct kc_inp * );

void forinput ( struct kc_inp *param )
{
    if ( key ( param ) == NOKEY)                /* No F-key */
    { if ( CV_END)
      func_control ( param );
      else
      cv_continue ( param );
    }
}
```

```

/*****
/*      function key for checking F-key      */
*****/

int key ( struct kc_inp *param )
{
    int key_value = NOKEY;

    if (fkey > 0)
        { switch (fkey)
          { case KDCOUT:

              memcpy (cmd, "KDCOUT ", 8);
              memcpy (contcode, "CD", 2);
              cut = 'N';
              memset (errcode, ' ', 4);
              key_value = FKEY;
              break;

              default: break;
          }
        }

    return key_value;
}

/*****
/*      function func_control: checking the next FUNCTION out of conversation*/
*****/

void func_control ( struct kc_inp * param )
{
    if (firstchar == '/')
        { /* check the first character*/
          { /* of input: '/' = KDCOFF */
            memcpy (cmd, KDCOFF, 8);
            memcpy (contcode, "CD", 2);
            cut = 'N';
            memset (errcode, ' ', 4);
          }
        }
    else
        { /* check control field */
          { /* no input in control field*/
            if (NOTAC)
                { memset (nexttac, ' ', 8);
                  memcpy (contcode, "ER", 2);
                  cut = 'N';
                  memcpy (errcode, "ER01", 4);
                }
            else
                { switch (TAC)
                  { /* TAC for the next */
                    { /* conversation */
                      case '1':
                        memcpy (nexttac, "DTAC1 ", 8);
                        memcpy (contcode, "SC", 2);
                        cut = 'Y';
                        memset (errcode, ' ', 4);
                        break;
                    }
                }
            }
        }
    }
}

```

```

        case '2':
            memcpy (nexttac, "DTAC3  ", 8);
            memcpy (contcode, "SC", 2);
            cut = 'Y';
            memset (errcode, ' ', 4);
            break;

        case '3':
            memcpy (nexttac, "DTAC6  ", 8);
            memcpy (contcode, "SC", 2);
            cut = 'Y';
            memset (errcode, ' ', 4);
            break;

        default:
            /* TAC is invalid */
            memset (nexttac, ' ', 8);
            memcpy (contcode, "ER", 2);
            cut = 'N';
            memcpy (errcode, "ER02", 4);
            break;
    }
}

/*****
/*          function cv_continue: continue the conversation          */
*****/

void cv_continue ( struct kc_inp * param )
{
    memset (nexttac, ' ', 8);
    memcpy (contcode, "CC", 2);
    cut = 'N';
    memset (errcode, ' ', 4);
}

```

## 9.2.4 Example: MSGTAC event service

The MSGTAC event service NOHACK counts the number of failed attempts in a TLS. If openUTM accepts a KDCSIGN (i.e. UTM message K008 or K033), then this TLS is deleted.

If after three invalid KDCSIGN attempts the 4th KDCSIGN attempt also fails, then the corresponding terminal should be automatically disconnected using an FPUT call with KCRN="KDCPTRMA". The KDCS message area contains the following administration command, see also the openUTM manual "Generating Applications":

```
PTERM=pterm,PRO=prname,ACT=DIS
```

The K-messages are each read from the MSGTAC program unit using FGET. After "processing" a K-message, the next K-message is read immediately within the same program run using FGET.

```
#include <stdio.h>

#include <kcmac.h>
#include <kcmsg.h>

#define _K008 (memcmp (NR, "K008", 4) == 0)
#define _K033 (memcmp (NR, "K033", 4) == 0)

/* K008: KDCSIGN accepted */
/* K033: Start-Format */

#define MESSAGE_OK (_K008 || _K033)

#define _K004 (memcmp (NR, "K004", 4) == 0)
#define _K006 (memcmp (NR, "K006", 4) == 0)
#define _K031 (memcmp (NR, "K031", 4) == 0)

/* K004: Invalid Identification */
/* K006: Invalid password */
/* K031: Card not ok */

#define OTHER_MESSAGE !(_K004 || _K006 || _K008 || _K031 || _K033)
#define HACK_MAX 3

#define PTERM " PTERM="
#define PRONAM " ,PRONAM="
#define DIS " ,ACTION=DIS"
#define OFF " ,STATUS=OFF"
#define kcrc_ca->ca return.kcrgcc
#define pa spab->param
#define NR spab->ma.kcmsgs.msghdr.MSGNR
#define MSG spab->ma.kcmsgs.msg
#define LTERM spab->ma.lterm
#define hacknr spab->hack_nr
#define admin spab->ma.adm

struct adm_line
{ char pterm_t[6];
```

```

        char pterm[8];
        char pronam_t[8];
        char pronam[8];
        char dis_t[11];
        char off_t[11];
    };

    struct ca_area
    { struct ca_hdr ca_head;
      struct ca_rti ca_return;
    };

    struct work
    { struct kc_pa param;

      short hack_nr;
      struct msg_area
      { char lterm[8];
        struct adm_line adm;
        struct KCM$SGS kcmsg;
      } ma;
      char buffer[100];
    };

    static void set_lterm( struct work * );
    static void set_pterm( struct work * );

    void NOHACK (struct ca_area *ca, struct work *spab )
    {
        int other_message = 0;

        /* INIT-Operation */

        KDCS_SET (&spab->param, &ca->ca_head, &ca->ca_return);
        KDCS_INIT (0,512);

        /*****
        /* while-loop: reading and processing all messages */
        /*****

        while (KCRCC == 0 )
        {
        /* FGET-Operation: reading the message */

            KDCS_FGET (&spab->ma.kcmsg,132,KDCS_SPACES);
            if (KCRCC != 0 )
                break;

            if (OTHER_MESSAGE)
            { other_message = 1;
              break;
            }

            set_lterm ( spab );

        /* read TLSB */

```



```

KDCS_GTDA (&hacknr,2,"TLSB",LTERM);
if (KCRCC != 0 )
    break;

    if ((hacknr < 0) || (hacknr > HACK_MAX))
        hacknr = 0;                                /* Initialize TLS          */
/* If KDCSIGN is correct, initialize the TLS, if not, count the number    */
/* of failed attempts. After the fourth invalid KDCSIGN disconnect the    */
/* corresponding terminal.                                                */
/*
    if ((hacknr < HACK_MAX) && MESSAGE_OK)
        hacknr = 0;                                /* Initialize TLS          */
else
    /* invalid KDCSIGN                                                    */
    {
        if (hacknr < HACK_MAX)
            ++hacknr;
        else
            { memcpy (admin.pterm_t, PTERM, 7);
              memcpy (admin.pronam_t, PRONAM, 8);
                set_pterm ( spab );
            }
    }
/* Disconnect the terminal by asynchronous administration                */
    memcpy (admin.dis_t, DIS, 11);
    memcpy (admin.off_t, OFF, 11);

KDCS_FPUTNE (&admin,sizeof(struct adm_line),"KDCPTRMA",KDCS_SPACES,KCNODF)
;
if (KCRCC != 0 )
    break;

    hacknr = 0;

/* log on User logging                                                  */
KDCS_LPUT (&admin,sizeof(struct adm_line));
if (KCRCC != 0 )
    break;
}

/* set up TLSB                                                         */
KDCS_PTDA (&hacknr,2,"TLSB",LTERM);
}
/* *****
/*                               End of while loop                       */
/* *****
if ( KCRCC != 10 || other_message)

```

```

/* other message or error in the while loop */
{
/* error line */
    sprintf(spab->buffer, "Error in program unit - conversation %8.8s\"
        ", TAC: %8.8s because %4.4s. RC= %3.3s " ,
        ca->ca_head.kccv_tac , ca->ca_head.kcpr_tac ,
        pa.kcop , KDCS_ERR );

/* RSET-Operation */
    KDCS_RSET();

/* LPUT-Operation: log on user logging */
    KDCS_LPUT( spab->buffer , strlen( spab->buffer ) );
}

/* PEND FI-Operation */
KDCS_PENDFI();
}
/*****
/*          function set_lterm ( )          */
*****/

void set_lterm ( struct work * spab )
{
    if _K004
    { memcpy (LTERM,MSG.K004.LTRM, 8);
      return;
    }
    if _K006
    { memcpy (LTERM, MSG.K006.LTRM, 8);
      return;
    }
    if _K008
    { memcpy (LTERM, MSG.K008.LTRM, 8);
      return;
    }
    if _K031
    { memcpy (LTERM, MSG.K031.LTRM, 8);
      return;
    }
    if _K033
    { memcpy (LTERM, MSG.K033.LTRM, 8);
      return;
    }
}

/*****
/*          function set_pterm ( )          */
*****/

```

```
/******  
void set_pterm ( struct work *spab )  
{ if _K004  
  { memcpy (admin.pterm, MSG.K004.PTRM, 8);  
    memcpy (admin.pronam, MSG.K004.PRNM, 8);  
    return;  
  }  
  if _K006  
  { memcpy (admin.pterm, MSG.K006.PTRM, 8);  
    memcpy (admin.pronam, MSG.K006.PRNM, 8);  
    return;  
  }  
  if _K031  
  { memcpy (admin.pterm, MSG.K031.PTRM, 8);  
    memcpy (admin.pronam, MSG.K031.PRNM, 8);  
    return;  
  }  
}
```

The above example for the MSGTAC event service simply indicates appropriate ways of evaluating messages and administering the application.

However, the K094 message (SIGNON SILENT-ALARM) should be used to monitor security infringements since this also includes UPIC and OSI TP clients. Furthermore, wider-ranging administration of the UTM application is possible using the programmed administration capability (ADMI interface).

## 9.2.5 Example of a complete UTM application

This application example administers address data that has been placed in a file. The application provides the following functions which are called by entering the corresponding TAC in the field used for this purpose. The input and output is performed in one format.

TAC	Function	
1	Read	outputs an address that is in the file. The search term used is the last name and the first two letters of the first name, both of which are to be specified in the appropriate fields.
2	Write	adds a new address to the file. There may not already be an address with the same search key (see above) in the file.
3	Update	modifies an address entry. The address must already exist in the file.
4	Delete	deletes an existing address from the file.

An error message appears in the lowest line of the format if used incorrectly.

The numbers listed above are the transaction codes (TACs) that control the application. Transaction code 1 calls the "TPREAD" program unit, and transaction codes 2, 3 and 4 call the "TPUPDATE" program unit. These program units then each branch to the "TPFILE" program unit. This program unit is utilized as the START and SHUT exits and contains the subroutines that perform the input/output operations on the address file. The "BADTACS" program unit is automatically called by openUTM if an invalid TAC is specified.

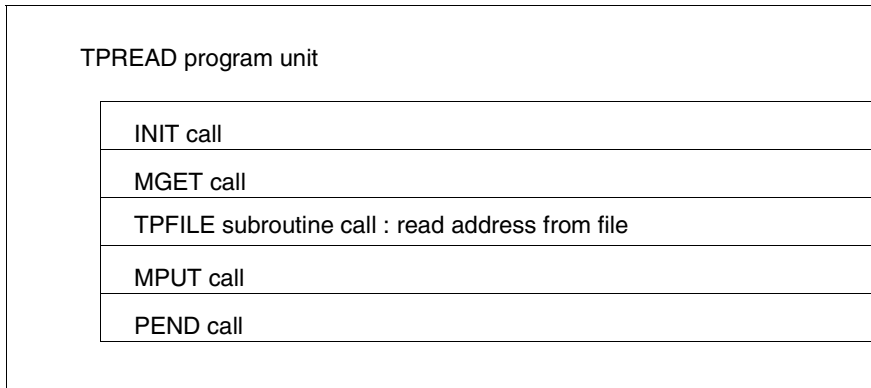
The "ERRCHECK" function handles errors that arise in the program units.

After the connection to the application has been established and a successful KDCSIGN, the format is immediately output by openUTM (start format). The interaction with the user is then done in a strictly controlled dialog, i.e. the application reacts to the input of the TAC and of the key by outputting the format containing the address searched for or by outputting a success or error message in the bottom line.

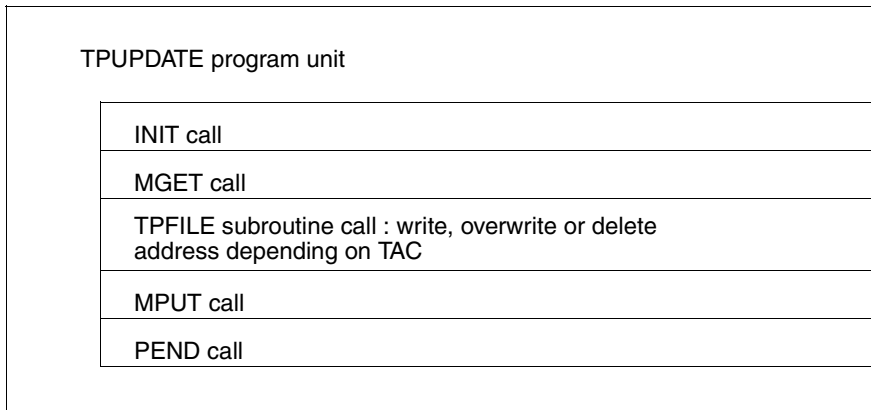


This program is only meant to demonstrate how you program using openUTM. The file accesses are not secured by the UTM transaction concept.

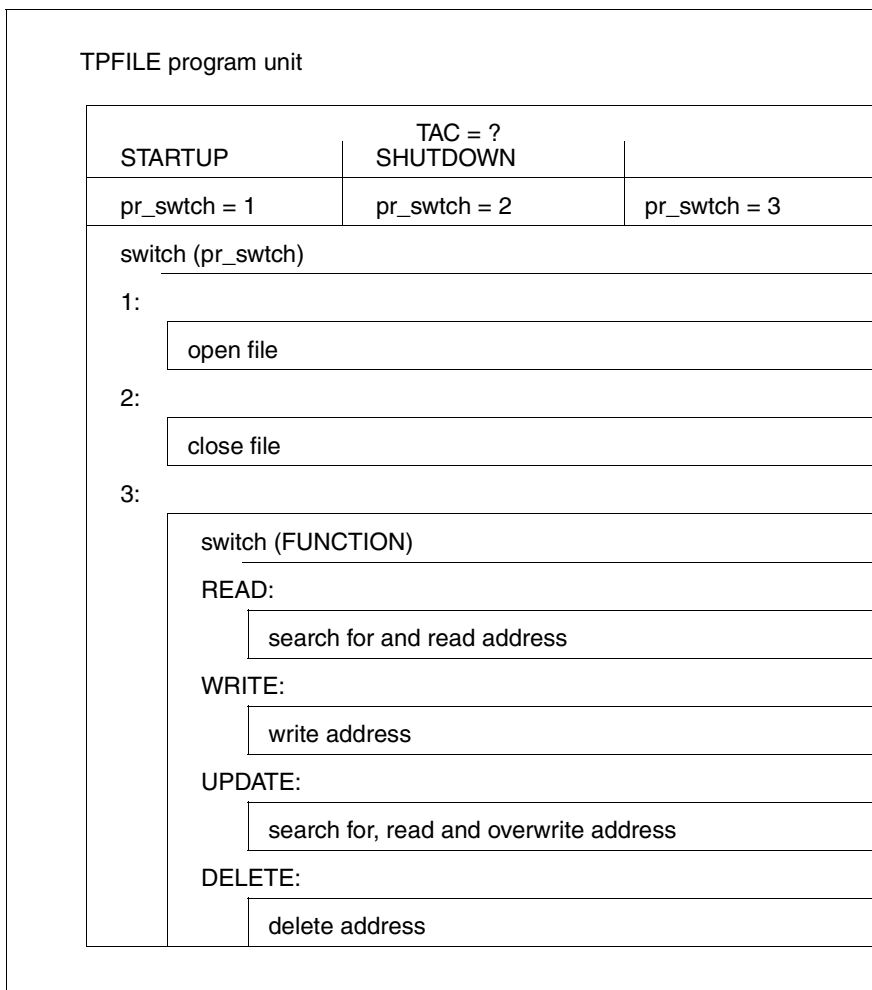
The following structure diagram presents the structure of the program units:



Structure diagram of the TPREAD program unit



Structure diagram of the TPUPDATE program unit



Structure diagram of the TPFILE program unit.

For the sake of completeness the generation of this application is listed at the end of the C program. Please consult the openUTM manual "Generating Applications" for the exact meaning of the individual operands and statements.

The following diagram shows the FORM1 format used for this application:

```

*****
                A d d r e s s   A d m i n i s t r a t i o n
*****
                Select function:

-----

Actual function:
Name:                               First Name:
Street:                             No.:
Postal Code:                         Residence:
Phone:

-----

                Function selection
1 = Show address                     |   4 = Delete address
2 = Enter address                    |
3 = Update address                   |   Finish with `kdcoeff`

-----

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

The data structure for the address assistant for this format is printed in the following.

```

typedef struct {
    char NAME [ 14 ] ;
    char FIRST_NAME [ 20 ] ;
    char STREET [ 26 ] ;
    char NUMBER [ 10 ] ;
    char POSTAL_CODE [ 5 ] ;
    char RESIDENCE [ 24 ] ;
    char PHONE [ 21 ] ;
} ADDRESS;

typedef struct {
    char TAC [ 8 ] ;
    char FUNCTION [ 27 ] ;
    ADDRESS addr;
    char MSGTEXT [ 80 ] ;
} FORM1 ;

```

The FUNCTION field is a protected output field with the "automatic input" attribute. Numerical data is allowed to be entered in the POSTAL\_CODE field and MSGTEXT is a protected output field. You can print out the entire list of attributes. You will find more information on this subject in your formatting system manual.

**FORM1.h header file**

```

/* FORMAT NAME      : FORM1    */
/* USER AREA LENGTH : 235     */

typedef struct {
    char NAME           [ 14 ] ;
    char FIRST_NAME    [ 20 ] ;
    char STREET         [ 26 ] ;
    char NUMBER         [ 10 ] ;
    char POSTAL_CODE    [ 5 ] ;
    char RESIDENCE      [ 24 ] ;
    char PHONE          [ 21 ] ;
} ADDRESS;

typedef struct {
    char TAC            [ 8 ] ;
    char FUNCTION       [ 27 ] ;
    ADDRESS addr;
    char MSGTEXT        [ 80 ] ;
} FORM1 ;

```

**tp.h header file**

```

#ifndef TP_H
#define TP_H

#include "FORM1.h"

#define kcrc ca->ca_return.kcrccc
#define pa spab->param

struct ca_area
{ struct ca_hdr ca_head;
  struct ca_rti ca_return;
};

struct work
{ union kc_paa param;
  FORM1 std_mask;
  char progname[8];
};

void BADTACS ( struct ca_area * , struct work * );
void errcheck ( struct ca_area * , struct work * );
void TPFILE ( struct ca_area * , struct work * );
void TPREAD ( struct ca_area * , struct work * );
void TPUPLICATE ( struct ca_area * , struct work * );

#endif

```



**TPREAD program unit**

```

#include <kcmac.h>
#include "tp.h"

void TPREAD (struct ca_area *ca , struct work *spab )
{
/* INIT-Operation */
KDCS_SET (&spab->param, &ca->ca_head, &ca->ca_return);
KDCS_INIT (0,sizeof(struct work));
if (KCRCC != 0 )
    { memcpy (spab->programe, "TPREAD ", 8);
      errcheck (ca, spab);
    }
else
    memcpy (spab->std_mask.TAC, ca->ca_head.kcpr_tac, 8);

/* MGET-Operation */
KDCS_MGET ( spab->std_mask.FUNCTION
            ,sizeof( FORM1 )
            ,"*FORM1 "
            );
if (KCRCC != 0 )
    { memcpy (spab->programe, "TPREAD", 8);
      errcheck (ca, spab);
    }
/* call function "tpfile" for reading address */
TPFILE (ca, spab);

/* MPUT-Operation */
KDCS_MPUTNT (&spab->std_mask,sizeof(FORM1)
            ,KDCS_SPACES,"*FORM1 ",KCNODEF);
if (KCRCC != 0 )
    { memcpy (spab->programe, "TPREAD", 8);
      errcheck (ca, spab);
    }

/* PEND FI-Operation */
KDCS_PENDFI();
}

```

**TPUPDATE program unit**

```

#include <kcmac.h>
#include "tp.h"

void TPUPDATE ( struct ca_area *ca , struct work *spab )
{
/* INIT-Operation */

KDCS_SET (&spab->param, &ca->ca_head, &ca->ca_return);
KDCS_INIT (0,sizeof(struct work));
if (KCRCC != 0 )
    { memcpy (spab->progrname, "TPUPDATE", 8);
      errcheck (ca, spab);
    }
else
    memcpy (spab->std_mask.TAC, ca->ca_head.kcpr_tac, 8);

/* MGET-Operation */

KDCS_MGET ( spab->std_mask.FUNCTION
            ,sizeof( FORM1 )
            ,"*FORM1 "
            );
if (KCRCC != 0 )
    { memcpy (spab->progrname, "TPUPDATE", 8);
      errcheck (ca, spab);
    }
/* call function "tpfile" for updating address */

TPFILE (ca, spab);

/* MPUT-Operation */

KDCS_MPUTNT (&spab->std_mask,
            sizeof(FORM1),KDCS_SPACES,"*FORM1 " ,KNODF);
if (KCRCC != 0 )
    { memcpy (spab->progrname, "TPUPDATE", 8);
      errcheck (ca, spab);
    }

/* PEND FI-Operation */

KDCS_PENDFI();
}

```

**BADTACS program unit**

```

#include <kcmac.h>
#include "tp.h"

#define ERRLINE "***** Wrong TAC - Please repeat \
Input *****"

void BADTACS (struct ca_area *ca, struct work *spab )
{
/* INIT-Operation */

KDCS_SET (&spab->param, &ca->ca_head, &ca->ca_return);
memset (&spab->std_mask.TAC, ' ', 8);
KDCS_INIT (0, sizeof(struct work));
if (KCRCC != 0 )
    { memcpy (spab->progrname, "BADTACS", 8);
      errcheck (ca, spab);
    }

/* MGET-Operation */

KDCS_MGET ( spab->std_mask.FUNCTION
            ,sizeof( FORM1 )
            ,ca->ca_return.kcrfn
            );
if (KCRCC != 0 )
    { memcpy (spab->progrname, "BADTACS", 8);
      errcheck (ca, spab);
    }

/* MPUT-Operation: Replace the standard error message */

memcpy (spab->std_mask.MSGTEXT, ERRLINE, 80);
memset (spab->std_mask.TAC, ' ', 8);
KDCS_MPUTNT ( &spab->std_mask
              ,sizeof( FORM1 )
              ,KDCS_SPACES
              ,"*FORM1 "
              ,KCNOF
              );
if (KCRCC != 0 )
    { memcpy (spab->progrname, "BADTACS", 8);
      errcheck (ca, spab);
    }

/* PEND FI call */

KDCS_PENDFI();
}

```

**ERRCHECK function**

```

#include <kcmac.h>
#include "tp.h"

void errcheck ( struct ca_area * ca , struct work * spab )
{
    struct err_line
    { char ftext[35];
      char progame[8];
      char optext[10];
      char op_code[4];
      char cctext[8];
      char cc[3];
      char cdtext[8];
      char cd[4];
    } err_msg;

    /* ----- Making connections for the KDCS_... macros ----- */
    KDCS_SET (&spab->param, &ca->ca_head, &ca->ca_return);

    /* making entries in the errorline */
    memcpy (err_msg.ftext, "***** E R R O R in program unit ",35);
    memcpy (err_msg.progame, spab->progame, 8);
    memcpy (err_msg.optext, " Op-Code: ",10);
    memcpy (err_msg.op_code, pa.kcop, 4);
    memcpy (err_msg.cctext, " kcrccc=", 8);
    memcpy (err_msg.cc, KDCS_ERR , 3);
    memcpy (err_msg.cdtext, " kcrcdc=", 8);
    memcpy (err_msg.cd, KDCS_RTI->kcrcdc, 4);

    memset (&spab->std_mask, '\0', sizeof (FORM1));
    memcpy (spab->std_mask.MSGTEXT, &err_msg, 80);

    /* MPUT-Operation */
    KDCS_MPUTNE (&spab->std_mask,sizeof(FORM1),KDCS_SPACES,"*FORM1 ",KCNOF);

    /* PEND ER-Operation */
    KDCS_PENDER();
}

```

**TPFILE program unit with START/SHUT exits and file accesses**

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <kcpa.h>
#include <kcca.h>

#include "tp.h"

#define M spab->std_mask
#define TAC ca->ca_head.kccv_tac
#define JOB TAC[0]
#define ADR_LENGTH ( (long) sizeof( ADDRESS ) )

#define READ      '1'
#define WRITE    '2'
#define UPDATE   '3'
#define DELETE   '4'
#ifdef __SNI_HOST_BS2000
#define FILE_NAME "link=CAPPLI"
#define FILE_MODE "r+b,type=record,forg=key"
#define FILE_MODE_FIRST "w+b,type=record,forg=key"
#else
#define FILE_NAME "cappli.address"
#define FILE_MODE "r+b"
#define FILE_MODE_FIRST "w+b"
#endif
#define PREVIOUS_POSITION  -ADR_LENGTH
#define SAG_NAME            "FUJITSU TECHNOLOGY SOLUTIONS"
#define SAG_STREET         "Musterstrasse"
#define SAG_NUMBER         "6"
#define SAG_PCODE         "12345"
#define SAG_RES            "Musterstadt"
#define SAG_PHONE          "+12 34 567-89"

typedef enum {
    FOUND      = 1
    ,NOT_FOUND = 2
} address_status;

static address_status addr_fetch(struct work * );

static ADDRESS  address;
static FILE    * filepointer;
static fpos_t   FilePosition;

void TPFIL ( struct ca_area * ca , struct work *spab )
{
    int pr_swch;
#ifdef __SNI_HOST_BS2000
    char BS2Cmd[500];
#endif

    if (strncmp (TAC,"STARTUP ", 8) == 0)
        pr_swch = 1;
    else

```

```

    { if (strncmp (TAC,"SHUTDOWN", 8) == 0)
      pr_swch = 2;
      else
      { pr_swch = 3;
        memset (M.MSGTEXT, '*', 80);
      }
    }

switch (pr_swch)

{ case 1:

#ifdef __SNI_HOST_BS2000
    sprintf( BS2Cmd ,
             "SET-FILE-LINK LINK-NAME = CAPPLI ,FILE-NAME = CAPPLI.ADDRESS\"
             " ,SUPPORT      = *DISK( SHARED-UPDATE = *YES)      " ,
             ADR_LENGTH );
    system( BS2Cmd );
#endif

    if ((filepointer = fopen (FILE_NAME, FILE_MODE)) == NULL) {
        if((filepointer = fopen( FILE_NAME , FILE_MODE_FIRST )) == NULL ) {
            perror("fopen:");
            exit(-1);
        }

        memcpy( address.NAME , SAG_NAME , sizeof( SAG_NAME )-1);
        memcpy( address.STREET, SAG_STREET , sizeof( SAG_STREET )-1);
        memcpy( address.NUMBER, SAG_NUMBER , sizeof( SAG_NUMBER )-1);
        memcpy( address.POSTAL_CODE , SAG_PCODE , sizeof( SAG_PCODE )-1);
        memcpy( address.RESIDENCE , SAG_RES , sizeof( SAG_RES )-1);
        memcpy( address.PHONE , SAG_PHONE , sizeof( SAG_PHONE )-1);

        fwrite(&address, ADR_LENGTH, 1, filepointer);
        fclose( filepointer );

        if ((filepointer = fopen (FILE_NAME, FILE_MODE)) == NULL) {
            perror("fopen:");
            exit(-1);
        }
    }
    break;

case 2:
    fclose (filepointer);
    break;

case 3:

    switch (JOB)

    { case READ:

        memcpy (M.FUNCTION, "Show address *****", 26);
        if (addr_fetch( spab ) == NOT_FOUND)
            { memcpy (M.MSGTEXT, "Address not found ", 22);
              break;
            }
        else

```

```

        { memcpy (M.addr.NAME, address.NAME, ADR_LENGTH );
          break;
        }

    case WRITE:

        memcpy (M.FUNCTION, "Enter address          ***", 26);
        if (addr_fetch( spab ) == FOUND)
            { memcpy (M.MSGTEXT, "Address already exists  ", 23);
              break;
            }
        else
            { memcpy (address.NAME, M.addr.NAME, ADR_LENGTH );
              fseek (filepointer, 0, 2);
              fwrite (&address, ADR_LENGTH, 1, filepointer);
              memcpy (M.MSGTEXT, "Address entered          ", 24);
              break;
            }
    case UPDATE:

        memcpy (M.FUNCTION, "Update address          *****", 26);
        if (addr_fetch( spab ) == NOT_FOUND)
            { memcpy (M.MSGTEXT, "Address not found      ", 22);
              break;
            }
        else
            { memcpy (address.NAME, M.addr.NAME, ADR_LENGTH );
              fsetpos(filepointer,&FilePosition );
              fwrite (&address, ADR_LENGTH, 1, filepointer);
              memcpy (M.MSGTEXT, "Address changed  ", 16);
              break;
            }

    case DELETE:

        memcpy (M.FUNCTION, "Delete address          *****", 26);
        if (addr_fetch( spab ) == NOT_FOUND)
            { memcpy (M.MSGTEXT, "Address not found      ", 22);
              break;
            }
        else
            {
#ifdef __SNI_HOST_BS2000
                memset (&address, '*', ADR_LENGTH);
                fsetpos(filepointer,&FilePosition );
                fwrite (&address, ADR_LENGTH, 1, filepointer);
#else
                fdelrec(filepointer , NULL );
                memcpy (M.MSGTEXT, "Address deleted          ", 22);
                break;
#endif
            }
        }
    }
    return;
}

```

```
/*
*****
Function addr_fetch
*****
*/

static address_status addr_fetch( struct work *spab )
{
    address_status filestatus = NOT_FOUND;

    memset (&address, 0X00, ADR_LENGTH);
    fseek (filepointer, 0, 0);

    while ( fgetpos( filepointer , &FilePosition ) ,
            (fread (&address, ADR_LENGTH, 1, filepointer)) != NULL)
    {
        if (address.NAME[0] != '*' )           /* Address not deleted */
        {
            if (strncmp (address.NAME, M.addr.NAME, sizeof M.addr.NAME) == 0)
                if (strncmp (address.FIRST_NAME
                            , M.addr.FIRST_NAME
                            , sizeof M.addr.FIRST_NAME) == 0)
                {
                    filestatus = FOUND;
                    break;
                }
        }
        memset (&address, 0X00, ADR_LENGTH);
    }

    return filestatus;
}
}
```



**KDCDEF statements**

```

REM *****
REM ***          D E F  -  S T A T E M E N T S          ***
REM ***
REM ***          KDCFILE = CAPPLI                      ***
REM *****
*
OPTION GEN=ALL
*
ROOT CAPPLI
*
*+-----+
*| MAX statements                                     |
*+-----+
*
MAX KDCFILE      = kdcfile
MAX APPLINAME    = CAPPLI
MAX APPLIMODE    = S
MAX TASKS        = 3
MAX ASYNTASKS    = 1
MAX PGPOOL       = (25)
MAX CACHESIZE    = (100,30)
MAX TRACEREC     = 10000
MAX RECBUF       = (5,4096)
MAX LPUTBUF      = 10
MAX LPUTLTH      = 4096
MAX TERMWAIT     = 60
MAX KB           = 1024
MAX NB           = 2048
MAX SPAB         = 4096
MAX CLRCH        = X'FF'
*
*+-----+
*| FORMSYS statement                                 |
*+-----+
*
FORMSYS ...
*
*+-----+
*| MESSAGE statement                                 |
*+-----+
*
MESSAGE MODULE=MSG
*
*+-----+
*| PROGRAM statements                                |
*+-----+
*
*
*
OPTION DATA = { PROGRAM-STATIC
                PROGRAM-SHARED-OBJ
                PROGRAM-BLS
                PROGRAM-OLD-DLL
                }
*

```

```

*+-----+
*| EXIT statements |
*+-----+
*
EXIT PROGRAM=TPFILE ,USAGE=START
EXIT PROGRAM=TPFILE ,USAGE=SHUT
*
*+-----+
*| TAC statements |
*+-----+
*
REM ***      ADMINISTRATION DIALOG      ***
TAC KDC TAC ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC LTERM ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC PTERM ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC SWTCH ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC SEND ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC APPL ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC USER ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC DIAG ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC LOG ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC INF ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC HELP ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC SHUT ,PROGRAM=KDCADM , ADMIN=Y
TAC KDC TCL ,PROGRAM=KDCADM , ADMIN=Y
*
REM ***      ADMINISTRATION ASYNCHRON      ***
TAC KDC TACA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC LTRMA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC PTRMA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC SWCHA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC USERA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC SENDA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC APPLA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC DIAGA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC LOGA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC INF A ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC HELPA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC SHUTA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
TAC KDC TCLA ,PROGRAM=KDCADM ,TYPE=A , ADMIN=Y
*
REM ***      Application specific TACs      ***
*
TAC KDC MSGTC , PROGRAM=NOHACK
TAC KDC BADTC , PROGRAM=BADTACS
TAC 1 , PROGRAM=TPREAD , LOCK = 1
TAC 2 , PROGRAM=TPUPDATE , TACCLASS=1 , LOCK = 2
TAC 3 , PROGRAM=TPUPDATE , TACCLASS=1 , LOCK = 2
TAC 4 , PROGRAM=TPUPDATE , TACCLASS=1 , LOCK = 2
*
*+-----+
*| TACCLASS statements |
*+-----+
*
TACCLASS 1,TASKS=1
*

```

```

*+-----+
*| USER statements |
*+-----+
*
USER NINA ,PASS=C 'SOLO' ,FORMAT=*FORM1 ,KSET=BUNDLE1 (1)
USER URSUS ,PASS=C 'SOLO' ,FORMAT=*FORM1 ,KSET=BUNDLE2 (1)
USER ADMIN ,PASS=C 'ADM' ,KSET=MASTER,PERMIT=ADMIN
*
*+-----+
*| TLS statements |
*+-----+
*
TLS TLSB
*
*+-----+
*| TPOOL statements |
*+-----+
*
TPOOL LTERM=... , NUMBER=2 , PRONAM=*ANY, PTYPE=... , KSET = MASTER
TPOOL LTERM=... , NUMBER=2 , PRONAM=*ANY, PTYPE=... , KSET = MASTER
*
*+-----+
*| PTERM / LTERM statements |
*+-----+
*
OPTION DATA= { TERM-BS2
*              TERM-Unix
*              TERM-WIN }
*
*+-----+
*| KSET statements |
*+-----+
*
KSET BUNDLE1 , KEYS=(1,2)
KSET BUNDLE2 , KEYS=(1)
KSET MASTER, KEYS=MASTER

```

## B (1) BS2000 format

## Input files for the generation procedure

*PROGRAM statements:*

- **PROGRAM-STATIC** (for static linking)

```
PROGRAM KDCADM ,COMP=C
PROGRAM TPREAD ,COMP=C
PROGRAM TPUUPDATE ,COMP=C
PROGRAM TPFILE ,COMP=C
PROGRAM NOHACK ,COMP=C
PROGRAM BADTACS ,COMP=C
```

- B** ● **PROGRAM-BLS** (for dynamic loading with BLS)

```
B * -----+
B DEFAULT PROGRAM COMP = C
B PROGRAM KDCADM
B * -----+
B MPOOL LCPOOL , SIZE = 10 -
B , SCOPE = GROUP -
B , ACCESS = READ
B * -----+
B LOAD-MODULE LLMTPS , VERSION = 001 -
B , LIB = DYNAMIC-LOADED-LIB -
B , LOAD-MODE = (POOL,LCPOOL,STARTUP)
B * -----+
B DEFAULT PROGRAM COMP = C , LOAD-MODULE = LLMTPS
B PROGRAM BADTACS
B PROGRAM TPUUPDATE
B PROGRAM TPREAD
B PROGRAM TPFILE
B PROGRAM NOHACK
```

- X** ● **PROGRAM-SHARED-OBJ** (for shared object)

```
X PROGRAM KDCADM ,COMP=C
X * -----+
X SHARED-OBJECT tp, VERSION=001 -
X , DIRECTORY=/home/utmuser -
X , LOAD-MODE=STARTUP
X * -----+
X PROGRAM TPREAD ,COMP = C, SHARED_OBJECT = tp
X PROGRAM TPUUPDATE ,COMP = C, SHARED_OBJECT = tp
X PROGRAM TPFILE ,COMP = C, SHARED_OBJECT = tp
X PROGRAM NOHACK ,COMP = C, SHARED_OBJECT = tp
X PROGRAM BADTACS ,COMP = C, SHARED_OBJECT = tp
```

*PTERM/LTERM statement:***B** ● **TERM-BS2**

```

B      PTERM TERM05 ,PTYPE=T9750 ,LTERM=DST01 ,PRONAM=D018KR06
B      LTERM DST01
B      PTERM TERM10 ,PTYPE=T9750 ,LTERM=DST02 ,PRONAM=D018KR06
B      LTERM DST02
B      PTERM TERM11 ,PTYPE=T9763 ,LTERM=DST03 ,PRONAM=D018KR06
B      LTERM DST03
B      PTERM TERM12 ,PTYPE=T9763 ,LTERM=DSTADMIN ,PRONAM=D018KR06
B      LTERM DSTADMIN
B      PTERM D17 ,PTYPE=T9001 ,LTERM=PRINTER1 ,PRONAM=D018KR06
B      LTERM PRINTER1 , USAGE = 0

```

**X** ● **TERM-Unix**

```

X      PTERM tty05,PTYPE=TTY,LTERM=DST01
X      LTERM DST01
X      PTERM tty10,PTYPE=TTY,LTERM=DST02
X      LTERM DST02
X      PTERM tty11,PTYPE=TTY,LTERM=DST03
X      LTERM DST03
X      PTERM tty12,PTYPE=TTY,LTERM=DSTADMIN
X      LTERM DSTADMIN
X      PTERM D17,PTYPE=PRINTER,LTERM=DRUCKER
X      LTERM DRUCKER, USAGE=0

```

**W** ● **TERM-WIN**

```

W      PTERM TT400,PTYPE=TTY,LTERM=DST01
W      LTERM DST01
W

```

**W** *Note:*

**W** A Windows system user can only use this LTERM connection point if he or she has  
**W** specified the `-PTT400` option when signing on (at the start of the dialog terminal process).



---

## 10 Additional information for COBOL

In addition to the general information in chapters 1 to 8, this chapter provides you with programming language-specific information which you will need in order to write COBOL program units:

The first section deals with how COBOL program units are structured. The second contains sample programs. The third lists the data structures KCBC and KCPAC.

### 10.1 Structure of COBOL program units

This section tells you:

- how to write a UTM COBOL program unit as a subroutine
- what you need to know when developing a LINKAGE SECTION and the WORKING-STORAGE SECTION
- how the PROCEDURE DIVISION has to look and how a KDCS call needs to be programmed in COBOL
- which platform-specific features you need to be aware of (dependencies on specific compilers, formatting systems etc.).

#### 10.1.1 COBOL program units as subroutines

UTM program units and event exits are subroutines of the UTM main routine. This fact leads to the following consequences:

- The program name defines the start address.
- At least one data structure must be defined in the LINKAGE SECTION.
- The program unit is terminated dynamically with the PEND call. Event exits that are exited with the statement EXIT PROGRAM are the exception. The statement STOP RUN is not permitted (exceptions: the START and SHUT event exits).

A series of COPY members are available to ensure compatibility and to enable you to work with error-free data structures. The section “Data structures for COBOL program units” on [page 549](#) describes how to use these COPY elements.

### PROGRAM-ID as a start name

You define the start name for the program unit in the PROGRAM-ID paragraph. This name is freely definable, but must be unique within a given application program. There must be no naming conflicts between the program name and the runtime systems, the database systems, the formatting system, the communication components and openUTM.

When choosing names, it is therefore important to bear in mind the following points:

- B** ● For BS2000/OSD:
  - B** – All names that begin with KDC, KC or I are reserved.
- X/W** ● For Unix systems and Windows systems:
  - X/W** – All names that begin with KDC, KC, x or ITS are reserved.
  - X/W** – Names that begin with t\_ are reserved for CMX and PCMX.
  - X/W** – Names that begin with a\_, o\_ or s\_ are reserved for OSS.
- The names you define must comply with COBOL conventions.

You must also specify the program names (start names) when generating the UTM application - each of them must be named in the KDCDEF application PROGRAM (see the openUTM manual “Generating Applications”).



Please note what the COBOL compiler manual has to say about how program names are to be handled in the IDENTIFICATION DIVISION and in the CALL call.

### WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION is used primarily for constant data.

To ensure that program units remain compatibility and to make them easier to read, a series of constants with predefined KDCS names is provided in the form of COPY members.

It is a good idea only to store fields with fixed values in the WORKING-STORAGE SECTION. If you also want to store areas which contain variable data in the WORKING-STORAGE SECTION, you can also define the KDCS parameter area and the message area. However, since it is useful to accommodate them in the SPAB in the interests of saving storage space, these areas are described in the following section.



## LINKAGE SECTION

You can use the LINKAGE SECTION for passing parameters and as a working area.

In the LINKAGE SECTION, each program unit must have a data structure with the level number 01 which describes the KDCS communication area.

This can be followed by a further data structure with the level number 01. This data structure describes the standard primary working area (SPAB). You can accommodate the KDCS parameter area and the message areas in the SPAB.

The data structures in the KB and in the KDCS parameter area are available as COPY members (KCKBC and KCPAC).

You will need to define the message areas yourself. However, specific data structures are provided in COPY members for calls which request information from openUTM (e.g. INFO, INIT PU). If you are working with a formatting system, you will be able to use automatically generated addressing aids to structure the message area (see the formatting system manual).

### *Example*

```
LINKAGE SECTION.
  COPY KCKBC.                1)
  05  KB-ANY                  PIC X(22).  2)
  05  KB-STARTLOC            PIC X(2).    2)
  05  KB-DESTLOC            PIC X(2).    2)
  05  KB-FLGTAG              PIC X(5).    2)
  05  KB-FLGN01              PIC X(5).    2)
  05  KB-FLGN02              PIC X(5).    2)
  COPY KCPAC.                3)
  03  NB.                    4)
  COPY IFORMA3.              4)
```

- 1) KDCS communication area
- 2) User-specific declaration of the KB program area
- 3) SPAB with KDCS parameter area
- 4) Message area: the COPY statement fetches the input addressing aid for the format "FORMA3".

## Extending the LINKAGE SECTION

In addition to the communication area and the SPAB, you can also accommodate still other areas in the LINKAGE SECTION which can then be used as common data areas within a UTM application.

You can declare these areas with the KDCDEF statement AREA. For more information, refer to the openUTM manual "Generating Applications".

In COBOL program units you can use AREAs as follows:

- In the LINKAGE SECTION you define these areas with the level number 01.
- In the PROCEDURE DIVISION you specify these areas under USING.

The order in which these areas are defined with the AREA statement is also important. If the area defined in  $n$ th position is required, you will need to specify all areas up until this one both in the LINKAGE SECTION and in the PROCEDURE DIVISION in the case of USING.

This function is not reflected in DIN standard 66 265.

### *Example 1*

The areas AREA1, AREA2 and AREA3 were defined in this order using the AREA statement. AREA3 is required in a program unit. All areas are defined with the length 2000.

```
      .  
      .  
LINKAGE SECTION.  
      COPY KCKBC.  
      .  
      .  
      COPY KCPAC.  
      .  
      .  
01 AREA1  PIC X(2000).  
01 AREA2  PIC X(2000).  
01 AREA3  PIC X(2000).  
      .  
      .  
PROCEDURE DIVISION USING KCKBC, KCSPAB, AREA1, AREA2, AREA3.  
      .  
      .
```

X *Example 2. Program unit in COBOL (in Unix systems)*

X In the following, two areas are generated, defined in a C source (see [page 489](#)) and passed  
X to a program unit. The following are defined:

- X – the area *area* for direct access (i.e. the data area is passed to the program unit directly)
- X – the area *areaind* for indirect access

```
X IDENTIFICATION DIVISION.  
X PROGRAM-ID. COBAREA.  
X ENVIRONMENT DIVISION.  
X DATA DIVISION.  
X WORKING-STORAGE SECTION.  
X LINKAGE SECTION.  
X COPY KCKBC.  
X 05 PROG PIC X.  
X COPY KCPAC.  
X 03 NB PIC X(4000).  
X 01 AREA1 PIC X(20).  
X 01 AREA2 PIC X(30).  
X PROCEDURE DIVISION USING KCKBC KCSPAB AREA1 AREA2.  
X MOVE AREA1 TO BUFFER.  
X MOVE AREA2 TO BUFFER1.  
X PERFORM INIT-OP.  
X .  
X .  
X .
```

## Alternatives to AREAs

If program units which use AREAs are to be copied from one application to another, problems may arise when using AREAs owing to possible differences in the parameter lists. For this reason, AREAs in the local part of an application program should be replaced by data declarations with the EXTERNAL clause. In this case, you do not have to program the AREA declaration in KDCDEF or the AREA data declaration in the LINKAGE SECTION and in the PROCEDURE DIVISION; a data declaration is required in the WORKING-STORAGE SECTION with an EXTERNAL clause.

### Example

Rather than define:

```
LINKAGE SECTION.
.
.
01 AREA1.
    02 DATA-ID PIC X(8).
    02 DATA-EX PIC X(4000).
```

it would be better to define:

```
WORKING-STORAGE SECTION.
01 COMMON1 IS EXTERNAL.
    02 DATA-ID PIC X(8).
    02 DATA-EX PIC X(4000).
```

In this example, the COMMON area COMMON1 is defined in such a way that can be loaded as shareable. It can be defined as follows:

**B** For BS2000, for example, in assembly language:

```
B COMMON1 CSECT PUBLIC
B COMMON1 RMODE ANY
B COMMON1 AMODE ANY
B *
B DATA_ID DC C'DATA-ID1'
B DATA_EX DS CL4000
B END
```

**B** If the COMMON area is to be loaded locally in the process, then you do not need to specify the PUBLIC attribute.

**X/W** For Unix systems and Windows systems:

```
X/W struct COMMON1 {
X/W char DATA_ID [8] = "DATA-ID1";
X/W char DATA_EX [4000];
X/W }
```

## 10.1.2 Data structures for COBOL program units

To help you structure your data areas, openUTM is supplied with the following COPY members containing predefined data structures:

B  
X/W

The data structures are present in the library SYSLIB.UTM.061.COB.

The data structures are present in the directory copy-cobol85

B  
B  
B  
B

Name	Contents and meaning
KCAPROC	Optional second parameter area for the APRO call: This area allows you to select OSI TP function combinations and the security type.
KCATC	KDCS attribute functions: Where +formats are used, you can use the symbolic names for attribute functions to modify the attribute fields for formats.
KCCFC	Defines the second parameter passed by openUTM for the event exit INPUT. In this parameter openUTM passes the contents of the control fields in screen formats to the program unit. For this reason, this second parameter is also referred to as the control fields area.
KCDADC	Data structure for the DADM call: You should place this data structure over the message area for the KDCS call DADM RQ.
KCDFC	KDCS screen functions: You can use this symbolic name to influence the screen output by entering the name of the function you want in the KCDF field in the KDCS parameter area.
KCINFC	Data structure for the INFO call: You should place this data structure over the message area for the KDCS call INFO DT/SI/PC.
KCINIC	Defines a second parameter area for the INIT call (only necessary for INIT PU). UTM returns the information requested by the INIT PU call in this parameter area.
KCINPC	Data structure for the INPUT exit: This data structure contains the input and output parameters for the INPUT exit.
KCKBC	Data structure for the KDCS communication area. It contains: <ul style="list-style-type: none"> <li>– current service and program data</li> <li>– data returned to UTM after a call</li> <li>– (if required) the KB program area for passing data between programs within a service. You will also need to define the fields in the KB program area.</li> </ul>
KCMMSGC	Data structure for the UTM messages: You will need this data structure if you have to handle UTM messages in a MSGTAC routine or if you want to use a program you have written to analyze the SYSLOG file.
KCOPC	KDCS operation codes: This data structure contains symbolic names for the KDCS operations. For your KDCS calls you can enter a name in the KCOP field in the KDCS parameter area. Please note that the symbolic name for the SIGN call is SGN.

Name	Contents and meaning
KCPAC	Data structure for the KDCS parameter area: This area accepts the parameters for a KDCS call.
KCPADC	Data structure for the PADM call: You should place this data structure over the message area for the KDCS call PADM AI/PI.
KCSGSTC	Data structure for the SIGN call: You should place this data structure over the message area for the KDCS call SIGN ST with KCLA > 0.

The data structures KCOPC, KCATC and KCDFC define constants. You should therefore copy these area to the WORKING-STORAGE SECTION.

Copy the remaining data structures to the LINKAGE SECTION.

The data structures will be copied to the program unit as illustrated in the example below.

### *Example*

```

DATA DIVISION.
*****
WORKING-STORAGE SECTION.
    COPY KCOPC.
    COPY KCATC.
    COPY KCDFC.
*****
LINKAGE SECTION.
    COPY KCKBC.
       05 KBPRG          PIC X(80).
    COPY KCPAC.
    COPY KCINFC.
       05 FILLER        PIC X(50).
       03 NB REDEFINES KCINFC.
*****
PROCEDURE DIVISION USING KCKBC, KCSPAB.
.
.
.

```

## Command section in a COBOL program unit

The command section of a COBOL program unit is freely definable. There are merely a few transaction processing rules, as described in chapter 2, which you need to note:

- program units are subroutines of the UTM main routine KDCROOT
- program units have to be reentrant
- dialog program units must strictly adhere to the rules governing dialogs.

Event exits are subject to special rules which are described on [page 553](#).

## Passing addresses

The PROCEDURE DIVISION in a COBOL program unit begins with the following statement:

```
PROCEDURE DIVISION USING kckbc[, spab[, param1[, ... paramn]]]
```

**kckbc** is the name of the KDCS communication area which must be defined with the level number 01 in the LINKAGE SECTION. Where the COPY member KCKBC is used, the name is KCKBC.

**spab** is the name of the standard primary working area defined with the level number 01 in the LINKAGE SECTION. Where the COPY member KCPAC is used, the name is KCSPAB. If an area from the WORKING-STORAGE SECTION was used in place of the SPAB, this specification is omitted.

**param<sub>1</sub> ... param<sub>n</sub>** are the names of further objects defined in the LINKAGE SECTION; see [“Extending the LINKAGE SECTION” on page 546](#). These objects can, for example, be AREAs which serve as an extension of the SPAB. If no such objects are used, this specification is omitted.

### 10.1.3 KDCS calls in COBOL program units

Before you call a UTM function in a program, the KDCS parameter area must already have been supplied with all the necessary parameters.

These parameters include:

- the operation code for the call
- additional parameters determined by the operation code (see [chapter “KDCS calls” on page 203](#)).

In some KDCS calls, unused parameter fields must be supplied with `LOW-VALUE`. To avoid errors, you should always issue the command `MOVE LOW-VALUE TO KCPAC` before supplying values to the parameter fields.

#### Format of the KDCS call

Once all the necessary data areas have been supplied, the KDCS call can be issued. The start address for all operations is "KDCS".

The format of the `CALL` is as follows:

```
CALL "KDCS" USING parm1[, parm2].
```

`parm1` is the data name of the KDCS parameter area. If the corresponding `COPY` member is used, the name is "KCPAC". This name must always be specified.

`parm2` is the data name for the storage area in the program to which messages or data may need to be written or in which messages or data have been made available. In the present description, this area is generally referred to as "NB" (from the German acronym for message area). You can, however, choose any name you wish.

The data names can be labeled if necessary.

The extended format would then be:

```
parm1 [{IN|OF} dataname1}...], [parm2 [{IN|OF} dataname2}...]].
```

For more details, refer to the description of the COBOL compiler.



*Example*

A data structure which exists more than once as a substructure is to be used as a message area.

```
.
.
03 BOOK5.
05 DATX          PIC X(50).
.
.
.
03 BOOK8.
05 DATX          PIC X(50).
.
.
.
CALL "KDCS" USING KCPAC, DATX IN BOOK5.
```

**Event exits**

The event exits INPUT, START, SHUT and VORGANG must not contain KDCS calls. They must be written as subroutines and must be terminated with the statement EXIT PROGRAM.

In the case of START, SHUT and VORGANG, the addresses for the communication area (KB) and the standard primary working area (SPAB) are passed in the PROCEDURE DIVISION. Accordingly, the structures of these areas are defined in the LINKAGE SECTION (as for the program units with KDCS calls). On [page 563ff](#) is given an example of a combined START/SHUT exit.

In the case of the INPUT exit, openUTM passes the address of the INPUT parameter area. The COPY member KCINPC is available to help you structure the INPUT parameter area; the name of the data structure is KCINPUTC.

**B** In BS2000/OSD, you can also pass the address of a control field area. The COPY member **B** KCCFC is available for the control field area; the name of the structure is KCCFILDC.

For further information on event exits, refer to [chapter "Event functions" on page 447](#).

*Example: INPUT exit*

```
DATA DIVISION.
*****
WORKING-STORAGE SECTION.
.
.
*****
LINKAGE SECTION.
COPY KCINPC.
*****
PROCEDURE DIVISION USING KCINPUTC [,KCCFILDC].
.
.
```

## 10.1.4 Platform-specific features in BS2000/OSD

### **B Notes on the DYNAMIC clause (COBOL85)**

**B** The DYNAMIC clause should not be used in load modules that are to be replaced, because  
**B** the working storage space allocated dynamically by this clause would not be released again  
**B** when replacement takes place.

**B** When replacing load modules which make use of the dynamic allocation of working storage,  
**B** memory overflow errors can occur.

### **B Programming program units with COBOL 2000**

**B** The lifetime of the objects is limited to one program unit run. At the PEND call, all objects  
**B** created in a program unit run are therefore released by the runtime system. This also  
**B** applies in the case of PEND variants without a change of process.

**B** Object references therefore cannot be preserved after the end of a program unit run in order  
**B** to be passed on to follow-up program units. In other words, they cannot be preserved in  
**B** UTM storage areas either.

**B** All LLMs that contain, use or inherit COBOL2000 modules with class definitions must  
**B** always be replaced together with the modified class definition. In other words, if a class  
**B** definition changes, all users of this class and the classes derived from it and their users  
**B** must be replaced together with the modified class definition.

### **B Compiling COBOL program units**

**B** COBOL program units can be compiled with COBOL85 or COBOL 2000 (see the  
**B** "COBOL85 User Guide" or "COBOL 2000 User Guide").

**B** You must specify the following COMOPT parameter TRUNCATE-LITERAL=NO when  
**B** compiling a UTM program unit. The COMOPT parameter TRUNCATE-LITERAL=NO is no  
**B** longer described for COBOL85 as of V1.2. In the interests of compatibility, COBOL85  
**B** nevertheless still supports it. When compiling a UTM program unit with COBOL85, it  
**B** continues to be mandatory to specify TRUNCATE-LITERAL=NO.

**B** COBOL85 then issues a message informing you that the parameter does not conform to  
**B** the ANS85 standard; this does not, however, affect the compiler run.

**B** When compiling with COBOL2000 you must specify the COMOPT parameter MARK-LAST-  
**B** PARAMETER=YES.

**B Using shareable code**

**B** If you are intending to load COBOL program units and make them shareable, you must  
**B** specify the following option when compiling them:

**B** \*COMOPT GENERATE-SHARED-CODE=YES

**B** The shareable code does not necessarily have to be stored in a separate object module: it  
**B** can also be stored together with the non-shareable part in a link and load module (LLM),  
**B** which is subdivided into a public slice and a private slice. To do this you must specify the  
**B** compiler option

**B** COMOPTGEN-LLM=YES.

**B** The shareable program units only need to be loaded once for all tasks in the application(s).  
**B** In the memory used locally by the task, all you then have to do is load the non-shareable  
**B** parts.

**B** openUTM offers a variety of ways of loading shareable objects:

- B** – as a non-privileged subsystem,
- B** – in a common memory pool in the user storage area (class 6 memory).

**B** For further information on how to compile shareable code, refer to the manual for your  
**B** compiler. The openUTM manual “Using openUTM Applications under BS2000/OSD”  
**B** provides a detailed explanation of how to link and load shareable code.

**B Generating formats with IFG**

**B** The manual “IFG for FHS” offers in-depth information on how to generate formats using the  
**B** IFG. If these formats are to be generated for use in conjunction with openUTM, please note  
**B** the following points:

- B** ● The format name must not be more than 7 characters long.
- B** ● In the user profile you must select "Structure of the data transfer area"
  - B** – for #formats: separate attribute blocks and field contents
  - B** – for \*formats: non-aligned, without attribute fields
  - B** – for +formats: non-aligned, with attribute fields

- For \* and + formats you declare two addressing aids, one each for input and output. You must also define a prefix for each addressing aid. The following example shows how addressing aids can be used:

```
LINKAGE SECTION.
COPY KCKBC.
05 KBPROGAREA PIC X(100).
COPY KCPAC.
03 NB-EINGABE.
COPY IFORMA-LIB.
03 NB-AUSGABE.
COPY OFORMA-LIB.
```

where FORMA is the format name defined with IFG, I is the prefix for input and O the prefix for output. When using this format, you specify the format name for MPUT, FPUT or DPUT calls in the field KCMF as “\*FORMA” (for addressing aids without attribute fields) and as “+FORMA” (for addressing aids with attribute fields).

- When defining addressing aids, please note that, in the case of + and \* formats, openUTM removes the transaction code from the message at the start of the service for MGETs and FGETs (unless an INPUT exit explicitly prevents it from doing so). If the first field in the format contains the transaction code, provision must be made for this in addressing aids for input formatting. The example below suggests one way of doing this for a \*format or a +format:

```
LINKAGE SECTION.
.
COPY KCPAC.
03 NB.
05 TACA PIC X(002). 1)
05 TAC PIC X(008).
05 DATEN PIC X(220).
03 FILLER REDEFINES NB.
COPY FORMA-LIB.
.
MOVE MGET TO KCOP.
.
IF KCKNZVG = "F"
THEN CALL "KDCS" USING KCPAC, DATEN (Service start)
ELSE CALL "KDCS" USING KCPAC, NB (During service)
END-IF.
```

- 1) This field is mandatory for +formats but must be omitted for \*formats.

- When preparing to implement these addressing aids, the formats are to be stored in the formats library. This name should be specified in the FHS start parameters.

**B Extended line mode**

**B** The COPY element TIAMCTRC is available for work in extended line mode, but is not supplied with openUTM. This COPY member contains the data structure LINE-MODE-CONTROL-CHARACTERS with the symbolic names of the control characters. TIAMCTRC can be copied to the WORKING-STORAGE SECTION.

## 10.1.5 Platform-specific features in Unix systems

X Compiler Server Express V2.2 or higher by MicroFocus is required for openUTM.

### X Keywords

X New COBOL compilers include the keywords OBJECT-ID and RESTRICTED. These  
X keywords clash with names in the COPY elements of the UTM interfaces. There are two  
X ways of preventing such conflicts.

X If the COBOL program units are not actually object-oriented, specify the compiler options  
X REMOVE(OBJECT-ID) and REMOVE(RESTRICTED) at compilation.

X – If you want to preserve object-oriented functionality, you must modify the COPY  
X statements as follows.

X COPY *COPY-element* REPLACING OBJECT-ID BY *NEW-OBJECT-ID*.

X COPY *COPY-element* REPLACING RESTRICTED BY *NEW-RESTRICTED*.

X The new names must be used when data access is performed.

### X Environment variable

X If you want to use COBOL program units, you must set the COBDIR environment variable  
X for the COBOL compiler as described in the COBOL documentation, typically to  
X /opt/lib/cobol. The LD\_LIBRARY\_PATH environment variable should also be set to  
X /opt/lib/cobol/lib.

### X Compiling a COBOL program unit

X COBOL source programs must be compiled with *cob*. You will need to set the following  
X switches:

X -c to create an o. file

X -x for static linking

X -g to retain the symbol table during linkage

X To load a program dynamically or test it with animator *-a* you can delete the *.int* and *.idy* files  
X created by the compiler.

X If you want to generate 64-bit applications, you must use *cob64* instead of *cob*. If *cob64* is  
X linked to *cob*, *cob* already generates 64-bit applications. In this case, you must compile  
X 32-bit applications using *cob32* instead of *cob*.

X More information on generating openUTM applications with COBOL programs is provided  
X in the openUTM manual “Using openUTM Applications under Unix Systems and Windows  
X Systems”.

- X** **Shared objects**
- X** You generate shared objects with the following call.
- X** `cob -z -o shared-object Cobol-objects`
- X** You must strictly observe the sequence of the options and specified objects.
- X** If you want to generate a 64-bit application, the shared object must also have 64-bit capability.
- X**

## 10.1.6 Platform-specific features in Windows systems

### W Creating a COBOL program unit

W You can create COBOL programs with MicroFocus NetExpress as of V4.0 . At generation  
W you must specify the following kdcdef statement for these programs:

W `PROGRAM objectname, COMP=COB2 [ ,SHARED-OBJECT=shared_object_name]`

### W Compiling a COBOL program unit

W When compiling COBOL programs, you have to set the corresponding environment  
W variables:

W – %COBCPY% must point to the %UTMPATH%\copy-cobol85 directory in which the  
W COBOL copies are stored.

W – The environment variable %INCLUDE% may have to be extended by `<path>\include`  
W where `<path>` is the installation directory of the COBOL compiler.

W If you create clients on the basis of UPIC-L and XATMI, you must also specify the following  
W paths in %COBCPY%:

W – %UTMPATH%\upic\copy-cobol85 and %UTMPATH%\xatmi\copy-cobol85

W You compile programs by entering the NetExpress command `cobol` in a prompt window. If  
W the program is to be animated, it must be compiled by means of the `cobol /ANIM` command.

### W Linking the utmwork process

W To link the process, you enter the NetExpress command `cbllink` in a prompt window.

W All COBOL must be specified individually when linking.

W If both COBOL and C program units are linked to the UTM application, additional C runtime  
W libraries must be specified as well.

W It may be necessary to set the %LIB% environment variable to the directory containing  
W these libraries. The environment variable %LIB% may have to be extended by `<path>\lib`  
W where `<path>` is the installation directory of the COBOL compiler.

W In certain cases – if you want to use the Microsoft Visual C++ debugger, for example – you  
W have to call the NetExpress command `cb1names` first before you call the Microsoft linker  
W `link`.

W It is easy to take the options for the `link` command from the openUTM QuickStart Kit. The  
W `nmake` make file is stored in the `filebase` directory under `workcob.mak`.



- W Please note the following:
- W – If the program is to be animated, the /BD switch must be specified.
  - W – To define the COBOL main entries, the /mMainUtm switch must be used.
  - W – Instead of the mainutm.obj module, %UTMPATH%\sys\mainutmcob.obj must be used.
  - W – %UTMPATH%\sys\libwork.lib must be specified as an import library.

W The EntryPointSymbol statements for the linker run are no longer required. If they are nevertheless still there, this leads to a link error. The same applies to UPIC-L clients.

### W Using the CPIC and XATMI COBOL interfaces

W COBOL programs that use the CPIC or XATMI interfaces must be adapted slightly for use with MicroFocus NetExpress because of the WIN32 call conventions used. This applies to the use of both openUTM Server and openUTM Client on the basis of UPIC (local or remote):

- W 1. Before the DATA DIVISION, the following SPECIAL-NAMES paragraph must be inserted to define the WINAPI call convention:

```
W SPECIAL-NAMES.  
W CALL-CONVENTION 74 IS WINAPI.
```

- W 2. Each call of the CPIC or XATMI interface must comply with this convention. For example:

```
W CALL WINAPI"CMACCP" USING CONVERSATION-ID CM_RETCODE
```

W All the programs of the openUTM Quickstart Kit have already been adapted as appropriate.

W *Reserved keyword TIMEOUT when CPIC is used*

W In NetExpress from MicroFocus, TIMEOUT is a reserved word. However, since this word is contained in the COBOL copy CMCOBOL on account of the CPIC specification, this name must be replaced in the source. For example:

```
W COPY CMCOBOL REPLACING TIMEOUT BY CPIC-TIMEOUT.
```

### W Dynamic loading of application programs from DLLs

W This functionality allows you to add application programs dynamically and replace them during operation.

W Dynamically loaded DLLs are generated as under Unix systems with the following statement:

```
W SHARED-OBJECT dll_name, DIRECTORY= ..., LOAD= ... VERSION=...
```

W The operands must be supplied as under Unix systems. Note that *dll\_name* must have the extension .dll. It is imperative that you specify VERSION.

W The %LD\_LIBRARY\_PATH% and %PATH% environment variables are not used with the  
W DIRECTORY operand.

W You will find detailed information on generation in the openUTM manual "Generating Appli-  
W cations" in the section on the SHARED-OBJECT statement.

### W *Creating DLLs*

W DLLs are created using Microsoft Visual Studio as of Version 2005.

W To do this, proceed as follows:

- W – Select DLL (dynamic link library) as the project type.
- W – Insert the declspec(dllexport) statement in all the application programs. Example:  
W void declspec(dllexport) func(struct kc\_ca \*kb, struct work \*spab)
- W – Link to the %UTMPATH%\sys\libwork.lib import library.

### W *Use of DLLs*

W DLLs are searched for under the directory specified in the SHARED-OBJECT statement.  
W The %PATH% and %LD\_LIBRARY\_PATH% environment variables are not evaluated.

W When the DLL is used, openUTM generates self-explanatory K078 messages.

### W *Replacing and administering DLLs*

W Programs in DLLs can be added to an application dynamically and replaced during  
W operation in the same way that programs in shared objects can be under Unix systems.

W The description of KCDPROG in the openUTM manual "Using openUTM Applications  
W under Unix Systems and Windows Systems" on the replacement of shared objects also  
W applies by analogy to DLLs.

## W **NLS - Native Language Support**

W The %LANG% environment variable allows you to select the language in which the UTM  
W messages are output.

W The values "De" for German and "En" for English are supported. Any other values are  
W treated as "En".

## W **Changing message destinations and message texts**

W To change message destinations, as under Unix systems you have to use the `kdcmmod`  
W program to create a separate message module that must be compiled by the C compiler  
W and linked to the `utmwork` process.

W Message texts cannot be changed. The message texts from the DLLs supplied with the  
W product are always used. These are stored in the %UTMPATH%\nls\msg\%LANG% directories.

## 10.2 Programming examples in COBOL

This section provides you some simple examples of how to code a KDCS call and an example of a complete UTM application, including the KDCDEF generation.

### 10.2.1 Examples of individual KDCS calls

This section includes examples of code for the following KDCS calls:

- MGET
- MPUT
- DPUT
- MCOM with DPUT in a job complex
- APRO with MPUT for distributed processing

Since the other KDCS calls are coded in a similar manner, they are not all illustrated explicitly at this point in the manual.

In a KDCS call, KCPAC indicates the address of the KDCS parameter area and NB the address of the message area. It is assumed that you are using the COPY member KCOPC (constants for the operation codes).

#### MGET call

- An unformatted dialog message that is exactly 80 bytes long is to be received. If less than 80 characters are read, a prompt for re-entry of the message is to be issued.

```

      .
      .
      MOVE LOW-VALUE TO KCPAC.
      MOVE MGET     TO KCOP.
      MOVE 80       TO KCLA.
      MOVE SPACES TO KCMF.
      CALL "KDCS" USING KCPAC, NB.
      IF KCRCCC NOT = ZERO
          THEN PERFORM MGET-RETURN-CODE.  1)
      IF KCRLM NOT = KCLA
          THEN PERFORM ITERATION.         2)
  
```

- 1) If more than 80 characters are read, error handling is activated.
- 2) The routine ITERATION sends a prompt to the terminal requesting that the entry be repeated.

- The format "PIC15" has been requested by a terminal. The unprotected data is 500 characters long in various different format fields. This format is to be received by the program.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE MGET      TO KCOP.
MOVE 500      TO KCLA.
MOVE "*PIC15 " TO KCMF.
CALL "KDCS" USING KCPAC, IPIC15.
IF KCRCCC = "05Z" GO TO FORMAT-ERROR.      1)
IF KCRCCC NOT = ZERO GO TO MGET-RETURN-CODE.
.
.

```

- 1) In the routine 'FORMAT-ERROR', the format has to be output again to enable you to continue working with the correct format.

- An ongoing service may receive input that consists of a short message generated with the function key F2 followed by 10 characters of data. This input is intended to trigger a special function. The F2 key was assigned the return code 21Z during generation.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE MGET TO KCOP.
.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC = "21Z"                                1)
    THEN PERFORM MGET-2.
.
.
MGET-2.                                          2)
MOVE LOW-VALUE TO KCPAC.
MOVE MGET      TO KCOP.
MOVE 10       TO KCLA.
MOVE SPACES TO KCMF.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM MGET-RETURN-CODE.

```

- 1) A special function is called.
- 2) An additional MGET is needed for the 10 characters of data.

**MPUT call**

- An 80-byte long unformatted message is to be sent to the terminal.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE MPUT      TO KCOP.
MOVE "NE"     TO KCOM.
MOVE 80       TO KCLM.
MOVE SPACES   TO KCRN.
MOVE SPACES   TO KCMF.
MOVE ZERO     TO KCDF.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM MPUT-RETURN-CODE.

```

- A 500-byte long formatted message is to be sent to the terminal. The name of the format is "PIC15". The screen is to be deleted before the message is sent.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE MPUT      TO KCOP.
MOVE "NE"     TO KCOM.
MOVE 500      TO KCLM.
MOVE SPACES   TO KCRN.
MOVE "*PIC15" TO KCMF.
MOVE KCREPL   TO KCDF.          1)
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM MPUT-RETURN-CODE.

```

- 1) **REPLACE** is executed by default whenever a format is replaced. The output is generated to preclude the possibility of errors resulting from undefined field contents.

- In a \*format called "PIC10" which, according to the most recent terminal input, still exists, all unprotected fields are to be deleted by way of a response.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE MPUT      TO KCOP.
MOVE "NE"     TO KCOM.
MOVE ZEROES   TO KCLM.
MOVE SPACES   TO KCRN.
MOVE "*PIC10" TO KCMF.
MOVE KCERAS   TO KCDF.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM MPUT-RETURN-CODE.

```

**DPUT call**

- An asynchronous job with an 11-character long message is to be passed on November 11 (= the 315th day of the year) at 11.11 a.m. to a program unit (absolute time specification). The relevant TAC is "ALAAF".

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE DPUT      TO KCOP.
MOVE "NE"      TO KCOM.
MOVE 11        TO KCLM.
MOVE "ALAAF"   TO KCRN.
MOVE ZERO      TO KCDF.
MOVE SPACES    TO KCMF.
MOVE "A"       TO KCMOD.
MOVE "315"     TO KCTAG.
MOVE "11"      TO KCSTD.
MOVE "11"      TO KCMIN.
MOVE "00"      TO KCSEK.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM DPUT-RETURN-CODE.

```

- An 80-character long message is to be output to the terminal 'DSS1' one hour from now (relative time specification), whereupon the screen function 'acoustic alarm' (BEL) is to be triggered.

```

.
.
MOVE LOW-VALUE TO KCPAC.
MOVE DPUT      TO KCOP.
MOVE "NE"      TO KCOM.
MOVE 80        TO KCLM.
MOVE "DSS1"    TO KCRN.
MOVE SPACES    TO KCMF.
MOVE KCALARM   TO KCDF.
MOVE "R"       TO KCMOD.
MOVE "000"     TO KCTAG.
MOVE "01"      TO KCSTD.
MOVE "00"      TO KCMIN.
MOVE "00"      TO KCSEK.
CALL "KDCS" USING KCPAC, NB.
IF KCRCCC NOT = ZERO
    THEN PERFORM DPUT-RETURN-CODE.

```

**Job complex: MCOM and DPUT calls**

A formatted asynchronous message (of 200 bytes) is to be printed out at 6.00 p.m. (= 18.00 hours) on the same day on PRINTER2. The confirmation returned by the printer is to be handled by a program.

If positive confirmation is returned, an asynchronous program with the TAC PRINTPOS receives a confirmation job with a 20-byte long message. If negative confirmation is returned, an asynchronous program with the TAC PRINTNEG is started (without a message). 80 bytes of user information is also logged in the event of negative confirmation. This information can be read with DADM UI as soon as the confirmation job becomes the main job. Confirmation jobs cannot be addressed by means of a job ID.

The job complex is encapsulated within two MCOM calls, which determine the destinations for the print job (= the basic job) and confirmation jobs in the MCOM BC call; the complex ID is "\*PRICOMP".

```

      .
      .
COMPLEX-BEGIN.
  MOVE LOW-VALUE TO KCPAC.
  MOVE MCOM      TO KCOP.
  MOVE BC       TO KCOM.
  MOVE "PRINTER2" TO KCRN.
  MOVE "PRINTPOS" TO KCPOS.
  MOVE "PRINTNEG" TO KCNEG.
  MOVE "*PRICOMP" TO KCCOMID.
  CALL "KDCS" USING KCPAC.
  IF KCRCCC NOT = ZERO
    THEN PERFORM MCOM-RETURN-CODE.
DPUT-NE.
  MOVE DPUT      TO KCOP.
  MOVE "NE"      TO KCOM.
  MOVE 200      TO KCLM.
  MOVE "*PRICOMP" TO KCRN.
  MOVE "*FORM1"  TO KCMF.
  MOVE ZERO     TO KCDF.
  MOVE "A"      TO KCMOD.
  MOVE KCTJHVG  TO KCTAG.
  MOVE "18"     TO KCSTD.
  MOVE "00"     TO KCMIN.
  MOVE "00"     TO KCSEK.
  CALL "KDCS" USING KCPAC, NB1.
  IF KCRCCC NOT = ZERO
    THEN PERFORM DPUT-RETURN-CODE.
DPUT-PLUS-T.

```

```

*****
* Confirmation job in positive case
*****
      MOVE LOW-VALUE TO KCPAC.
      MOVE DPUT      TO KCOP.
      MOVE "+T"      TO KCOM.
      MOVE 20        TO KCLM.
      MOVE "*PRICOMP" TO KCRN.
      MOVE SPACES    TO KCMF.
      MOVE ZERO      TO KCDF.
      CALL "KDCS" USING KCPAC, NB2.
      IF KCRCCC NOT = ZERO
          THEN PERFORM DPUT-RETURN-CODE.
      DPUT-USER-INFO.
*****
* User information for negative case
*****
      MOVE LOW-VALUE TO KCPAC.
      MOVE DPUT      TO KCOP.
      MOVE "-I"      TO KCOM.
      MOVE 80        TO KCLM.
      MOVE "*PRICOMP" TO KCRN.
      MOVE SPACES    TO KCMF.
      MOVE ZERO      TO KCDF.
      CALL "KDCS" USING KCPAC, NB3.
      IF KCRCCC NOT = ZERO
          THEN PERFORM DPUT-RETURN-CODE.
      DPUT-MINUS-T.
*****
* Confirmation job in negative case
*****
      MOVE LOW-VALUE TO KCPAC.
      MOVE DPUT      TO KCOP.
      MOVE "-T"      TO KCOM.
      MOVE ZERO      TO KCLM.
      MOVE "*PRICOMP" TO KCRN.
      MOVE SPACES    TO KCMF.
      MOVE ZERO      TO KCDF.
      CALL "KDCS" USING KCPAC, NB4.
      IF KCRCCC NOT = ZERO
          THEN PERFORM DPUT-RETURN-CODE.
      COMPLEX-END.
      MOVE LOW-VALUE TO KCPAC.
      MOVE MCOM      TO KCOP.
      MOVE EC        TO KCOM.
      MOVE "*PRICOMP" TO KCCOMID.
      CALL "KDCS" USING KCPAC.
      IF KCRCCC NOT = ZERO
          THEN PERFORM MCOM-RETURN-CODE.
      .
      .

```



**Example of distributed processing: APRO call with a subsequent MPUT**

The job-submitting service is to address the dialog service with the transaction code 'LTAC1' for the job-receiving application 'PARTNER1' (double-step addressing). In this context, the job-receiving service is to be assigned the service ID '>VGID1'. A 100-byte long MPUT message is then to be sent in line mode to the partner application.

```
.  
MOVE LOW-VALUE TO KCPAC.  
MOVE APRO      TO KCOP.  
MOVE "DM"      TO KCOM.  
MOVE ZERO      TO KCLM.  
MOVE "LTAC1   " TO KCRN.  
MOVE "PARTNER1" TO KCPA.  
MOVE ">VGID1  " TO KCPI.  
CALL "KDCS" USING KCPAC.  
IF KCRCCC NOT = ZERO  
    THEN PERFORM APRO-RETURN-CODE.  
.br/>.br/>MOVE LOW-VALUE TO KCPAC.  
MOVE MPUT      TO KCOP.  
MOVE "NE"      TO KCOM.  
MOVE 100       TO KCLM.  
MOVE ">VGID1" TO KCRN.  
MOVE SPACES    TO KCMF.  
MOVE ZEROES    TO KCDF.  
CALL "KDCS" USING KCPAC, NB.  
IF KCRCCC NOT = ZERO  
    THEN PERFORM MPUT-RETURN-CODE.
```

## 10.2.2 Example of an INPUT exit

The INPUT exit "FORINPUT" is called for input made in format mode and responds to such input as follows:

- User commands are issued:

Press the F1 key: KDCOUT

Press the F2 key: KDCDISP

KDCOFF: The first character in the input is "/"; this is accepted only outside of a service.

- Missing or invalid input elicits an error code with the message K098.

If the user is also to be permitted to enter KDCLAST and KDCFOR, the program will have to be extended accordingly.

This INPUT exit is generated with the KDCDEF generation tool in the EXIT statement with  
EXIT PROGRAM=FORINPUT,USAGE=(INPUT,FORMMODE).

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
    FORINPUT.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 FUNC.
    05 FUNC2.
        10 COMMAND PIC X.
            88 KDCOFF VALUE "/".
        10 REST PIC X(7).
*
77 KDCDISP PIC 9(4) COMP VALUE 2.
*
77 KDCOUT PIC 9(4) COMP VALUE 1.
*
77 CV-END PIC X(2) VALUE "EC".
*
*****
LINKAGE SECTION.
COPY KCINPC.
*****
PROCEDURE DIVISION USING KCINPUTC.
*****
P1-KEY-CONTROL-SECTION.
*
    IF KCIFKEY = KDCOUT
    THEN
        MOVE "KDCOUT" TO KCINCMD
        MOVE "CD" TO KCICCD
        MOVE "N" TO KCICUT
        MOVE SPACES TO KCIERRCD
        GO TO P99-END.
    Check F-keys *
```

```

      IF KCIFKEY = KDCDISP
      THEN
          MOVE "KDCDISP" TO KCINCMD
          MOVE "CD"      TO KCICCD
          MOVE "N"       TO KCICUT
          MOVE SPACES    TO KCIERRCD
          GO TO P99-END.
P2-CV-CONTROL.
      IF KCICVST NOT = CV-END
      THEN
          MOVE SPACES    TO KCINTAC
          MOVE "CC"      TO KCICCD
          MOVE "N"       TO KCICUT
          MOVE SPACES    TO KCIERRCD
          GO TO P99-END
      ELSE
          PERFORM P10-FUNC-CONTROL
          GO TO P99-END.
*****
P10-FUNC-CONTROL.
*****
*                               Check the first character of input *
      MOVE KCIFCH        TO FUNC2.
      IF KDCOFF
      THEN
          MOVE "KDCOFF"  TO KCINCMD
          MOVE "CD"      TO KCICCD
          MOVE "N"       TO KCICUT
          MOVE SPACE     TO KCIERRCD
          GO TO P10-END.
      IF KCICFINF NOT = "ON"
      THEN
          MOVE SPACE     TO KCINTAC
          MOVE "ER"      TO KCICCD
          MOVE "N"       TO KCICUT
          MOVE "ER01"    TO KCIERRCD
          GO TO P10-END.
P10-END.
      EXIT.
P99-END.
      EXIT PROGRAM.

```

### 10.2.3 Example of an asynchronous MSGTAC program unit

The MSGTAC program unit NOHACK counts the number of incorrect sign-on attempts in TLS. If openUTM accepts a KDCSIGN (i.e. with the message K008 or K033), the TLS is deleted.

If, after three invalid KDCSIGN attempts, the fourth KDCSIGN attempt is also incorrect, the relevant terminal is to be disconnected by means of "asynchronous administration", using an FPUT call with KCRN="KDCPTRMA". The message area contains the following administration command (see also the openUTM manual "Generating Applications":

```
PTERM=pterm, PRONAM=processor,ACT=DIS
```

The administration command is then written with LPUT to the user log file and the TLS is deleted.

The K messages are each read with an FGET by the MSGTAC program unit. Once a K message has been "processed", an FGET immediately reads the next K message within the same program unit run.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    MSGTAC.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
    COPY KCOPC.

77  ID-HACK-TLS                PIC X(8) VALUE "TLSHACK".
77  HACK-MAX                   PIC 9(4) COMP VALUE 3.

01  ADM-SATZ.
    02  ADM-TXT.
        03  F                   PIC X(07) VALUE "PTERM=( ".
        03  F                   PIC X(08).
        03  F                   PIC X(09) VALUE " ),PRONAM=" .
        03  F                   PIC X(08).
        03  F                   PIC X(11) VALUE " ,ACTION=DIS".
01  UTM-FEHLER-ZEILE.
    03  F                   PIC X(18) VALUE "Error in prog. unit".
    03  F-MODUL                PIC X(08) VALUE "NOHACK".
    03  F                       PIC X(12) VALUE " ; Vorg./TAC".
    03  F-VG                    PIC X(08).
    03  F                       PIC X(01) VALUE "/" .
    03  F-AL                    PIC X(08).
    03  F                       PIC X(05) VALUE " wg. ".
    03  F-OP                    PIC X(04).
    03  F                       PIC X(07) VALUE " (RC:".
    03  F-RC                    PIC X(08).
    03  F                       PIC X(01) VALUE ")".
```

```

LINKAGE SECTION.
COPY KCKBC.
05 FILLER                PIC X.

COPY KCPAC.

COPY KCMMSGC.
03 NB.
05 HACKER-LTERM        PIC X(8).
05 NB-ADM.
07 F                   PIC X(07).
07 PTRM                PIC X(08).
07 F                   PIC X(09).
07 PRNM                PIC X(08).
07 F                   PIC X(11).
05 TLS-HACK.
07 HACK-ANZ            PIC 9(4) COMP.

PROCEDURE DIVISION USING KCKBC, KCSPAB.

MAIN SECTION .
INIT-ANF.
MOVE LOW-VALUE TO KCPAC
MOVE INIT      TO KCOP
MOVE 0         TO KCLKBPRG
COMPUTE KCLPAB = FUNCTION LENGTH (KCSPAB)
CALL "KDCS" USING KCPAC.
IF KCRCCC NOT = ZERO
THEN GO TO PEND-LPUT.

FGET-ANF.
MOVE LOW-VALUE TO KCPAC
MOVE FGET      TO KCOP
COMPUTE KCLA   = FUNCTION LENGTH (KCMMSGC)
MOVE SPACE    TO KCMF
CALL "KDCS" USING KCPAC, KCMMSGC
IF KCRCCC NOT = ZERO
THEN
  IF KCRCCC = "10Z"
  THEN
    GO TO PEND-ANF
  ELSE
    GO TO PEND-LPUT.
IF MSGNR = "K004"
*                                     Invalid identification *
  MOVE LTRM OF K004 TO HACKER-LTERM
ELSE IF MSGNR = "K006"
*                                     Invalid password *
  MOVE LTRM OF K006 TO HACKER-LTERM
ELSE IF MSGNR = "K008"
*                                     KDCSIGN accepted *
  MOVE LTRM OF K008 TO HACKER-LTERM
ELSE IF MSGNR = "K031"
*                                     Card not ok *
  MOVE LTRM OF K031 TO HACKER-LTERM
ELSE IF MSGNR = "K033"
*                                     if no K008 is generated *
  MOVE LTRM OF K033 TO HACKER-LTERM

```



```

        CALL "KDCS" USING KCPAC, TLS-HACK
ELSE
    IF MSGNR = "K008"
        OR = "K033"
    THEN
*
        MOVE LOW-VALUE      TO KCPAC
        MOVE PTDA           TO KCOP
        MOVE 0              TO KCLA
        MOVE ID-HACK-TLS   TO KCRN
        MOVE HACKER-LTERM  TO KCLT
        CALL "KDCS" USING KCPAC, TLS-HACK
    ELSE
        PERFORM CHECK-NO.
A9.
    EXIT.
/
PRUEF-ANZ SECTION .
P0.
    ADD 1 TO HACK-NO
    IF HACK-NO NOT > HACK-MAX
    THEN
*
        MOVE LOW-VALUE      TO KCPAC
        MOVE PTDA           TO KCOP
        MOVE 2              TO KCLA
        MOVE ID-HACK-TLS   TO KCRN
        MOVE HACKER-LTERM  TO KCLT
        CALL "KDCS" USING KCPAC, TLS-HACK
        GO TO P9.
*
        MOVE ADM-TXT TO NB-ADM
        IF MSGNR = "K004"
            MOVE CORR K004 TO NB-ADM
        ELSE IF MSGNR = "K006"
            MOVE CORR K006 TO NB-ADM
        ELSE
            MOVE CORR K031 TO NB-ADM.
P-FPUT.
    MOVE LOW-VALUE TO KCPAC
    MOVE FPUT      TO KCOP
    MOVE "NE"     TO KCOM
    MOVE "KDCPTRMA" TO KCRN
    COMPUTE KCLM = FUNCTION LENGTH (NB-ADM)
    MOVE SPACE   TO KCMF
    MOVE ZERO    TO KCDF
    CALL "KDCS" USING KCPAC, NB-ADM
    IF KCRCCC NOT = ZERO
        GO TO P9.
P-LPUT.
*
    MOVE LOW-VALUE TO KCPAC
    MOVE LPUT      TO KCOP
    COMPUTE KCLA = FUNCTION LENGTH (NB-ADM)
    CALL "KDCS" USING KCPAC, NB-ADM
    IF KCRCCC NOT = ZERO
        GO TO P9.
P-PTDA.
*
    MOVE LOW-VALUE TO KCPAC

```

Ok; delete TLS \*

Try it once more \*

Disconnect !! \*

Write to user log \*

Delete TLS \*

```
MOVE PTDA          TO KCOP
MOVE ZERO          TO KCLA
MOVE ID-HACK-TLS  TO KCRN
MOVE HACKER-LTERM TO KCLT
CALL "KDCS" USING KCPAC, TLS-HACK.
P9. EXIT.
```

The above example for the MSGTAC program unit simply indicates appropriate ways of evaluating messages and administering the application.

However, the K094 message (SIGNON SILENT-ALARM) should be used to monitor security infringements since this also includes UPIC and OSI TP clients. Furthermore, wider-ranging administration of the UTM application is possible using the programmed administration capability (ADMI interface).



## 10.2.4 Example of a complete UTM application

### Example of address management

This sample application allows you to manage address data stored in a file. The application provides the following management functions for this purpose; each function can be called by means of an entry in the appropriate field in the relevant TAC. Input and output are both made in a format.

TAC	Function	
1	Display	displays one of the addresses in the file. The search string consists of the surname and the first two letters of the forename, which must be entered in the appropriate fields.
2	Add	enters a new address in the file. The file must not already contain an address with the same search string (see above).
3	Update	modifies an address entry. The address must already exist in the file.
4	Delete	deletes an existing address from the file.

If the user makes an error, an error message is displayed in the bottom line of the format.

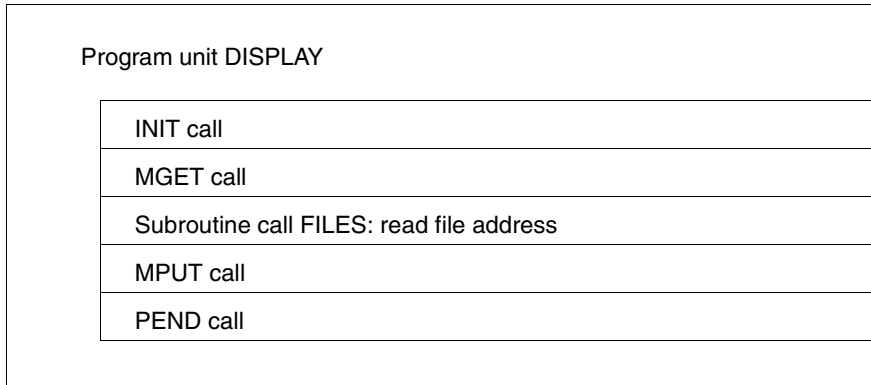
The figures indicated above are the transaction codes (TACs) which control the application. Transaction code 1 calls the program unit DISPLAY; transaction codes 2, 3 and 4 all call the program unit UPDATE. These program units then branch to the program unit FILES. The program unit FILES is implemented as the START and SHUT exit and contains the subroutines which implement input to and output from the address file.

openUTM calls the program unit BADTACS automatically if an invalid TAC is entered. Once the connection to the application has been established and KDCSIGN has been called successfully, openUTM immediately outputs the format (start format). Subsequent interaction with the user is strictly dialog-driven; in other words, the application responds to the input of a TAC and a key by outputting the format which contains the address being searched for and/or by outputting a success or an error message in the bottom line.

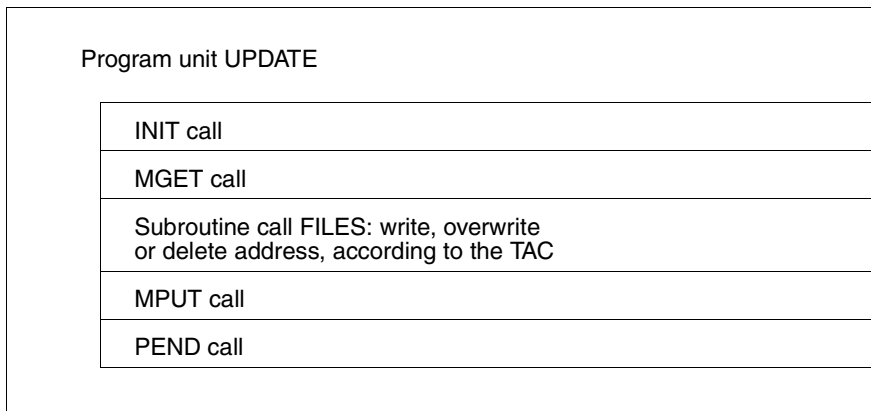


This program is intended merely to show you how you can program with openUTM. The file accesses depicted here are not subject to UTM's transaction management concept.

The following structure diagrams show the structure of the program units:



Structure diagram of program unit DISPLAY



Structure diagram of program unit UPDATE

For the sake of completeness, the generation of the application has also been appended to the COBOL program listings. To find out the exact meanings of the individual operands and statements, please refer to the openUTM manual "Generating Applications".

The figure below shows the format used for this application:

```

*****
*****      A d d r e s s   M a n a g e m e n t
*****
Please select a function: .....

Current function:  @@@@

Last name:+++++++          First name:++.....

Street:.....              No:.....

ZIP code:nnnnn           City:.....

Phone:.....

                                Function menu
1 = Display addresses      | 4 = Delete addresses
2 = Add new addresses     |
3 = Update addresses      | Quit with kdcoff
                            |

@@@@@
    
```

The \*format "FORMA" with which the application works

The structure of the addressing aid for this format is provided below:

```

*      USER-AREA-LEN: 228
41 TACO          PIC X(8).
41 FUNCTIONO    PIC X(26).
41 LASTNAMEO    PIC X(14).
41 FSTO         PIC X(2).
41 FSTRESTO     PIC X(18).
41 STREETO      PIC X(26).
41 HOUSENOO     PIC X(10).
41 ZIPO         PIC X(5).
41 CITYO        PIC X(26).
41 PHONEO       PIC X(18).
41 MSGTEXTO     PIC X(80).
    
```

The fields "FUNCTIONO" and "MSGTEXTO" are protected fields, the field "ZIPO" is numeric.

**Program unit DISPLAY**

```

IDENTIFICATION DIVISION.
PROGRAM-ID.      DISPLAY.
*****
ENVIRONMENT DIVISION.
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY KCOPC.
01 ERROR-TEXT.
   05 FILLER          PIC X(21)
      VALUE "***  E R R O R   ***".
   05 FILLER          PIC X(14)
      VALUE "PROGRAM UNIT: ".
   05 F-TP           PIC X(08).
   05 FILLER          PIC X(17)
      VALUE "      KDCS  OPCODE: ".
   05 F-OP           PIC X(04).
   05 FILLER          PIC X(13)
      VALUE "RETURN CODE: ".
   05 F-CD           PIC X(03).
LINKAGE SECTION.
COPY KCKBC.
   05 KBRPG          PIC X(228).
COPY KCPAC.
   03 NB.
      05 TAC          PIC X(008).
      05 DATA1       PIC X(220).
   03 FILLER REDEFINES NB.
COPY FORMAO.
*****
PROCEDURE DIVISION USING KCKBC KCSPAB.

INIT-OPERATION-SECTION.
MOVE SPACES TO NB.
MOVE INIT TO KCOP.
MOVE 0 TO KCLKBPRG.
MOVE 512 TO KCLPAB.
CALL "KDCS" USING KCPAC.
IF KCRCC NOT = ZERO
  THEN MOVE INIT TO F-OP GO TO ERROR-HANDLING.

MGET-OPERATION.
MOVE MGET TO KCOP.
MOVE 228 TO KCLA.
MOVE "*FORMA" TO KCMF.
CALL "KDCS" USING KCPAC DATA1.
IF KCRCC NOT = ZERO
  THEN MOVE MGET TO F-OP GO TO ERROR-HANDLING.
* CALL PROGRAM UNIT "FILES" IN ORDER TO CALL *
* READ ROUTINE *
READ-OPERATION.
CALL "FILES" USING KCKBC, KCSPAB.

```

```
MPUT-OPERATION.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 228       TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE "*FORMA"  TO KCMF.  
  CALL "KDCS"    USING KCPAC NB.  
  IF KCRCCC NOT = ZERO  
    THEN MOVE MPUT TO F-OP GO TO ERROR-HANDLING.  
  
PEND-OPERATION.  
  MOVE PEND      TO KCOP.  
  MOVE "FI"      TO KCOM.  
  CALL "KDCS"    USING KCPAC NB.  
  
PROG-END.  
  EXIT PROGRAM.  
  
ERROR-HANDLING.  
  MOVE "DISPLAY" TO F-TP.  
  MOVE KCRCCC    TO F-CD.  
  MOVE ERROR-TEXT TO NB.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 80        TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE SPACES    TO KCMF.  
  MOVE ZEROES    TO KCDF.  
  CALL "KDCS"    USING KCPAC NB.  
  MOVE PEND      TO KCOP.  
  MOVE "ER"      TO KCOM.  
  CALL "KDCS"    USING KCPAC.  
  GO TO PROG-END.
```

**Program unit UPDATE**

```

IDENTIFICATION DIVISION.
PROGRAM-ID.          UPDATE.
*****
ENVIRONMENT DIVISION.
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY KCOPC.
01 ERROR-TEXT.
   05 FILLER          PIC X(21)
      VALUE "***  E R R O R   ***".
   05 FILLER          PIC X(14)
      VALUE "PROGRAM UNIT: ".
   05 F-TP           PIC X(08).
   05 FILLER          PIC X(17)
      VALUE "      KDCS  OPCODE: ".
   05 F-OP           PIC X(04).
   05 FILLER          PIC X(13)
      VALUE "RETURN CODE: ".
   05 F-CD           PIC X(03).
LINKAGE SECTION.
COPY KCKBC.
   05 KBPRG          PIC X(228).
COPY KCPAC.
   03 NB.
      05 TAC          PIC X(008).
      05 DATA1       PIC X(220).
   03 FILLER REDEFINES NB.
COPY FORMAO.
*****
PROCEDURE DIVISION USING KCKBC, KCSPAB.
*****

INIT-OPERATION-SECTION.
MOVE SPACES TO NB.
MOVE INIT TO KCOP.
MOVE 0 TO KCLKBPRG.
MOVE 512 TO KCLPAB.
CALL "KDCS" USING KCPAC.
IF KCRCCC NOT = ZERO
    THEN MOVE INIT TO F-OP GO TO ERROR-HANDLING.

MGET-OPERATION.
MOVE MGET TO KCOP.
MOVE 228 TO KCLA.
MOVE "*FORMA" TO KCMF.
CALL "KDCS" USING KCPAC DATA1.
IF KCRCCC NOT = ZERO
    THEN MOVE MGET TO F-OP GO TO ERROR-HANDLING.
* CALL PROGRAM UNIT "FILES" IN ORDER TO BRANCH TO *
* WRITING, OVERWRITING AND DELETING ROUTINES *
* ACCORDING TO THE TAC *
FILE-OPERATION.
CALL "FILES" USING KCKBC, KCSPAB.

```

```
MPUT-OPERATION.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 228       TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE "*FORMA"  TO KCMF.  
  CALL "KDCS"    USING KCPAC NB.  
  IF KCRCCC NOT = ZERO  
    THEN MOVE MPUT TO F-OP GO TO ERROR-HANDLING.  
  
PEND-OPERATION.  
  MOVE PEND      TO KCOP.  
  MOVE "FI"      TO KCOM.  
  CALL "KDCS"    USING KCPAC NB.  
  
PROG-END.  
  EXIT PROGRAM.  
  
ERROR-HANDLING.  
  MOVE "UPDATE"  TO F-TP.  
  MOVE KCRCCC    TO F-CD.  
  MOVE ERROR-TEXT TO NB.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 80        TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE SPACES    TO KCMF.  
  MOVE ZEROES    TO KCDF.  
  CALL "KDCS"    USING KCPAC NB.  
  MOVE PEND      TO KCOP.  
  MOVE "ER"      TO KCOM.  
  CALL "KDCS"    USING KCPAC.  
  GO TO PROG-END.
```

**Program unit FILES with START/SHUT exit and file access operations**

```

IDENTIFICATION DIVISION.
PROGRAM-ID.    FILES.

ENVIRONMENT DIVISION.
*****

INPUT-OUTPUT SECTION.
*-----

FILE-CONTROL.
  SELECT ADDRESSES ASSIGN TO "addresses"
  ACCESS MODE IS RANDOM
  ORGANIZATION IS INDEXED
  RECORD KEY IS D-NAME
  FILE STATUS IS FILE-STATUS.

DATA DIVISION.
*****

FILE SECTION.
*-----

FD  ADDRESSES      LABEL RECORD IS STANDARD.
01  D-ADDRESSRECORD.
    05  D-NAME.
        10  D-LASTNAME          PIC X(14).
        10  D-FST               PIC X(02).
    05  D-FIRSTNAME          PIC X(18).
    05  D-STREET             PIC X(26).
    05  D-HOUSENO           PIC X(10).
    05  D-ZIP                PIC X(05).
    05  D-CITY               PIC X(26).
    05  D-PHONE              PIC X(18).

WORKING-STORAGE SECTION.
*-----

01  FILE-ERROR-LINE.
    05  FILLER                PIC X(24)
        VALUE "    *** FILE ERROR NO.: ".
    05  FILE-STATUS          PIC X(02).
    05  FILLER                PIC X(04)
        VALUE " ***".
    05  FILLER                PIC X(50) VALUE SPACES.

LINKAGE SECTION.
*-----

COPY KCKBC.
05  KBPRG                    PIC X(228).
COPY KCPAC.
03  NB.
    05  TAC                    PIC X(008).
    05  DATA1                 PIC X(220).
03  FILLER REDEFINES NB.
COPY FORMAO.

```



```
PROCEDURE DIVISION USING KCKBC KCSPAB.  
*****
```

```
CONTROLLING SECTION.  
*-----
```

```
CONTROLLING-BEGIN.  
  IF KCTACVG = "STARTUP"  
  THEN OPEN I-O ADDRESSES GO TO CONTROLLING-END.  
  IF KCTACVG = "SHUTDOWN"  
  THEN CLOSE ADDRESSES GO TO CONTROLLING-END.  
  IF KCTACVG = "1"  
  THEN GO TO READING-BEGIN.  
  IF KCTACVG = "2"  
  THEN GO TO WRITING-BEGIN.  
  IF KCTACVG = "3"  
  THEN GO TO OVERWRITING-BEGIN.  
  IF KCTACVG = "4"  
  THEN GO TO DELETING-BEGIN.
```

```
CONTROLLING-END.  
  EXIT PROGRAM.
```

```
READING SECTION.  
*-----
```

```
READING-BEGIN.  
  
*   SET THE ISAM KEY  
    MOVE LASTNAMEO TO D-LASTNAME.  
    MOVE FSTO     TO D-FST.  
    MOVE SPACES TO STREETO HOUSENOO CITYO PHONEO.  
    MOVE ZEROES TO ZIPO.  
    MOVE KCTACVG TO TACO.  
    MOVE "DISPLAY ADDRESSES" TO FUNCTIONO.  
    READ ADDRESSES RECORD  
    INVALID KEY PERFORM FILE-ERROR GO TO READING-END.  
    MOVE D-LASTNAME TO LASTNAMEO.  
    MOVE D-FST     TO FSTO.  
    MOVE D-FIRSTNAME TO FSTRESTO.  
    MOVE D-STREET TO STREETO.  
    MOVE D-HOUSENO TO HOUSENOO.  
    MOVE D-ZIP TO ZIPO.  
    MOVE D-CITY TO CITYO.  
    MOVE D-PHONE TO PHONEO.
```

```
READING-END.  
  EXIT PROGRAM.
```

```
WRITING SECTION.
*-----

WRITING-BEGIN.
  ENTRY  "WRITING"      USING  ADDRESSRECORD.
  MOVE  FSTO    TO  D-FST.
  MOVE  FSTRESTO TO  D-FIRSTNAME.
  MOVE  STREETO TO  D-STREET.
  MOVE  HOUSENOO TO  D-HOUSENO.
  MOVE  ZIPO    TO  D-ZIP.
  MOVE  CITYO   TO  D-CITY.
  MOVE  PHONEO  TO  D-PHONE.
  MOVE  KCTACVG TO  TACO.
  MOVE  "ADD NEW ADDRESSES" TO FUNCTIONO.
  MOVE  " * ADDRESS ADDED * " TO MSGTEXTO.
  WRITE D-ADDRESSRECORD INVALID KEY PERFORM FILE-ERROR.

WRITING-END.
  EXIT PROGRAM.

OVERWRITING SECTION.
*-----

OVERWRITING-BEGIN.

*   Read record to lock record
  MOVE LASTNAMEO TO D-LASTNAME.
  MOVE FSTO      TO D-FST.
  MOVE "UPDATE ADDRESSES      " TO FUNCTIONO.
  READ ADDRESSES RECORD
  INVALID KEY PERFORM FILE-ERROR GO TO OVERWRITING-END.
  MOVE FSTRESTO TO D-FIRSTNAME.
  MOVE STREETO  TO D-STREET.
  MOVE HOUSENOO TO D-HOUSENO.
  MOVE ZIPO     TO D-ZIP.
  MOVE CITYO    TO D-CITY.
  MOVE PHONEO   TO D-PHONE.
  MOVE " * ADDRESS UPDATED * " TO MSGTEXTO.
  REWRITE D-ADDRESSRECORD INVALID KEY PERFORM FILE-ERROR.

OVERWRITING-END.
  EXIT PROGRAM.
```

```
DELETING SECTION.
*-----

DELETING-BEGIN.

*   Read record to lock record
    MOVE LASTNAME0 TO D-LASTNAME.
    MOVE FST0 TO D-FST.
    MOVE "DELETE ADDRESSES" TO FUNCTION0.
    READ ADDRESSES RECORD
    INVALID KEY PERFORM FILE-ERROR GO TO DELETING-END.
    DELETE ADDRESSES RECORD
    INVALID KEY PERFORM FILE-ERROR GO TO DELETING-END.
    MOVE KCTACVG TO TACO.
    MOVE "*" ADDRESS DELETED "*" TO MSGTEXT0.

DELETING-END.
    EXIT PROGRAM.

FILE-ERROR SECTION.
*-----

FILE-ERROR-BEGIN.
    IF FILE-STATUS = 22 THEN
        MOVE "*** ADDRESS WITH THIS NAME ALREADY EXISTS ***"
        TO MSGTEXT0 GO TO FILE-ERROR-END.
    IF FILE-STATUS = 23 THEN
        MOVE "*** ADDRESS WITH THIS NAME DOES NOT EXIST ***"
        TO MSGTEXT0 GO TO FILE-ERROR-END.
    MOVE FILE-ERROR-LINE TO MSGTEXT0.

FILE-ERROR-END.
    EXIT.
```

**Program unit BADTACS**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. BADTACS.
*****
ENVIRONMENT DIVISION.
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
77 BTEXT          PIC X(41) VALUE
   "INCORRECT TAC - PLEASE REPEAT INPUT".
77 STAR          PIC X(6) VALUE ALL "*".
COPY KCOPC.
01 ERRORTXT.
05 FILLER          PIC X(21)
   VALUE "*** E R R O R ***".
05 FILLER          PIC X(14)
   VALUE "PROGRAM UNIT: ".
05 F-TP           PIC X(08).
05 FILLER          PIC X(17)
   VALUE "      KDCS OPCODE: ".
05 F-OP           PIC X(04).
05 FILLER          PIC X(13)
   VALUE "RETURN CODE: ".
05 F-CD           PIC X(03).
LINKAGE SECTION.
COPY KCKBC.
COPY KCPAC.
03 NB.
05 TRANSAC        PIC X(08).
05 DATA1         PIC X(220).
03 NB-A REDEFINES NB.
COPY FORMAO.
41 ERROR1 REDEFINES MSGTEXT0.
45 STAR1          PIC X(6).
45 BADTEXT        PIC X(41).
45 STAR2          PIC X(6).
45 REST           PIC X(27).
*****
PROCEDURE DIVISION USING KCKBC KCSPAB.
*****

INIT-OPERATION-SECTION.
MOVE SPACES TO NB.
MOVE INIT TO KCOP.
MOVE 0 TO KCLKBPRG.
MOVE 228 TO KCLPAB.
CALL "KDCS" USING KCPAC.
IF KCRCCC NOT = ZERO
  THEN MOVE INIT TO F-OP GO TO ERROR-HANDLING.

```

```
MGET-OPERATION.  
  MOVE MGET      TO KCOP.  
  MOVE 228       TO KCLA.  
  MOVE "*FORMA" TO KCMF.  
  CALL "KDCS"    USING KCPAC, DATA1.  
  IF KCRCCC = "05Z"  
    THEN MOVE SPACES TO NB-A.  
    GO TO MPUT-OPERATION.  
  IF KCRCCC NOT = ZERO  
    THEN MOVE MGET TO F-OP GO TO ERROR-HANDLING.  
  
MPUT-OPERATION.  
  MOVE BTEXT     TO BADTEXT.  
  MOVE STAR      TO STAR1.  
  MOVE STAR      TO STAR2.  
  MOVE SPACES    TO REST.  
  MOVE SPACES    TO TAC.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 228       TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE "*FORMA" TO KCMF.  
  CALL "KDCS"    USING KCPAC, NB.  
  IF KCRCCC NOT = ZERO  
    THEN MOVE MPUT TO F-OP GO TO ERROR-HANDLING.  
  
PEND-OPERATION.  
  MOVE PEND      TO KCOP.  
  MOVE "FI"      TO KCOM.  
  CALL "KDCS"    USING KCPAC.  
  
PROG-END.  
  EXIT PROGRAM.  
  
ERROR-HANDLING.  
  MOVE "BADTACS" TO F-TP.  
  MOVE KCRCCC    TO F-CD.  
  MOVE ERRORTXT TO NB.  
  MOVE MPUT      TO KCOP.  
  MOVE "NE"      TO KCOM.  
  MOVE 80        TO KCLM.  
  MOVE SPACES    TO KCRN.  
  MOVE SPACES    TO KCMF.  
  MOVE ZEROES    TO KCDF.  
  CALL "KDCS"    USING KCPAC NB.  
  MOVE PEND      TO KCOP.  
  MOVE "ER"      TO KCOM.  
  CALL "KDCS"    USING KCPAC.  
  GO TO PROG-END.
```

**B Generation of the sample application in BS2000**

```

B REM *****
B REM ***          D E F  -  S T A T E M E N T S          ***
B REM ***
B REM ***          K D C F I L E  =  A P P L I          ***
B REM *****
B MAX APPLNAME=A
B MAX KDCFILE=(KDCFILE.APPLI,S),TASKS=2,ASYNTASKS=0
B MAX CONRTIME=5,LOGACKWAIT=60
B ROOT ADR1ROOT
B OPTION GEN=ALL
B REM *****
B REM *****          P R O G R A M  S T A T E M E N T S          *****
B REM *****
B PROGRAM KDCADM,COMP=C
B PROGRAM DISPLAY,COMP=COB1
B PROGRAM UPDATE,COMP=COB1
B PROGRAM FILES,COMP=COB1
B PROGRAM BADTACS,COMP=COB1
B REM *****
B REM *****          E X I T  S T A T E M E N T S          *****
B REM *****
B EXIT PROGRAM=TPFILE,USAGE=START
B EXIT PROGRAM=TPFILE,USAGE=SHUT
B REM *****
B REM *****          T A C  S T A T E M E N T S          *****
B REM *****
B DEFAULT TAC ADMIN=Y,PROGRAM=KDCADM
B TAC KDCTAC
B TAC KDCLTERM
B TAC KDCPTERM
B TAC KDCSWTCH
B TAC KDCUSER
B TAC KDCSEND
B TAC KDCAPPL
B TAC KDCDIAG
B TAC KDCLOG
B TAC KDCINF
B TAC KDCHELP
B TAC KDCSHUT
B DEFAULT TAC TYPE=A,ADMIN=Y,PROGRAM=KDCADM
B TAC KDCTACA
B TAC KDCLTRMA
B TAC KDCPTRMA
B TAC KDCSWCHA
B TAC KDCUSERA
B TAC KDCSENA
B TAC KDCAPPLA
B TAC KDCDIAGA
B TAC KDCLOGA
B TAC KDCINF A
B TAC KDCHELPA
B TAC KDCSHUTA
B TAC KDCTCLA
B DEFAULT TAC TYPE=D,PROGRAM=(STD)
B TAC KDCBADTC,PROGRAM=BADTACS
B TAC 1,LOCK=1,PROGRAM=DISPLAY
B TAC 2,LOCK=2,PROGRAM=UPDATE

```

```

D  TAC 3, LOCK=2, PROGRAM=UPDATE
B  TAC 4, LOCK=2, PROGRAM=UPDATE

B  REM *****
B  REM *****          USER STATEMENTS          *****
B  REM *****
B  USER  GUENTER, PASS=C 'AUFGEHTS', KSET=BUND1, PERMIT=ADMIN, FORMAT=*FORMA
B  USER  BESSY, PASS=C 'HH', KSET=BUND2, STATUS=ON, FORMAT=*FORMA
B  USER  HAPPI, KSET=BUND3, STATUS=ON, FORMAT=*FORMA
B  REM *****
B  REM *****          FORMSYS STATEMENTS          *****
B  REM *****
B  FORMSYS TYPE=FHS
B  REM *****
B  REM *****          PTERM/LTERM STATEMENTS          *****
B  REM *****
B  DEFAULT PTERM PRONAM=DSR01, PTYPE=T9750
B  PTERM  DSS01, LTERM=UTMDST1
B  PTERM  DSS02, LTERM=UTMDST2
B  PTERM  DSS03, LTERM=UTMDST3
B  DEFAULT PTERM PRONAM=DSR01, PTYPE=T9022, USAGE=0
B  PTERM  GO1, LTERM=DRUCKER, CONNECT=A
B  LTERM  UTMDST1, KSET=BUND1
B  LTERM  UTMDST2, LOCK=4, KSET=BUND1
B  LTERM  UTMDST3, LOCK=5, KSET=BUND1
B  LTERM  DRUCKER, USAGE=0
B  REM *****
B  REM *****          KSET STATEMENTS          *****
B  REM *****
B  KSET  BUND1, KEYS=(1,2,3,4,5)
B  KSET  BUND2, KEYS=(1,2,4)
B  KSET  BUND3, KEYS=(1)
B  REM *****
B  REM *****          TLS STATEMENTS          *****
B  REM *****
B  TLS   TLSA
B  TLS   TLSB
B  END

```





# 11 Appendix

## 11.1 Overview of all KDCS calls

Overview of the entries in the KDCS parameter area and message area (NB) for KDCS calls. Binary zero should be set in fields that are not listed.

The key to the tables below is as follows:

- 0 binary zero
- B blank
- X any other specifications
- \* return values

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
APRO	AM DM		X X	X X	X X		X X	[X] [X]
CTRL	PR PE AB	0 0 0	0 0 0	X X X	B B B			Currently not used; must be passed

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
DADM	RQ	X	0	X	X	0	X <sup>1</sup>	*
	UI	X	0	X	0	0	X	*
	CS	0	0	X	0	0	X	*
	DL	0	0	X	X	0	X <sup>1</sup>	*
	DA	0	0	B	X	0	X <sup>1</sup>	*
	MV	0	0	X	X	0	X	*
	MA	0	0	B	X	0	X	*
DGET	FT	X	0	X	B	0	X	*
	NT	X	0	X	B	0	X	*
	BF	X	0	X	X	0	X	*
	BN	X	0	X	X	0	X	*
	PF	X	0	X	X	0	X	*
	PN	X	0	X	X	0	X	*
DPUT	NT		X	X	X	X	X	X
	NE		X	X	X	X	X	X
	NI	0	X	X	B	0	X	X
	QT	0	X	X	X	0	X <sup>1</sup>	X
	QE	0	X	X	X	0	X <sup>1</sup>	X
	QI	0	X	X	B	0	X	X
	+T	0	X	X	B	0	0	X
	-T	0	X	X	B	0	0	X
	+I	0	X	X	B	0	0	X
	-I	0	X	X	B	0	0	X
	RP	0	X	X	B	0	X	X
FGET		X			X			*
FPUT	NT		X	X	X	X		X
	NE		X	X	X	X		X
	RP		X	X	B	0		X
GTDA		X		X	X			*

B

B

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
INFO	CD	X						*
	DT	X						*
	LO	X						*
	PC	X						*
	SI	X						*
	CK							X
INIT		X	X					
	PU	X	X	0	0	X	0	*
	MD	X	0	0	0	0	0	
LPUT		X						X
MCOM	BC	0	0	X	X			
	EC	0	0	0	X			
MGET		X			X			*
	NT	X		X	X			*
MPUT	NT		X	X	X	X		X
	NE		X	X	X	X		X
	PM	0	X	B	X	X	0	X
	RM	0	X	X	0	X	0	X
	EM	0	0	X	B	0	0	X
	ES	0	X	0	X	0	0	X
	HM	0	0	X	B	0	0	X
PADM	OK	0	0	X	X	0	0	*
	PR	0	0	X	X	0	0	*
	AT	0	0	X	X	0	0	*
	AC	0	0	X	X	0	0	*
	CA	0	0	X	X	0	X	*
	CS	0	0	X	X	0	X	*
	AI	X	0	X	X	0	0	*
	PI	X	0	X	X	0	0	*

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
PEND	PA PR PS KP RE SP FC RS FR FI ER	0  0 0 0	0  0 0 0	X X X X X X B B	0  0 0 0	0  0 0 0	0  0 0 0	
PGWT	KP PR CM RB	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	X X X X	0 0 0 0	* * * *
PTDA		X		X	X			X
QCRE	NN	X	0	B	B	0	X	
	WN	X	0	X	B	0	X	
QREL	RL	0	0	X	B	0	0	
RSET								
SGET	KP	X		X				*
	RL	X		X				*
	GB	X		X				*
	US	X	0	X	X	0	0	*
SIGN	ST	X	0	0	0	0	0	X
	ON	X	0	0	X	0	0	X
	CP	X	0	0	0	0	0	X
	CK	X	0	0	0	0	0	X
	OF	0	0	0	0	0	0	
	OB	0	0	0	0	0	0	
	CL	0	0	0	0	0	0	

B

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
SPUT	DL	X		X				X
	MS	X		X				X
	ES	X		X				X
	GB	X		X				X
	US	X	0	X	X	0	0	X
SREL	LB			X				
	GB			X				

KDCS parameter area								NB/ 2nd parameter area
KCOP	KCOM	KCLA KCLKBPRG /kclcapa	KCLM KCLPAB /kclspa	KCRN	KCMCOM			
					KCMF /kcfn KCLT KCUS KCPA KCGTM	KCDF KCLI KCQRC	KCAPRO KCDPUT KCDGET KCQCRE KCEVENT KCPADM KCSGCL /kc_sgcl	
UNLK	GB DA US	0	0	X X X	X X	0	0	

<sup>1</sup> If KCLT contains the name of a USER or temporary queue, a value of U or Q must be specified in the KCQTYP field.

Overview of the values returned to the KDCS communication area for KDCS calls.

In the tables below, an asterisk (\*) always indicates a return value.

Call		KDCS communication area							KB program area
		KB hdr.	KB return area						
			KCRDF	KCRLM	KCRINFCC KCRMGT KCRST KCRSIGN	KCRCCC KCRCDC	KCRMF /kcrfn	KCRPI KCRUS KCRWVG KCRQN KCRQRC KCRGTM KCRD- PID KCRRC	
APRO	AM DM					*	*		
CTRL	PR PE AB					*	*	*	
DADM	RQ UI CS DL DA MV MA		*	*		*	*		

Call		KDCS communication area							
		KB hdr.	KB return area						KB program area
			KCRDF	KCRLM	KCRINFCC KCRMGT KCRST KCRSIGN	KCRCCC KCRCDC	KCRMF /kcrfn	KCRPI KCRUS KCRWVG KCRQN KCRQRC KCRGTM KCRD- PID KCRRC	
DGET	FT NT BF BN PF PN		*		*	*	*		
DPUT	NT NE NI QT QE QI +T -T +I -I RP				*				
FGET			*		*	*			
FPUT	NT NE RP				*				
GTDA			*		*				
INFO	CD DT LO PC SI		*		*				
	CK		*	*	*				

B

B

Call		KDCS communication area							
		KB hdr.	KB return area						KB program area
			KCRDF	KCRLM	KCRINFCC KCRMGT KCRST KCRSIGN	KCRCCC KCRCDC	KCRMF /kcrfn	KCRPI KCRUS KCRWVG KCRQN KCRQRC KCRGTM KCRD- PID KCRRC	
INIT		*				*	*	*	*
	PU	*		*		*	*	*	*
	MD					*			
LPUT					*				
MCOM	BC					*			
	EC					*			
MGET			*	*	*	*	*		
	NT			*	*	*	*	*	
MPUT	NT		*			*			
	NE					*			
	PM					*			
	RM					*			
	EM					*			
	ES					*			
	HM					*			
PADM	OK					*			
	PR					*			
	AT					*			
	AC					*			
	CA					*			
	CS					*			
	AI			*		*	*		
	PI			*		*	*		



Call		KDCS communication area							KB program area
		KB hdr.	KB return area					KCRPI KCRUS KCRWVG KCRQN KCRQRC KCRGTM KCRD- PID KCRRC	
			KCRDF	KCRLM	KCRINFCC KCRMGT KCRST KCRSIGN	KCRCCC KCRCDC	KCRMF /kcrfn		
PEND	PA PR PS KP RE SP FI FC RS ER FR					*			
PGWT <sup>1</sup>	KP PR CM RB		*			*	*	*	
PTDA						*			
QCRE	NN					*		*	
	WN					*			
QREL	RL					*			
RSET						*			*
SGET	KP RL GB US		*			*			
SIGN	ST		*	*		*	*	*	
	ON CP CK OF OB CL			*		*			

B

Call		KDCS communication area							
		KB hdr.	KB return area						KB program area
			KCRDF	KCRLM	KCRINFCC KCRMGT KCRST KCRSIGN	KCRCCC KCRCDC	KCRMF /kcrfn	KCRPI KCRUS KCRWVG KCRQN KCRQRC KCRGTM KCRD- PID KCRRC	
SPUT	DL MS ES GB US					*			
SREL	LB GB					*			
UNLK	GB DA US					*			

<sup>1</sup> KCRLM is only supplied when KCLI>0 was specified.

## 11.2 Different field names for C/C++ and COBOL

Throughout this manual, the COBOL field names (which are always written as uppercase letters) are used for the communication area and the KDCS parameter area. In C/C++, field names always use lowercase letters.

Since the field names for C/C++ are (unlike those for COBOL) derived from the corresponding English terms, further differences arise between the KB fields and those in the KDCS parameter area, apart from the different cases used.

Throughout this manual, wherever discrepancies arise which go beyond the simple issue of uppercase and lowercase, the C/C++ field name specified immediately following the COBOL field name; the two are separated by a slash. For example: “KCTAG/kcday”.

In the tables on the next few pages, all COBOL field names which differ from C/C++ field names in more than just the case used are shaded gray.

Moreover, you should also note the following points:

- in the data structures *kcdad.h* and *kcpad.h*, the fields for time specifications in C/C++ are **not** combined in a group.
- In the data structure *kcini.h*, the field names are structured differently to their counterparts in the COBOL data structure KCINIC (see table on [page 316ff](#)). These data structures allow you to structure the message area for the INIT call with the modifier PU, and for the call with KCLI>0.
- The screen function for reading the ID card reader in C/C++ has the symbolic name KCCARDRD.

B  
B

**Field names in the KB header area KCKBKOPF (ca\_hdr)**

<b>COBOL name</b>	<b>C/C++ name</b>	<b>Meaning</b>
KCBENID	kcuserid	user identification
KCTACVG	kccv_tac	service: name of the transaction code
KCTAGVG	kccv_day	start of service: day
KCMONVG	kccv_month	start of service: month
KCJHRVG	kccv_year	start of service: year
KCTJHVG	kccv_doy	start of service: day of the year
KCSTDVG	kccv_hour	start time of service: hour
KCMINVG	kccv_minute	start time of service: minute
KCSEKVG	kccv_second	start time of service: second
KCKNZVG	kccv_status	service: status information
KCTACAL	kcpr_tac	program run: name of the transaction code
KCSTDAL	kcpr_hour	start time of program run: hour
KCMINAL	kcpr_minute	start time of program run: minute
KCSEKAL	kcpr_second	start time of program run: second
KCAUSWEIS	kccard	status of card reader
KCTAIND	kctaind	transaction indicator
KCLOGTER	kclogter	logical terminal name
KCTERMN	kctermn	terminal mnemonic
KCLKBPB	kclpa	length of program area
KCHSTA	kchsta	stack level
KCDSTA	kcdsta	change in stack level
KCPRIND	kcprind	program indicator
KCOF1	kcof1	OSI TP functional unit
KCCP	kccp	client protocol
KCTARB	kctarb	transaction rollback indicator
KCYEARVG	kccv_year4	start of service: day of the year, 4-digit

**Field names in the KB return area KCRFELD (ca\_rti)**

<b>COBOL name</b>	<b>C/C++ name</b>	<b>Meaning</b>
KCRDF	kcrdf	device feature
KCRMLM	kcrilm	input message length
KCRINFCC	kcrinfcc	return information from INFO CK
KCVGST	kcpcv_state	service state of partner
KCTAST	kcpta_state	transaction state of partner
KCRMGT	kcrmgt	type of message
KCRSIGN	kcrsign	status of SIGN ON (complete code)
KCRSIGN1	kcrsign1	primary code of SIGN ON
KCRSIGN2 <sup>1</sup>	—	secondary code of SIGN ON
KCRCCC	kcrccc	compatible return code
KCRCKZ	kcrclid	identifier of DC system
KCRCDC	kcrcdc	return code of DC system
KCRMF	kcrfn	format name
KCRPI	kcrpi	service identification
KCRQN	kcrqn	return queue name
KCRWVG	kcrwvg	return number waiting vg
KCRUS	kcrus	return user (SIGN ST, DGET FT)
KCRQRC	kcrqrc	queue specific redelivery counter
KCRGTM	kcrgtm	creation time of DGET message
KCRDPID	kcrdpid	DPUT ID of DGET message
KCRRC	kcrrc	redelivery counter of DGET message

<sup>1</sup> The field KCRSIGN2 is not defined in the C/C++ data structure ca\_rti; the secondary code for the SIGN call is defined in the second and third byte of kcrinfcc.

**Field names in the KDCS parameter area KCPAC (kc\_pa)**

<b>COBOL name</b>	<b>C/C++ name</b>	<b>Meaning</b>
KCOP	kcop	operation code
KCOM	kcom	operation modification
KCLA	kcla	length of data area
KCLKBPRG	kclcapa	length of ca program area
KCLM	kclm	length of message (part)
KCWTIME	kcwtime	waiting time for DGET messages
KCLPAB	kclspa	length of standard primary area
KCRN	kcrn	reference name
KCMF	kcfm	format name
KCLT	kclt	logical terminal name
KCUS	kcus	name of user
KCPA	kcpa	partner application name
KCOF	kcof	OSI functions
KCDF	kcdf	device feature
KCLI	kcli	length of init area
EXTENT	kcext	extensions for DPUT, APRO and PADM
KCDPUT	kcdput	extension for DPUT function
KCMOD	kcmmod	DPUT: modifier
KCTAG	kcdays	DPUT: days
KCSTD	kchour	DPUT: hours
KCMIN	kcmin	DPUT: minutes
KCSEK	kcsec	DPUT: seconds
KCQTYP	kcqtyp	queue type
KCQMODE	kcqmode	queue mode
KCAPRO	kcapro	extension for APRO function
KCPI	kcpi	APRO: process identification
KCPADM	kcpadm	extension for PADM function
KCACT	kcact	PADM: action
KCADRLT	kcadrllt	PADM: lterm name
KCMCOM	kcmcom	redefinition for MCOM function
KCPOS	kcpos	MCOM: destination in positive case
KCNEG	kcneg	MCOM: destination in negative case

	<b>COBOL name</b>	<b>C/C++ name</b>	<b>Meaning</b>
	KCCOMID	kccomid	MCOM: complex identification
<b>B</b>	KCSGCL	kc_sgcl	extensions for SIGN CL
<b>B</b>	KCLANGID	kclangid	language ID of user
<b>B</b>	KCTERRID	kcterrid	territorial ID of user
<b>B</b>	KCCSNAME	kccsname	character set name of user
	KCGTM	kcgtm	creation time of message (generation time)
	KCQRC	kcqrc	queue-specific redelivery counter
	KCDPID	kcdpid	DPUT ID of message

## 11.3 ASCII-EBCDIC code conversion

### 11.3.1 BS2000/OSD

B In the case of communication from BS2000 to partners by means of the TCP/IP protocol,  
B the partners generally work with ASCII or ISO 8859-1 code, whereas BS2000 generally  
B works with EBCDIC code. To ensure that communication from BS2000 to these partners is  
B nevertheless easy, openUTM offers automatic code conversion. You can activate automatic  
B code conversion at KDCDEF generation partner-specifically with the help of the MAP=  
B operand in the PTERM or TPOOL statement. You can use different conversion tables for  
B the conversion.

#### B Conversion tables

B Up to four different conversion tables can be used for conversion. Table 1 is supplied already  
B filled. It is provided for 7-bit ASCII (the eight bit is ignored in ASCII code). Tables 2, 3 and 4  
B are still free and can be filled by the user. They are evaluated with the full 8 bits.

B The tables are defined in the assembly language module KDCEA. The source for KDCEA  
B is in the SYSLIB.UTM.061.EXAMPLE library. If a table is to be changed, this module must  
B be modified. The module must then be reassembled and linked to the root by means of an  
B INCLUDE statement. This must be done before the resolve takes effect on the  
B SYSLNK.UTM.061 library.

B Which of the tables is to be used is specified for the MAP= operand in the PTERM or  
B TPOOL statement when the partner is generated.

### 11.3.2 Unix systems and Windows systems

X/W When exchanging unformatted messages of a UTM application with a partner application,  
X/W openUTM has the ability to automatically execute a ASCII-EBCDIC code conversion. You  
X/W can activate automatic code conversion during KDCDEF generation for specific partners  
X/W using the MAP=SYSTEM operand in the KDCDEF statements PTERM, TPOOL, OSI-CON  
X/W and SESCHA.

X/W openUTM uses a standard table for conversion. This is in:

X *utmpath/src/kcsaeea.c* (Unix systems)

W *utmpath\src\kcsaeea.c* (Windows systems)



### 11.3.2.1 Modifying the code table in Unix systems

- X You can modify this standard table for UTM applications in Unix systems. Proceed as follows to do this:
- X 1. Copy the file `kcsaeea.c` into a separate directory.
- X 2. Modify the table as desired.
- X 3. Compile the modified source file.
- X 4. Link the work process by specifying the `.o` object before the `libwork.a` library. If you link with the `libwork.so` library, then you must regenerate this library beforehand. The script `utmpath/shsc/stat2dyn` is provided for this purpose.

### 11.3.2.2 Modifying the code table in Windows systems

- W The code conversion tables are contained in the library `utmconvt.dll`. `utmconvt.dll` is located in the same directory as `libwork.dll`.
- W You can modify these conversion tables to suit your own needs by modifying the supplied source files and then creating a modified `utmconvt.dll`.

#### W Source files for creating new code tables

- W The directory `UTMPATH\src` contains the following source files which you may need to modify:

- W ● `kcsaeea.c` and `kcxaent.c`

- W `kcsaeea.c` contains the conversion tables for previous openUTM-Server versions and is used in the `utmconvt.dll` supplied with the product.
- W `kcxaent.c` contains the complete tables for conversion between the Windows character set and EBCDIC.

- W These files are C source files and each contain two character arrays with 256 elements. One array is used for conversion from ASCII to EBCDIC and the other array is used for conversion from EBCDIC to ASCII (see the example below).

- W ● `utmconvt.def`

- W Definition file containing EXPORT statements. It is not necessary to modify this file.

- W ● `utmconvt.rc` and `resource.h`

- W Resource files containing version and copyright information. This information is displayed when you right-click the DLL file and choose *Properties*. These files need not necessarily be linked in.

**W** **Modifying the library utmconvt.dll****W** Three steps are necessary to convert the library `utmconvt.dll`:**W** 1. Modify the code table to suit your requirements (as necessary). Do this by editing  
**W** `kcsaeee.c` or `kcxaent.c` with a text editor.**W** If you only wish to use the conversion table `kcxaent.c` in place of the standard  
**W** conversion table `kcsaeee.c`, you can omit step 1.**W** *Example***W** You wish to incorporate German umlauts in the conversion table `kcsaeee.c`. Proceed  
**W** as follows for the letter 'Ä':**W** a) Find out the code for 'Ä'. 'Ä' has the code X'C4' (= decimal 196) in ISO 8859-1 and  
**W** the code X'63' (= decimal 99) in EBCDIC.DF.04-1.**W** b) Change the value of `kcsaebc[196]` from `0xff` to `0x63` (ASCII-to-EBCDIC  
**W** conversion)**W** c) Change the value of `kcseasc[99]` from `0x1a` to `0xc4` (EBCDIC-to-ASCII conversion)**W** Proceed in the same way for the remaining German umlauts.**W** 2. Start Microsoft Visual Studio and proceed as follows:**W** – Create a new project with the name `utmconvt` in the directory `UTMPATH\utmconvt`.  
**W** The project must be of the type `Dynamic-Link Library`.**W** – Add the following files to the project:**W** – The code table you modified (either `kcsaeee.c` or `kcxaent.c`, see 1.),**W** – `utmconvt.def`**W** – and, if required, `utmconvt.rc`.**W** – From this project, create the library `utmconvt.dll`.**W** – Close Visual Studio**W** 3. Replace the old library `utmconvt.dll` with the new library:**W** – First back up the library under a different name, so that you can access it again if  
**W** anything goes wrong.**W** – Copy the new `utmconvt.dll` to the directory which contains the UTM library**W** `libwork.dll` (this is generally `UTMPATH\ex`). Make sure that the original**W** `utmconvt.dll` is actually replaced by the new `utmconvt.dll`.**W** The new conversion library is ready for use

---

# Glossary

A term in *italic* font means that it is explained somewhere else in the glossary.

## **abnormal termination of a UTM application**

Termination of a *UTM application*, where the *KDCFILE* is not updated. Abnormal termination is caused by a serious error, such as a crashed computer or an error in the system software. If you then restart the application, openUTM carries out a *warm start*.

## **abstract syntax (OSI)**

Abstract syntax is defined as the set of formally described data types which can be exchanged between applications via *OSI TP*. Abstract syntax is independent of the hardware and programming language used.

## **acceptor (CPI-C)**

The communication partners in a *conversation* are referred to as the *initiator* and the acceptor. The acceptor accepts the conversation initiated by the initiator with `Accept_Conversation`.

## **access list**

An access list defines the authorization for access to a particular *service*, *TAC queue* or *USER queue*. An access list is defined as a *key set* and contains one or more *key codes*, each of which represent a role in the application. Users or LTERMs or (OSI) LPAPs can only access the service or *TAC queue/USER queue* when the corresponding roles have been assigned to them (i.e. when their *key set* and the access list contain at least one common *key code*).

## **access point (OSI)**

See *service access point*.

## **ACID properties**

Acronym for the fundamental properties of *transactions*: atomicity, consistency, isolation and durability.

## **administration**

Administration and control of a *UTM application* by an *administrator* or an *administration program*.

**administration command**

Commands used by the *administrator* of a *UTM application* to carry out administration functions for this application. The administration commands are implemented in the form of *transaction codes*.

**administration journal**

See *cluster administration journal*.

**administration program**

*Program unit* containing calls to the *program interface for administration*. This can be either the standard administration program *KDCADM* that is supplied with openUTM or a program written by the user.

**administrator**

User who possesses administration authorization.

**AES**

AES (Advanced Encryption Standard) is the current symmetric encryption standard defined by the National Institute of Standards and Technology (NIST) and based on the Rijndael algorithm developed at the University of Leuven (Belgium). If the AES method is used, the UPIC client generates an AES key for each session.

**Apache Axis**

Apache Axis (Apache eXtensible Interaction System) is a SOAP engine for the design of Web services and client applications. There are implementations in C++ and Java.

**Apache Tomcat**

Apache Tomcat provides an environment for the execution of Java code on Web servers. It was developed as part of the Apache Software Foundation's Jakarta project. It consists of a servlet container written in Java which can use the JSP Jasper compiler to convert JavaServer pages into servlets and run them. It also provides a fully featured HTTP server.

**application context (OSI)**

The application context is the set of rules designed to govern communication between two applications. This includes, for instance, abstract syntaxes and any assigned transfer syntaxes.

**application entity (OSI)**

An application entity (AE) represents all the aspects of a real application which are relevant to communications. An application entity is identified by a globally unique name (“globally” is used here in its literal sense, i.e. worldwide), the *application entity title* (AET). Every application entity represents precisely one *application process*. One application process can encompass several application entities.

**application entity qualifier (OSI)**

Component of the *application entity title*. The application entity qualifier identifies a *service access point* within an application. The structure of an application entity qualifier can vary. openUTM supports the type “number”.

**application entity title (OSI)**

An application entity title is a globally unique name for an *application entity* (“globally” is used here in its literal sense, i.e. worldwide). It is made up of the *application process title* of the relevant *application process* and the *application entity qualifier*.

**application information**

This is the entire set of data used by the *UTM application*. The information comprises memory areas and messages of the UTM application including the data currently shown on the screen. If operation of the UTM application is coordinated with a database system, the data stored in the database also forms part of the application information.

**application process (OSI)**

The application process represents an application in the *OSI reference model*. It is uniquely identified globally by the *application process title*.

**application process title (OSI)**

According to the OSI standard, the application process title (APT) is used for the unique identification of applications on a global (i.e. worldwide) basis. The structure of an application process title can vary. openUTM supports the type *Object Identifier*.

**application program**

An application program is the core component of a *UTM application*. It comprises the main routine *KDCROOT* and any *program units* and processes all jobs sent to a *UTM application*.

**application restart**

see *warm start*

### **application service element (OSI)**

An application service element (ASE) represents a functional group of the application layer (layer 7) of the *OSI reference model*.

### **application warm start**

see *warm start*.

### **association (OSI)**

An association is a communication relationship between two application entities. The term “association” corresponds to the term *session* in *LU6.1*.

### **asynchronous conversation**

CPI-C conversation where only the *initiator* is permitted to send. An asynchronous transaction code for the *acceptor* must have been generated in the *UTM application*.

### **asynchronous job**

*Job* carried out by the job submitter at a later time. openUTM includes *message queuing* functions for processing asynchronous jobs (see *UTM-controlled queue* and *service-controlled queue*). An asynchronous job is described by the *asynchronous message*, the recipient and, where applicable, the required execution time.

If the recipient is a terminal, a printer or a transport system application, the asynchronous job is a *queued output job*. If the recipient is an *asynchronous service* of the same application or a remote application, the job is a *background job*. Asynchronous jobs can be *time-driven jobs* or can be integrated in a *job complex*.

### **asynchronous message**

Asynchronous messages are messages directed to a *message queue*. They are stored temporarily by the local *UTM application* and then further processed regardless of the job submitter. Distinctions are drawn between the following types of asynchronous messages, depending on the recipient:

- In the case of asynchronous messages to a *UTM-controlled queue*, all further processing is controlled by openUTM. This type includes messages that start a local or remote *asynchronous service* (see also *background job*) and messages sent for output on a terminal, a printer or a transport system application (see also *queued output job*).
- In the case of asynchronous messages to a *service-controlled queue*, further processing is controlled by a *service* of the application. This type includes messages to a *TAC queue*, messages to a *USER queue* and messages to a *temporary queue*. The USER queue and the temporary queue must belong to the local application, whereas the TAC queue can be in both the local application and the remote application.

**asynchronous program**

*Program unit started by a background job.*

**asynchronous service (KDCS)**

*Service which processes a background job. Processing is carried out independently of the job submitter. An asynchronous service can comprise one or more program units/transactions. It is started via an asynchronous transaction code.*

**audit (BS2000/OSD)**

During execution of a *UTM application*, UTM events which are of relevance in terms of security are logged by *SAT* for auditing purposes.

**authentication**

See *system access control*.

**authorization**

See *data access control*.

**Axis**

See *Apache Axis*.

**background job**

Background jobs are *asynchronous jobs* destined for an *asynchronous service* of the current application or of a remote application. Background jobs are particularly suitable for time-intensive processing or processing which is not time-critical and where the results do not directly influence the current dialog.

**basic format**

Format in which terminal users can make all entries required to start a service.

**basic job**

*Asynchronous job in a job complex.*

**browsing asynchronous messages**

A *service* sequentially reads the *asynchronous messages* in a *service-controlled queue*. The messages are not locked while they are being read and they remain in the queue after they have been read. This means that they can be read simultaneously by different services.

**bypass mode (BS2000/OSD)**

Operating mode of a printer connected locally to a terminal. In bypass mode, any *asynchronous message* sent to the printer is sent to the terminal and then redirected to the printer by the terminal without being displayed on screen.

### cache

Used for buffering application data for all the processes of a *UTM application*. The cache is used to optimize access to the *page pool* and, in the case of UTM cluster applications, the *cluster page pool*.

### CCS name (BS2000/OSD)

See *coded character set name*.

### client

Clients of a *UTM application* can be:

- terminals
- UPIC client programs
- transport system applications (e.g. DCAM, PDN, CMX, socket applications or UTM applications which have been generated as *transport system applications*).

Clients are connected to the UTM application via LTERM partners. openUTM clients which use the OpenCPIC carrier system are treated just like *OSI TP partners*.

### client side of a conversation

This term has been superseded by *initiator*.

### cluster

A number of computers connected over a fast network and which in many cases can be seen as a single computer externally. The objective of clustering is generally to increase the computing capacity or availability in comparison with a single computer.

### cluster administration journal

The administration journal files serve to pass on to the other node applications those administrative actions that are to apply throughout the cluster to all node applications in a UTM cluster application.

### cluster configuration file

File containing the central configuration data of a *UTM cluster application*. The cluster configuration file is created using the UTM generation tool *KDCDEF*.

### cluster GSSB file

File used to administer GSSBs in a *UTM cluster application*. The cluster GSSB file is created using the UTM generation tool *KDCDEF*.

### cluster lock file

File in a *UTM cluster application* used to manage cross-node locks of user data areas.



**cluster page pool**

The cluster page pool consists of an administration file and up to 10 files containing a *UTM cluster application's* user data that is available globally in the cluster (service data including LSSB, GSSB and ULS). The cluster page pool is created using the UTM generation tool *KDCDEF*.

**cluster start serialization file**

Lock file used to serialize the start-up of individual node applications (only in Unix systems and Windows systems).

**cluster ULS file**

File used to administer the ULS areas of a *UTM cluster application*. The cluster ULS file is created using the UTM generation tool *KDCDEF*.

**cluster user file**

File containing the user management data of a *UTM cluster application*. The cluster user file is created using the UTM generation tool *KDCDEF*.

**coded character set name (BS2000/OSD)**

If the product *XHCS* (eXtended Host Code Support) is used, each character set used is uniquely identified by a coded character set name (abbreviation: "CCS name" or "CCSN").

**cold start**

Start of a *UTM application* after the application terminates normally (*normal termination*) or after a new generation (see also *warm start*).

**communication area (KDCS)**

KDCS *primary storage area*, secured by transaction logging and which contains service-specific data. The communication area comprises 3 parts:

- the KB header with general service data
- the KB return area for returning values to KDCS calls
- the KB program area for exchanging data between UTM program units within a single *service*.

**communication resource manager**

In distributed systems, communication resource managers (CRMs) control communication between the application programs. openUTM provides CRMs for the international OSI TP standard, for the LU6.1 industry standard and for the proprietary openUTM protocol UPIC.

### configuration

Sum of all the properties of a *UTM application*. The configuration describes:

- application parameters and operating parameters
- the objects of an application and the properties of these objects. Objects can be *program units* and *transaction codes*, communication partners, printers, *user IDs*, etc.
- defined measures for controlling data and system access.

The configuration of a UTM application is defined at generation time and can be changed dynamically by the administrator (while the application is running). The configuration is stored in the *KDCFILE*.

Also:

### configuration

The process of defining the configuration of the UTM application. A distinction is made between *static* and *dynamic configuration*.

### confirmation job

Component of a *job complex* where the confirmation job is assigned to the *basic job*. There are positive and negative confirmation jobs. If the *basic job* returns a positive result, the positive confirmation job is activated, otherwise, the negative confirmation job is activated.

### connection bundle

see *LTERM bundle*.

### connection user ID

User ID under which a *TS application* or a *UPIC client* is signed on at the *UTM application* directly after the connection has been established. The following applies, depending on the client (= LTERM partner) generation:

- The connection user ID is the same as the USER in the LTERM statement (explicit connection user ID). An explicit connection user ID must be generated with a USER statement and cannot be used as a “genuine” *user ID*.
- The connection user ID is the same as the LTERM partner (implicit connection user ID) if no USER was specified in the LTERM statement or if an LTERM pool has been generated.

In a *UTM cluster application*, the service belonging to a connection user ID (RESTART=YES in LTERM or USER) is bound to the connection and is therefore local to the node.

A connection user ID generated with RESTART=YES can have a separate service in each *node application*.

**contention loser**

Every connection between two partners is managed by one of the partners. The partner that manages the connection is known as the *contention winner*. The other partner is the contention loser.

**contention winner**

A connection's contention winner is responsible for managing the connection. Jobs can be started by the contention winner or by the *contention loser*. If a conflict occurs, i.e. if both partners in the communication want to start a job at the same time, then the job stemming from the contention winner uses the connection.

**conversation**

In CPI-C, communication between two CPI-C application programs is referred to as a conversation. The communication partners in a conversation are referred to as the *initiator* and the *acceptor*.

**conversation ID**

CPI-C assigns a local conversation ID to each *conversation*, i.e. the *initiator* and *acceptor* each have their own conversation ID. The conversation ID uniquely assigns each CPI-C call in a program to a conversation.

**CPI-C**

CPI-C (Common Programming Interface for Communication) is a program interface for program-to-program communication in open networks standardized by X/Open and CIW (**C**PI-C **I**mplementor's **W**orkshop). The CPI-C implemented in openUTM complies with X/Open's CPI-C V2.0 CAE Specification. The interface is available in COBOL and C. In openUTM, CPI-C can communicate via the OSI TP, *LU6.1* and UPIC protocols and with openUTM-LU62.

**Cross Coupled System / XCS**

Cluster of BS2000 computers with the *Highly Integrated System Complex Multiple System Control Facility* (HIPLEX<sup>®</sup> MSCF).

**data access control**

In data access control openUTM checks whether the communication partner is authorized to access a particular object belonging to the application. The access rights are defined as part of the configuration.

### **dead letter queue**

The dead letter queue is a TAC queue which has the fixed name KDCDLETQ. It is always available to save queued messages sent to transaction codes or TAC queues but which could not be processed. The saving of queued messages in the dead letter queue can be activated or deactivated for each message destination individually using the TAC statement's DEAD-LETTER-Q parameter.

### **DES**

DES (Data Encryption Standard) is an international standard for encrypting data. One key is used in this method for encoding and decoding. If the DES method is used, the UPIC client generates a DES key for each session.

### **dialog conversation**

CPI-C conversation in which both the *initiator* and the *acceptor* are permitted to send. A dialog transaction code for the *acceptor* must have been generated in the *UTM application*.

### **dialog job, interactive job**

Job which starts a *dialog service*. The job can be issued by a *client* or, when two servers communicate with each other (*server-server communication*), by a different application.

### **dialog message**

A message which requires a response or which is itself a response to a request. The request and the response both take place within a single service. The request and reply together form a dialog step.

### **dialog program**

*Program unit* which partially or completely processes a *dialog step*.

### **dialog service**

*Service* which processes a *job* interactively (synchronously) in conjunction with the job submitter (*client* or another server application) . A dialog service processes *dialog messages* received from the job submitter and generates dialog messages to be sent to the job submitter. A dialog service comprises at least one *transaction*. In general, a dialog service encompasses at least one dialog step. Exception: in the event of *service chaining*, it is possible for more than one service to comprise a dialog step.

### **dialog step**

A dialog step starts when a *dialog message* is received by the *UTM application*. It ends when the UTM application responds.

**dialog terminal process (Unix systems/Windows systems)**

A dialog terminal process connects a terminal of a Unix system or a Windows system with the work processes of the *UTM application*. Dialog terminal processes are started either when the user enters `utmdtp` or via the LOGIN shell. A separate dialog terminal process is required for each terminal to be connected to a UTM application.

**Distributed Lock Manager / DLM (BS2000/OSD)**

Concurrent, cross-computer file accesses can be synchronized using the Distributed Lock Manager.

DLM is a basic function of HIPLEX<sup>®</sup> MSCF.

**distributed processing**

Processing of *dialog jobs* by several different applications or the transfer of *background jobs* to another application. The higher-level protocols *LU6.1* and *OSI TP* are used for distributed processing. `openUTM-LU62` also permits distributed processing with *LU6.2* partners. A distinction is made between distributed processing with *distributed transactions* (transaction logging across different applications) and distributed processing without distributed transactions (local transaction logging only). Distributed processing is also known as server-server communication.

**distributed transaction**

*Transaction* which encompasses more than one application and is executed in several different (sub)-transactions in distributed systems.

**distributed transaction processing**

*Distributed processing with distributed transactions.*

**dynamic configuration**

Changes to the *configuration* made by the administrator. UTM objects such as *program units*, *transaction codes*, *clients*, *LU6.1 connections*, printers or *user IDs* can be added, modified or in some cases deleted from the configuration while the application is running. To do this, it is necessary to create separate *administration programs* which use the functions of the *program interface for administration*. The WinAdmin administration program can be used to do this, or separate *administration programs* must be created that utilize the functions of the *administration program interface*.

**encryption level**

The encryption level specifies if and to what extent a client message and password are to be encrypted.

**event-driven service**

This term has been superseded by *event service*.

**event exit**

Routine in an application program which is started automatically whenever certain events occur (e.g. when a process is started, when a service is terminated). Unlike *event services*, an event exit must not contain any KDCS, CPI-C or XATMI calls.

**event function**

Collective term for *event exits* and *event services*.

**event service**

*Service* started when certain events occur, e.g. when certain UTM messages are issued. The *program units* for event-driven services must contain KDCS calls.

**generation**

*Static configuration* of a *UTM application* using the UTM tool KDCDEF and creation of an application program.

**global secondary storage area**

See *secondary storage area*.

**hardcopy mode**

Operating mode of a printer connected locally to a terminal. Any message which is displayed on screen will also be sent to the printer.

**heterogeneous link**

In the case of *server-server communication*: a link between a *UTM application* and a non-UTM application, e.g. a CICS or TUXEDO application.

**Highly Integrated System Complex / HIPLEX<sup>®</sup>**

Product family for implementing an operating, load sharing and availability cluster made up of a number of BS2000 servers.

**HIPLEX<sup>®</sup> MSCF**

(MSCF = **M**ultiple **S**ystem **C**ontrol **F**acility)

Provides the infrastructure and basic functions for distributed applications with HIPLEX<sup>®</sup>.

**homogeneous link**

In the case of *server-server communication*: a link between two *UTM applications*. It is of no significance whether the applications are running on the same operating system platforms or on different platforms.

**inbound conversation (CPI-C)**

See *incoming conversation*.

**incoming conversation (CPI-C)**

A conversation in which the local CPI-C program is the *acceptor* is referred to as an incoming conversation. In the X/Open specification, the term “inbound conversation” is used synonymously with “incoming conversation”.

**initial KDCFILE**

In a *UTM cluster application*, this is the *KDCFILE* generated by *KDCDEF* and which must be copied for each node application before the node applications are started.

**initiator (CPI-C)**

The communication partners in a *conversation* are referred to as the initiator and the *acceptor*. The initiator sets up the conversation with the CPI-C calls `Initialize_Conversation` and `Allocate`.

**insert**

Field in a message text in which openUTM enters current values.

**inverse KDCDEF**

A function which uses the dynamically adapted configuration data in the *KDCFILE* to generate control statements for a *KDCDEF* run. An inverse *KDCDEF* can be started “offline” under *KDCDEF* or “online” via the *program interface for administration*.

**JDK**

Java Development Kit  
Standard development environment from Sun Microsystems for the development of Java applications.

**job**

Request for a *service* provided by a *UTM application*. The request is issued by specifying a transaction code. See also: *queued output job*, *dialog job*, *background job*, *job complex*.

**job complex**

Job complexes are used to assign *confirmation jobs* to *asynchronous jobs*. An asynchronous job within a job complex is referred to as a *basic job*.

**job-receiving service (KDCS)**

A job-receiving service is a *service* started by a *job-submitting service* of another server application.

### **job-submitting service (KDCS)**

A job-submitting service is a *service* which requests another service from a different server application (*job-receiving service*) in order to process a job.

### **KDCADM**

Standard administration program supplied with openUTM. KDCADM provides administration functions which are called with transaction codes (*administration commands*).

### **KDCDEF**

UTM tool for the *generation of UTM applications*. KDCDEF uses the configuration information in the KDCDEF control statements to create the UTM objects *KDCFILE* and the ROOT table sources for the main routine *KDCROOT*. In UTM cluster applications, KDCDEF also creates the *cluster configuration file*, the *cluster user file*, the *cluster page pool*, the *cluster GSSB file* and the *cluster ULS file*.

### **KDCFILE**

One or more files containing data required for a *UTM application* to run. The KDCFILE is created with the UTM generation tool *KDCDEF*. Among other things, it contains the *configuration* of the application.

### **KDCROOT**

Main routine of an *application program* which forms the link between the *program units* and the UTM system code. KDCROOT is linked with the *program units* to form the *application program*.

### **KDCS message area**

For KDCS calls: buffer area in which messages or data for openUTM or for the *program unit* are made available.

### **KDCS parameter area**

See *parameter area*.

### **KDCS program interface**

Universal UTM program interface compliant with the national DIN 66 265 standard and which includes some extensions. KDCS (compatible data communications interface) allows dialog services to be created, for instance, and permits the use of *message queuing* functions. In addition, KDCS provides calls for *distributed processing*.



**Kerberos**

Kerberos is a standardized network authentication protocol (RFC1510) based on encryption procedures in which no passwords are sent to the network in clear text.

**Kerberos principal**

Owner of a key.

Kerberos uses symmetrical encryption, i.e. all the keys are present at two locations, namely with the key owner (principal) and the KDC (Key Distribution Center).

**key code**

Code that represents specific access authorization or a specific role. Several key codes are grouped into a *key set*.

**key set**

Group of one or more *key codes* under a particular a name. A key set defines authorization within the framework of the authorization concept used (lock/key code concept or *access list* concept). A key set can be assigned to a *user ID*, an *LTERM partner* an (OSI) *LPAP partner*, a *service* or a *TAC queue*.

**linkage program**

See *KDCROOT*.

**local secondary storage area**

See *secondary storage area*.

**Log4j**

Log4j is part of the Apache Jakarta project. Log4j provides information for logging information (runtime information, trace records, etc.) and configuring the log output. *WS4UTM* uses the software product Log4j for trace and logging functionality.

**lock code**

Code protecting an LTERM partner or transaction code against unauthorized access. Access is only possible if the *key set* of the accesser contains the appropriate *key code* (lock/key code concept).

### **LPAP bundle**

LPAP bundles allow messages to be distributed to LPAP partners across several partner applications. If a UTM application has to exchange a very large number of messages with a partner application then load distribution may be improved by starting multiple instances of the partner application and distributing the messages across the individual instances. In an LPAP bundle, *openUTM* is responsible for distributing the messages to the partner application instances. An LPAP bundle consists of a master LPAP and multiple slave LPAPs. The slave LPAPs are assigned to the master LPAP on generation. LPAP bundles exist for both the OSI TP protocol and the LU6.1 protocol.

### **LPAP partner**

In the case of *distributed processing* via the *LU6.1* protocol, an LPAP partner for each partner application must be configured in the local application. The LPAP partner represents the partner application in the local application. During communication, the partner application is addressed by the name of the assigned LPAP partner and not by the application name or address.

### **LTERM bundle**

An LTERM bundle (connection bundle) consists of a master LTERM and multiple slave LTERMs. An LTERM bundle (connection bundle) allows you to distribute queued messages to a logical partner application evenly across multiple parallel connections.

### **LTERM group**

An LTERM group consists of one or more alias LTERMs, the group LTERMs and a primary LTERM. In an LTERM group, you assign multiple LTERMs to a connection.

### **LTERM partner**

LTERM partners must be configured in the application if you want to connect clients or printers to a *UTM application*. A client or printer can only be connected if an LTERM partner with the appropriate properties is assigned to it. This assignment is generally made during *configuration*, but can also be made dynamically using terminal pools.

### **LTERM pool**

The TPOOL statement allows you to define a pool of LTERM partners instead of issuing one LTERM and one PTERM statement for each *client*. If a client establishes a connection via an LTERM pool, an LTERM partner is assigned to it dynamically from the pool.

**LU6.1**

Device-independent data exchange protocol (industrial standard) for transaction-oriented *server-server communication*.

**LU6.1-LPAP bundle**

*LPAP bundle* for *LU6.1* partner applications.

**main process (Unix systems / Windows systems)**

Process which starts the *UTM application*. It starts the *work processes*, *printer processes*, *network processes* and the *timer process* and monitors the *UTM application*.

**main routine KDCROOT**

See *KDCROOT*.

**mapped host name**

Mapping of the partner application's UTM host name to a real host name or vice versa.

**message definition file**

The message definition file is supplied with openUTM and, by default, contains the UTM message texts in German and English together with the definitions of the message properties. Users can take this file as a basis for their own message modules.

**message destination**

Output medium for a *message*. Possible message destinations for a message from the openUTM transaction monitor include, for instance, terminals, *TS applications*, the *event service MSGTAC*, the *system log file SYSLOG* or *TAC queues*, *asynchronous TACs*, *USER queues*, *SYSOUT/SYSLST* or *stderr/stdout*.

The message destinations for the messages of the UTM tools are *SYSOUT/SYSLST* and *stderr/stdout*.

**message queue**

Queue in which specific messages are kept with transaction management until further processed. A distinction is drawn between *service-controlled queues* and *UTM-controlled queues*, depending on who monitors further processing.

**message queuing**

Message queuing (MQ) is a form of communication in which the messages are exchanged via intermediate queues rather than directly. The sender and recipient can be separated in space or time, and transfer of the message is still guaranteed, irrespective of whether a network connection is available at the time or not. In openUTM there are *UTM-controlled queues* and *service-controlled queues*.

**message router (BS2000/OSD)**

Device in a central host or a communication computer which distributes queued input messages to different *UTM applications* which can be located on different computers. The message router also allows you to work with *multiplex connections*.

**MSGTAC**

Special event service that processes messages with the message destination MSGTAC by means of a program. MSGTAC is an asynchronous service and is created by the operator of the application.

**multiplex connection (BS2000/OSD)**

Special method of connecting terminals to a *UTM application*. A multiplex connection enables several terminals to share a single transport connection.

**multi-step service (KDCS)**

*Service* carried out in a number of *dialog steps*.

**multi-step transaction**

*Transaction* which comprises more than one *processing step*.

**Network File System/Service / NFS**

Allows Unix systems to access file systems across the network.

**network process (Unix systems / Windows systems)**

A process in a *UTM application* for connection to the network.

**network selector**

The network selector identifies a service access point to the network layer of the *OSI reference model* in the local system.

**node**

Individual computer of a *cluster*.

**node application**

*UTM application* that is executed on an individual *node* as part of a *UTM cluster application*.

**node bound service**

A node bound service belonging to a user can only be continued at the node at which the user was last signed on. The following services are always node bound:

- Services that have started communications with a job receiver via LU6.1 or OSI TP and for which the job-receiving service has not yet been terminated
- Inserted services in a service stack
- Services that have completed a SESAM transaction

In addition, a user's service is node bound as long as the user is signed-on at a node application.

**normal termination of a UTM application**

Controlled termination of a *UTM application*. Among other things, this means that the administration data in the *KDCFILE* are updated. The *administrator* initiates normal termination (e.g. with *KDCSHUT N*). After a normal termination, *openUTM* carries out any subsequent start as a *cold start*.

**object identifier**

An object identifier is an identifier for objects in an OSI environment which is globally unique (i.e. throughout the world). An object identifier comprises a sequence of integers which represent a path in a tree structure.

**open terminal pool**

*Terminal pool* which is not restricted to clients of a single computer or particular type. Any client for which no computer- or type-specific terminal pool has been generated can connect to this terminal pool.

**online import**

In a *UTM cluster application*, online import refers to the import of application data from a normally terminated node application into a running node application.

**online update**

In a *UTM cluster application*, online update refers to a change to the application configuration or the application program or the use of a new UTM revision level while a *UTM cluster application* is running.

**openSM2**

The openSM2 product line offers a consistent solution for the enterprise-wide performance management of server and storage systems. openSM2 offers the acquisition of monitoring data, online monitoring and offline evaluation.

**openUTM application**

See *UTM application*.

**openUTM cluster**

From the perspective of UPIC clients, **not** from the perspective of the server: Combination of several node applications of a UTM cluster application to form one logical application that is addressed via a common symbolic destination name.

**openUTM-D**

openUTM-D (openUTM distributed) is a component of openUTM which allows *distributed processing*. openUTM-D is an integral component of openUTM.

**OSI-LPAP bundle**

*LPAP bundle* for *OSI TP* partner applications.

**OSI-LPAP partner**

OSI-LPAP partners are the addresses of the *OSI TP partners* generated in openUTM. In the case of *distributed processing* via the *OSI TP* protocol, an OSI-LPAP partner for each partner application must be configured in the local application. The OSI-LPAP partner represents the partner application in the local application. During communication, the partner application is addressed by the name of the assigned OSI-LPAP partner and not by the application name or address.

**OSI reference model**

The OSI reference model provides a framework for standardizing communications in open systems. ISO, the International Organization for Standardization, described this model in the ISO IS7498 standard. The OSI reference model divides the necessary functions for system communication into seven logical layers. These layers have clearly defined interfaces to the neighboring layers.

**OSI TP**

Communication protocol for distributed transaction processing defined by ISO. OSI TP stands for Open System Interconnection Transaction Processing.

**OSI TP partner**

Partner of the UTM application that communicates with the UTM application via the OSI TP protocol.

Examples of such partners are:

- a UTM application that communicates via OSI TP
- an application in the IBM environment (e.g. CICS) that is connected via openUTM-LU62
- an application of the OpenCPIC carrier system of the openUTM client
- applications from other TP monitors that support OSI TP

**outbound conversation (CPI-C)**

See *outgoing conversation*.

**outgoing conversation (CPI-C)**

A conversation in which the local CPI-C program is the *initiator* is referred to as an outgoing conversation. In the X/Open specification, the term “outbound conversation” is used synonymously with “outgoing conversation”.

**page pool**

Part of the *KDCFILE* in which user data is stored.

In a *standalone application* this data consists, for example, of *dialog messages*, messages sent to *message queues*, *secondary memory areas*.

In a UTM cluster application, it consists, for example, of messages to *message queues*, *TLS*.

**parameter area**

Data structure in which a program unit passes the operands required for a UTM call to openUTM.

**postselection (BS2000/OSD)**

Selection of logged UTM events from the SAT logging file which are to be evaluated. Selection is carried out using the SATUT tool.

**predialog (BS2000/OSD)**

Request from a terminal user to the data communication system to establish a *virtual connection* to the *application*. The predialog is unnecessary if the application requests the establishment of a virtual connection.

**prepare to commit (PTC)**

Specific state of a distributed transaction

Although the end of the distributed transaction has been initiated, the system waits for the partner to confirm the end of the transaction.

**preselection (BS2000/OSD)**

Definition of the UTM events which are to be logged for the *SAT audit*. Preselection is carried out with the UTM-SAT administration functions. A distinction is made between event-specific, user-specific and job-specific (TAC-specific) preselection.

**presentation selector**

The presentation selector identifies a service access point to the presentation layer of the *OSI reference model* in the local system.

**primary storage area**

Area in main memory to which the *KDCS program unit* has direct access, e.g. *standard primary working area, communication area*.

**print administration**

Functions for *print control* and the administration of *queued output jobs*, sent to a printer.

**print control**

openUTM functions for controlling print output.

**printer control LTERM**

A printer control LTERM allows a client or terminal user to connect to a UTM application. The printers assigned to the printer control LTERM can then be administered from the client program or the terminal. No administration rights are required for these functions.

**printer control terminal**

This term has been superseded by *printer control LTERM*.

**printer group (Unix systems)**

For each printer, a Unix system sets up one printer group by default that contains this one printer only. It is also possible to assign several printers to one printer group or to assign one printer to several different printer groups.

**printer pool**

Several printers assigned to the same *LTERM partner*.

**printer process (Unix systems)**

Process set up by the *main process* for outputting *asynchronous messages* to a *printer group*. The process exists as long as the printer group is connected to the *UTM application*. One printer process exists for each connected printer group.



**process**

The openUTM manuals use the term “process” as a collective term for processes (Unix systems / Windows systems) and tasks (BS2000/OSD).

**processing step**

A processing step starts with the receipt of a *dialog message* sent to the *UTM application* by a *client* or another server application. The processing step ends either when a response is sent, thus also terminating the *dialog step*, or when a dialog message is sent to a third party.

**program interface for administration**

UTM program interface which helps users to create their own *administration programs*. Among other things, the program interface for administration provides functions for *dynamic configuration*, for modifying properties and application parameters and for querying information on the configuration and the current workload of the application.

**program unit**

UTM *services* are implemented in the form of one or more program units. The program units are components of the *application program*. Depending on the employed API, they may have to contain KDCS, XATMI or CPIC calls. They can be addressed using *transaction codes*. Several different transaction codes can be assigned to a single program unit.

**queue**

See *message queue*.

**queued output job**

Queued output jobs are *asynchronous jobs* which output a message, such as a document, to a printer, a terminal or a transport system application. Queued output jobs are processed by UTM system functions exclusively, i.e. it is not necessary to create program units to process them.

**Quick Start Kit**

A sample application supplied with openUTM (Windows systems).

**redelivery**

Repeated delivery of an *asynchronous message* that could not be processed correctly because, for example, the *transaction* was rolled back or the *asynchronous service* was terminated abnormally. The message is returned to the message queue and can then be read and/or processed again.

**reentrant program**

Program whose code is not altered when it runs. In BS2000/OSD this constitutes a prerequisite for using *shared code*.

**request**

Request from a *client* or another server for a *service function*.

**requestor**

In XATMI, the term requestor refers to an application which calls a service.

**resource manager**

Resource managers (RMs) manage data resources. Database systems are examples of resource managers. openUTM, however, also provides its own resource managers for accessing message queues, local memory areas and logging files, for instance. Applications access RMs via special resource manager interfaces. In the case of database systems, this will generally be SQL and in the case of openUTM RMs, it is the KDCS interface.

**restart**

See *screen restart*,  
see *service restart*.

**RFC1006**

A protocol defined by the IETF (Internet Engineering Task Force) belonging to the TCP/IP family that implements the ISO transport services (transport class 0) based on TCP/IP.

**RSA**

Abbreviation for the inventors of the RSA encryption method (Rivest, Shamir and Adleman). This method uses a pair of keys that consists of a public key and a private key. A message is encrypted using the public key, and this message can only be decrypted using the private key. The pair of RSA keys is created by the UTM application.

**SAT audit (BS2000/OSD)**

*Audit* carried out by the SAT (Security Audit Trail) component of the BS2000 software product SECOS.

**screen restart**

If a *dialog service* is interrupted, openUTM again displays the *dialog message* of the last completed *transaction* on screen when the service restarts provided that the last transaction output a message on the screen.

**secondary storage area**

Memory area secured by transaction logging and which can be accessed by the KDCS *program unit* with special calls. Local secondary storage areas (LSSBs) are assigned to one *service*. Global secondary storage areas (GSSBs) can be accessed by all services in a *UTM application*. Other secondary storage areas include the *terminal-specific long-term storage (TLS)* and the *user-specific long-term storage (ULS)*.

**selector**

A selector identifies a service access point to services of one of the layers of the *OSI reference model* in the local system. Each selector is part of the address of the access point.

**semaphore (Unix systems / Windows systems)**

Unix systems and Windows systems resource used to control and synchronize processes.

**server**

A server is an *application* which provides *services*. The computer on which the server applications are running is often also referred to as the server.

**server-server communication**

See *distributed processing*.

**server side of a conversation (CPI-C)**

This term has been superseded by *acceptor*.

**service**

Services process the *jobs* that are sent to a server application. A service of a UTM application comprises one or more transactions. The service is called with the *service TAC*. Services can be requested by *clients* or by other servers.

**service access point**

In the OSI reference model, a layer has access to the services of the layer below at the service access point. In the local system, the service access point is identified by a *selector*. During communication, the *UTM application* links up to a service access point. A connection is established between two service access points.

**service chaining (KDCS)**

When service chaining is used, a follow-on service is started without a *dialog message* specification after a *dialog service* has completed .

### **service-controlled queue**

*Message queue* in which the calling and further processing of messages is controlled by *services*. A service must explicitly issue a KDCS call (DGET) to read the message. There are service-controlled queues in openUTM in the variants *USER queue*, *TAC queue* and *temporary queue*.

### **service restart (KDCS)**

If a service is interrupted, e.g. as a result of a terminal user signing off or a *UTM application* being terminated, openUTM carries out a *service restart*. An *asynchronous service* is restarted or execution is continued at the most recent *synchronization point*, and a *dialog service* continues execution at the most recent *synchronization point*. As far as the terminal user is concerned, the service restart for a dialog service appears as a *screen restart* provided that a dialog message was sent to the terminal user at the last synchronization point.

### **service routine**

See *program unit*.

### **service stacking (KDCS)**

A terminal user can interrupt a running *dialog service* and insert a new dialog service. When the inserted *service* has completed, the interrupted service continues.

### **service TAC (KDCS)**

Transaction code used to start a *service*.

### **session**

Communication relationship between two addressable units in the network via the SNA protocol *LU6.1*.

### **session selector**

The session selector identifies an *access point* in the local system to the services of the session layer of the *OSI reference model*.

### **shared code (BS2000/OSD)**

Code which can be shared by several different processes.

### **shared memory**

Virtual memory area which can be accessed by several different processes simultaneously.

**shared objects (Unix systems / Windows systems)**

Parts of the *application program* can be created as shared objects. These objects are linked to the application dynamically and can be replaced during live operation. Shared objects are defined with the KDCDEF statement SHARED-OBJECT.

**sign-on check**

See *system access control*.

**sign-on service (KDCS)**

Special *dialog service* for a user in which *program units* control how a user signs on to a UTM application.

**single-step service**

*Dialog service* which encompasses precisely one *dialog step*.

**single-step transaction**

*Transaction* which encompasses precisely one *dialog step*.

**SOA**

(Service-Oriented Architecture)

An SOA is a system architecture concept in which functions are implemented in the form of re-usable, technically independent, loosely coupled *services*. Services can be called independently of the underlying implementations via interfaces which may possess public and, consequently, trusted specifications. Service interaction is performed via a communication infrastructure made available for this purpose.

**SOAP**

SOAP (Simple Object Access Protocol) is a protocol used to exchange data between systems and run remote procedure calls. SOAP also makes use of the services provided by other standards, XML for the representation of the data and Internet transport and application layer protocols for message transfer.

**socket connection**

Transport system connection that uses the socket interface. The socket interface is a standard program interface for communication via TCP/IP.

**standalone application**

See *standalone UTM application*.

**standalone UTM application**

Traditional *UTM application* that is not part of a *UTM cluster application*.

**standard primary working area (KDCS)**

Area in main memory available to all KDCS *program units*. The contents of the area are either undefined or occupied with a fill character when the program unit starts execution.

**start format**

Format output to a terminal by openUTM when a user has successfully signed on to a *UTM application* (except after a *service restart* and during sign-on via the *sign-on service*).

**static configuration**

Definition of the *configuration* during generation using the UTM tool *KDCDEF*.

**SYSLOG file**

See *system log file*.

**synchronization point, consistency point**

The end of a *transaction*. At this time, all the changes made to the *application information* during the transaction are saved to prevent loss in the event of a crash and are made visible to others. Any locks set during the transaction are released.

**system access control**

A check carried out by openUTM to determine whether a certain *user ID* is authorized to work with the *UTM application*. The authorization check is not carried out if the UTM application was generated without user IDs.

**system log file**

File or file generation to which openUTM logs all UTM messages for which SYSLOG has been defined as the *message destination* during execution of a *UTM application*.

**TAC**

See *transaction code*.

**TAC queue**

*Message queue* generated explicitly by means of a KDCDEF statement. A TAC queue is a *service-controlled queue* that can be addressed from any service using the generated name.

**temporary queue**

*Message queue* created dynamically by means of a program that can be deleted again by means of a program (see *service-controlled queue*).

**terminal-specific long-term storage (KDCS)**

*Secondary storage area* assigned to an *LTERM*, *LPAP* or *OSI-PAP partner* and which is retained after the application has terminated.

**time-driven job**

*Job* which is buffered by openUTM in a *message queue* up to a specific time until it is sent to the recipient. The recipient can be an *asynchronous service* of the same application, a *TAC queue*, a partner application, a terminal or a printer. Time-driven jobs can only be issued by KDCS *program units*.

**timer process (Unix systems / Windows systems)**

Process which accepts jobs for controlling the time at which *work processes* are executed. It does this by entering them in a job list and releasing them for processing after a time period defined in the job list has elapsed.

**TNS (Unix systems / Windows systems)**

Abbreviation for the Transport Name Service. TNS assigns a transport selector and a transport system to an application name. The application can be reached through the transport system.

**Tomcat**

see *Apache Tomcat*

**transaction**

Processing section within a *service* for which adherence to the *ACID properties* is guaranteed. If, during the course of a transaction, changes are made to the *application information*, they are either made consistently and in their entirety or not at all (all-or-nothing rule). The end of the transaction forms a *synchronization point*.

**transaction code/TAC**

Name which can be used to identify a *program unit*. The transaction code is assigned to the program unit during *static* or *dynamic configuration*. It is also possible to assign more than one transaction code to a program unit.

**transaction rate**

Number of *transactions* successfully executed per unit of time.

**transfer syntax**

With *OSI TP*, the data to be transferred between two computer systems is converted from the local format into transfer syntax. Transfer syntax describes the data in a neutral format which can be interpreted by all the partners involved. An *Object Identifier* must be assigned to each transfer syntax.

**transport selector**

The transport selector identifies a service access point to the transport layer of the *OSI reference model* in the local system.

**transport system application**

Application which is based directly on the transport system interface (e.g. CMX or socket). When transport system applications are connected, the partner type APPLI or SOCKET must be specified during *configuration*. A transport system application cannot be integrated in a *distributed transaction*.

**TS application**

See *transport system application*.

**typed buffer (XATMI)**

Buffer for exchanging typed and structured data between communication partners. Typed buffers ensure that the structure of the exchanged data is known to both partners implicitly.

**UPIC**

Carrier system for openUTM clients. UPIC stands for Universal Programming Interface for Communication.

**UPIC client**

The designation for openUTM clients with the UPIC carrier system.

**user exit**

This term has been superseded by *event exit*.

**user ID**

Identifier for a user defined in the *configuration* for the *UTM application* (with an optional password for *system access control*) and to whom special data access rights (*system access control*) have been assigned. A terminal user must specify this ID (and any password which has been assigned) when signing on to the UTM application.

For other clients, the specification of a user ID is optional, see also *connection user ID*.

UTM applications can also be generated without user IDs.

**user log file**

File or file generation to which users write variable-length records with the KDCS LPUT call. The data from the KB header of the *KDCS communication area* is prefixed to every record. The user log file is subject to transaction management by openUTM.



**USER queue**

*Message queue* made available to every user ID by openUTM. A USER queue is a *service-controlled queue* and is always assigned to the relevant user ID. You can restrict the access of other UTM users to your own USER queue.

**user-specific long-term storage**

*Secondary storage area* assigned to a *user ID*, a *session* or an *association* and which is retained after the application has terminated.

**USLOG file**

See *user log file*.

**UTM application**

A UTM application provides *services* which process jobs from *clients* or other applications. openUTM is responsible for transaction logging and for managing the communication and system resources. From a technical point of view, a UTM application is a process group which forms a logical server unit at runtime.

**UTM cluster application**

*UTM application* that has been generated for use on a cluster and that can be viewed logically as a **single** application.

In physical terms, a UTM cluster application is made up of several identically generated UTM applications running on the individual cluster *nodes*.

**UTM cluster files**

Blanket term for all the files that are required for the execution of a UTM cluster application. This includes the following files:

- *Cluster configuration file*
- *Cluster user file*
- Files belonging to the *cluster page pool*
- *Cluster GSSB file*
- *Cluster ULS file*
- Files belonging to the *cluster administration journal*\*
- *Cluster lock file*\*
- Lock file for start serialization\* (only in Unix systems and Windows systems)

The files indicated by \* are created when the first node application is started. All the other files are created on generation using KDCDEF.

**UTM-controlled queue**

Message queues in which the calling and further processing of messages is entirely under the control of openUTM. See also *asynchronous job*, *background job* and *asynchronous message*.

### UTM-D

See *openUTM-D*.

### UTM-F

UTM applications can be generated as UTM-F applications (UTM fast). In the case of UTM-F applications, input from and output to hard disk is avoided in order to increase performance. This affects input and output which *UTM-S* uses to save user data and transaction data. Only changes to the administration data are saved.

In UTM cluster applications that are generated as UTM-F applications (APPLI-MODE=FAST), application data that is valid throughout the cluster is also saved. In this case, GSSB and ULS data is treated in exactly the same way as in UTM cluster applications generated with UTM-S. However, service data relating to users with RESTART=YES is written only when the relevant user signs off and not at the end of each transaction.

### UTM message

Messages are issued to *UTM message destinations* by the openUTM transaction monitor or by UTM tools (such as *KDCDEF*). A message comprises a message number and a message text, which can contain *inserts* with current values. Depending on the message destination, either the entire message is output or only certain parts of the message, such as the inserts).

### UTM page

A UTM page is a unit of storage with a size of either 2Kb or 4Kb. In *standalone UTM applications*, the size of a UTM page on generation of the UTM application can be set to 2K or 4K. The size of a UTM page in a *UTM cluster application* is always 4K. The *page pool* and the restart area for the *KDCFILE* and *UTM cluster files* are divided into units of the size of a UTM page.

### utmpath (Unix systems / Windows systems)

The directory under which the openUTM components are installed is referred to as *utmpath* in this manual.

To ensure that openUTM runs correctly, the environment variable *UTMPATH* must be set to the value of *utmpath*. On Unix systems, you must set *UTMPATH* before a UTM application is started. On Windows systems, *UTMPATH* is set on installation.

### UTM-S

In the case of UTM-S applications, openUTM saves all user data as well as the administration data beyond the end of an application and any system crash which may occur. In addition, UTM-S guarantees the security and consistency of the application data in the event of any malfunction. UTM applications are usually generated as UTM-S applications (UTM secure).

**UTM SAT administration (BS2000/OSD)**

UTM-SAT administration functions control which UTM events relevant to security which occur during operation of a *UTM application* are to be logged by *SAT*. Special authorization is required for UTM-SAT administration.

**UTM terminal**

This term has been superseded by *LTERM partner*.

**virtual connection**

Assignment of two communication partners.

**warm start**

Start of a *UTM-S* application after it has terminated abnormally. The *application information* is reset to the most recent consistent state. Interrupted *dialog services* are rolled back to the most recent *synchronization point*, allowing processing to be resumed in a consistent state from this point (*service restart*). Interrupted *asynchronous services* are rolled back and restarted or restarted at the most recent *synchronization point*.

For *UTM-F* applications, only configuration data which has been dynamically changed is rolled back to the most recent consistent state after a restart due to a preceding abnormal termination.

In UTM cluster applications, the global locks applied to GSSB and ULS on abnormal termination of this node application are released. In addition, users who were signed on at this node application when the abnormal termination occurred are signed off.

**Web service**

Application which runs on a Web server and is (publicly) available via a standardized, programmable interface. Web services technology makes it possible to make UTM program units available for modern Web client applications independently of the programming language in which they were developed.

**work process (Unix systems / Windows systems)**

A process within which the *services* of a *UTM application* run.

**WS4UTM**

WS4UTM (**Web**Services for open**UTM**) provides you with a convenient way of making a service of a UTM application available as a Web service.

### **XATMI**

XATMI (X/Open Application Transaction Manager Interface) is a program interface standardized by X/Open for program-program communication in open networks.

The XATMI interface implemented in openUTM complies with X/Open's XATMI CAE Specification. The interface is available in COBOL and C. In openUTM, XATMI can communicate via the OSI TP, *LU6.1* and UPIC protocols.

### **XHCS (BS2000/OSD)**

XHCS (Extended Host Code Support) is a BS2000/OSD software product providing support for international character sets.

### **XML**

XML (eXtensible Markup Language) is a metalanguage standardized by the W3C (WWW Consortium) in which the interchange formats for data and the associated information can be defined.

---

# Abbreviations

Please note: Some of the abbreviations used here derive from the German acronyms used in the original German product(s).

ACSE	Association Control Service Element
AEQ	Application Entity Qualifier
AES	Advanced Encryption Standard
AET	Application Entity Title
APT	Application Process Title
ASCII	American Standard Code for Information Interchange
ASE	Application Service Element
Axis	Apache eXtensible Interaction System
BCAM	Basic Communication Access Method
BER	Basic Encoding Rules
BLS	Binder - Loader - Starter (BS2000/OSD)
CCP	Communication Control Program
CCR	Commitment, Concurrency and Recovery
CCS	Coded Character Set
CCSN	Coded Character Set Name
CICS	Customer Information Control System
CID	Control Identification
CMX	Communication Manager in Unix Systems
COM	Component Object Model
CPI-C	Common Programming Interface for Communication
CRM	Communication Resource Manager
CRTE	Common Runtime Environment (BS2000/OSD)
DB	Database
DC	Data Communication
DCAM	Data Communication Access Method

## Abbreviations

---

DCOM	Distributed Component Object Mode
DES	Data Encryption Standard
DLM	Distributed Lock Manager (BS2000/OSD)
DMS	Data Management System
DNS	Domain Name Service
DP	Distributed Processing
DSS	Terminal (Datensichtstation)
DTD	Document Type Definition
DTP	Distributed Transaction Processing
EBCDIC	Extended Binary-Coded Decimal Interchange Code
EJB	Enterprise JavaBeans <sup>TM</sup>
FGG	File Generation Group
FHS	Format Handling System
FT	File Transfer
GSSB	Global Secondary Storage Area
HIPLEX <sup>®</sup>	Highly Integrated System Complex (BS2000/OSD)
HLL	High-Level Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IFG	Interactive Format Generator
ILCS	Inter-Language Communication Services (BS2000/OSD)
IMS	Information Management System (IBM)
IPC	Inter-Process Communication
IRV	International Reference Version
ISO	International Organization for Standardization
J2EE	Java 2 Enterprise Edition Technologie
JCA	Java Connector Architecture
JDK	Java Development Kit
JEE5	Java Enterprise Edition 5.0
KA	KDCS Application Area
KB	Communication Area
KBPRG	KB Program Area

KDCS	Compatible Data Communication Interface
KTA	KDCS Task Area
LAN	Local Area Network
LCF	Local Configuration File
LLM	Link and Load Module (BS2000/OSD)
LSSB	Local Secondary Storage Area
LU	Logical Unit
MIGRAT	Migration Program
MQ	Message Queuing
MSCF	Multiple System Control Facility (BS2000/OSD)
NB	Message Area
NEA	Network Architecture for TRANSDATA Systems
NFS	Network File System/Service
NLS	Native Language Support
OCX	OLE Control Extension
OLTP	Online Transaction Processing
OML	Object Module Library
OSI	Open System Interconnection
OSI TP	Open System Interconnection Transaction Processing
OSS	OSI Session Service
PCMX	Portable Communication Manager
PDN	Program System for Remote Data Processing and Network Control
PID	Process Identification
PIN	Personal Identification Number
PLU	Primary Logical Unit
PTC	Prepare to commit
RAV	Computer Center Accounting Procedure
RDF	Resource Definition File
RM	Resource Manager
RSA	Encryption algorithm according to Rivest, Shamir, Adleman
RSO	Remote SPOOL Output (BS2000/OSD)
RTS	Runtime System
SAT	Security Audit Trail (BS2000/OSD)

## Abbreviations

---

SECOS	Security Control System
SGML	Standard Generalized Markup Language
SLU	Secondary Logical Unit
SM2	Software Monitor 2 (BS2000/OSD)
SNA	Systems Network Architecture
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SPAB	Standard Primary Working Area
SQL	Structured Query Language
SSB	Secondary Storage Area
SSO	Single Sign-On
TAC	Transaction Code
TCEP	Transport Connection End Point
TCP/IP	Transport Control Protocol / Internet Protocol
TIAM	Terminal Interactive Access Method
TLS	Terminal-Specific Long-Term Storage
TM	Transaction Manager
TNS	Transport Name Service
TP	Transaction Processing (Transaction Mode)
TPR	Privileged Function State in BS2000/OSD (Task Privileged)
TPSU	Transaction Protocol Service User
TSAP	Transport Service Access Point
TSN	Task Sequence Number
TU	Non-Privileged Function State in BS2000/OSD (Task User)
TX	Transaction Demarcation (X/Open)
UDDI	Universal Description, Discovery and Integration
UDS	Universal Database System
UDT	Unstructured Data Transfer
ULS	User-Specific Long-Term Storage
UPIC	Universal Programming Interface for Communication
USP	UTM Socket Protocol
UTM	Universal Transaction Monitor
UTM-D	UTM Variant for Distributed Processing in BS2000



UTM-F	UTM Fast Variant
UTM-S	UTM Secure Variant
UTM-XML	openUTM XML Interface
VGID	Service ID
VTSU	Virtual Terminal Support
WAN	Wide Area Network
WS4UTM	Web-Services for openUTM
WSDD	Web Service Deployment Descriptor
WSDL	Web Services Description Language
XA	X/Open Access Interface (X/Open interface for access to the resource manager)
XAP	X/OPEN ACSE/Presentation programming interface
XAP-TP	X/OPEN ACSE/Presentation programming interface Transaction Processing extension
XATMI	X/Open Application Transaction Manager Interface
XCS	Cross Coupled System
XHCS	eXtended Host Code Support
XML	eXtensible Markup Language



---

## Related publications



PDF files of all openUTM manuals are included on the Enterprise DVD with open platforms and on the openUTM WinAdmin DVD (for BS2000/OSD).

All manuals are available as online manuals, see <http://manuals.ts.fujitsu.com>.

### openUTM documentation

**openUTM**  
**Concepts and Functions**  
User Guide

**openUTM**  
**Generating Applications**  
User Guide

**openUTM**  
**Using openUTM Applications under BS2000/OSD**  
User Guide

**openUTM**  
**Using openUTM Applications under Unix Systems and Windows Systems**  
User Guide

**openUTM**  
**Administering Applications**  
User Guide

**openUTM**  
**Messages, Debugging and Diagnostics in BS2000/OSD**  
User Guide

**openUTM**  
**Messages, Debugging and Diagnostics in Unix Systems and Windows Systems**  
User Guide

**openUTM** (BS2000/OSD, Unix systems, Windows NT)  
**Creating Applications with X/Open Interfaces**  
Core Manual

**openUTM**  
**XML for openUTM**

**openUTM Client** (Unix systems)  
**for the OpenCPIC Carrier System**  
**Client-Server Communication with openUTM**  
User Guide

**openUTM Client**  
**for the UPIC Carrier System**  
**Client-Server Communication with openUTM**  
User Guide

**openUTM WinAdmin**  
**Graphical Administration Workstation for openUTM**  
Online description and online help system

**openUTM, openUTM-LU62**  
**Distributed Transaction Processing**  
**between openUTM and CICS, IMS and LU6.2 Applications**  
User Guide

**openUTM** (BS2000/OSD)  
**Programming Applications with KDCS for Assembler**  
Supplement to Core Manual

**openUTM** (BS2000/OSD)  
**Programming Applications with KDCS for Fortran**  
Supplement to Core Manual

**openUTM** (BS2000/OSD)  
**Programming Applications with KDCS for Pascal-XT**  
Supplement to Core Manual

**openUTM** (BS2000/OSD)  
**Programming Applications with KDCS for PL/I**  
Supplement to Core Manual

**WS4UTM** (Unix systems and Windows systems)  
**WebServices for openUTM**

**openUTM**  
**Master Index**

## Documentation for the openSEAS product environment

### **BeanConnect**

User Guide

### **JConnect**

#### **Connecting Java Clients to openUTM**

User documentation and Java docs

### **WebTransactions**

#### **Concepts and Functions**

### **WebTransactions**

#### **Template Language**

### **WebTransactions**

#### **Web Access to openUTM Applications via UPIC**

### **WebTransactions**

#### **Web Access to MVS Applications**

### **WebTransactions**

#### **Web Access to OSD Applications**

## Documentation for the BS2000/OSD environment



Most of these manuals are available in printed form which must be paid and ordered separately at <http://manualshop.ts.fujitsu.com>

**AID (BS2000/OSD)**  
**Advanced Interactive Debugger**  
**Core Manual**  
User Guide

**BCAM (BS2000/OSD)**  
**BCAM Volume 1/2**  
User Guide

**BINDER (BS2000/OSD)**  
User Guide

**BS2000/OSD**  
**Executive Macros**  
User Guide

**BS2000/OSD-BC**  
**BLSSERV**  
**Dynamic Binder Loader / Starter**  
User Guide

**DCAM (BS2000/OSD)**  
**COBOL Calls**  
User Guide

**DCAM (BS2000/OSD)**  
**Macros**  
User Guide

**DCAM (BS2000/OSD)**  
**Program Interfaces**  
Description

**FHS (BS2000/OSD)**  
**Format Handling System for openUTM, TIAM, DCAM**  
User Guide

**IFG for FHS**  
User Guide

**FHS-DOORS** (BS2000/OSD,MS-Windows)  
**Graphical Interface for BS2000/OSD Applications**  
User Guide

**HIPLEX AF** (BS2000/OSD)  
**High-Availability of Applications in BS2000/OSD**  
Product Manual

**HIPLEX MSCF** (BS2000/OSD)  
**BS2000 Processor Networks**  
User Guide

**IMON** (BS2000/OSD)  
**Installation Monitor**  
User Guide

**MT9750** (MS Windows)  
**9750 Emulation under Windows**  
Product Manual

**OMNIS/OMNIS-MENU** (BS2000/OSD)  
**Functions and Commands**  
User Guide

**OMNIS/OMNIS-MENU** (BS2000)  
**Administration and Programming**  
User Guide

**OMNIS-MENU** (BS2000/OSD)  
User Guide

**OSS** (BS2000/OSD)  
**OSI Session Service**  
User Guide

**RSO** (BS2000/OSD)  
**Remote SPOOL Output**  
User Guide

**SECOS** (BS2000/OSD)  
**Security Control System**  
User Guide



**SECOS** (BS2000/OSD)  
**Security Control System**  
Ready Reference

**SESAM/SQL** (BS2000/OSD)  
**Database Operation**  
User Guide

**openSM2** (BS2000/OSD)  
**Software Monitor**  
Volume 1: Administration and Operation

**TIAM** (BS2000/OSD)  
User Guide

**UDS/SQL** (BS2000/OSD)  
**Database Operation**  
User Guide

**Unicode in BS2000/OSD**  
Introduction

**VTSU** (BS2000/OSD)  
**Virtual Terminal Support**  
User Guide

**XHCS** (BS2000/OSD)  
**8-Bit Code and Unicode Support in BS2000/OSD**  
User Guide

## Documentation for the Unix system environment

**CMX V6.0 (Solaris)**  
**Operation and Administration**  
User Guide

**CMX V6.0 (Unix systems)**  
**Operation and Administration**  
User Guide

**CMX V6.0**  
Programming CMX Applications  
Programming Guide

**OSS (UNIX)**  
**OSI Session Service**  
User Guide

PRIMECLUSTER<sup>TM</sup>  
**Concepts Guide (Solaris, Linux)**

**openSM2**  
The documentation of openSM2 is provided in the form of detailed online help systems, which are delivered with the product.

## Other publications

**CPI-C (X/Open)**

Distributed Transaction Processing  
X/Open CAE Specification, Version 2  
ISBN 1 85912 135 7

**Reference Model Version 2 (X/Open)**

Distributed Transaction Processing  
X/Open Guide  
ISBN 1 85912 019 9

**TX (Transaction Demarcation) (X/Open)**

Distributed Transaction Processing  
X/Open CAE Specification  
ISBN 1 85912 094 6

**XTAMI (X/Open)**

Distributed Transaction Processing  
X/Open CAE Specification  
ISBN 1 85912 130 6

**XML**

W3C specification (www consortium)  
Web page: <http://www.w3.org/XML>



---

# Index

#format 109  
\$LANG  
    read with INFO LO 295  
\*format 109  
+format 109

64-bit applications  
    COBOL 558

## A

abort dialogue 219  
abstract syntax 361  
address assistant 506  
addressing 122, 208  
    asynchronous service 122, 208  
    dialog service 122, 208  
    double-step 122, 208  
    single-step 122, 208  
addressing aid 556  
administration  
    asynchronous 475  
    asynchronous jobs 227  
    message queue 95  
    printers and printouts 367  
administration journal 616  
ANNOAMSG 281  
appl\_info 316  
application name  
    reading by event exit 449  
    request 297  
application program 30  
application start 286  
applnm 318

APRO 122, 208  
    C/C++ example 514  
    COBOL example 569  
AREA 87  
    C/C++ 489  
    COBOL 546  
as\_day 292  
as\_doy 292  
as\_dt\_day 316  
as\_dt\_doy 317  
as\_dt\_month 316  
as\_dt\_year 316  
as\_hour 292  
as\_min 292  
as\_mon 292  
as\_season 317  
as\_sec 292  
as\_tm\_hour 317  
as\_tm\_minute 317  
as\_tm\_second 317  
as\_year 292  
ASN1 compiler 361  
ASSOCIATION name 313  
asynchronous administration 475  
asynchronous job 50, 190  
    generate 272  
    in distributed processing 190  
    parallel 265  
    to remote dialog service 192  
asynchronous message 50  
    read 266  
    write 244  
    writing 272  
asynchronous processing 50  
asynchronous program unit 53

- asynchronous service 34
  - addressing 208
  - issuing jobs 57
  - multiple program units 56
  - structure 53
- ATAC job, see background job
- auto 39
- automatic screen restart 119
- availability of card reader 100
- B**
- background job 52, 244
- BADTACS 448, 472, 588
  - C/C++ example 531
  - COBOL example 588
- basic format 112
- basic functions (OSI TP) 150, 212
- basic job 58, 258
- bcapnm 318
- BEGIN WORK 105
- BINDER 505
- bottom-up strategy 137
- browse in message queue 64
  - DGET call 233
- byte stream 196
- C**
- C++ program unit
  - compile 505
- C/C++ 485
  - BS2000/OSD specific characteristics 505
  - data structures 494
  - error handling 503
  - example 510
  - Unix system specific characteristics 507
  - Windows specific characteristics 508
- C/C++ macro 497
  - names 498
  - parameters 498
  - statement follow-up 502
- ca\_hdr 310
  - field names 604
- ca\_rti
  - field names 605
- card identifier 311
- card information
  - read 291
- card reader 99, 100
  - data input 100
- CCSN 319
- chained transactions 148
- chaining services 47
- change
  - format 110
  - password 417
- character array 498
- character set 319
- checking availability 100
- client protocol
  - indicator 311
- client\_enclev 321
- cluster administration journal 616
- cob32 558
- cob64 558
- COBDIR 558
- COBOL 543
  - BS2000/OSD specific features 554
  - example 563
  - KDCS call 552
  - Unix system specific features 558
  - Windows specific characteristics 560
- code conversion 608
  - socket applications 608
- command section
  - C/C++ 496
  - COBOL 551
- commit 148
- commit functions (OSI TP) 212
- COMMIT WORK 105
- communication area 80, 310
  - C/C++ 488
  - field names 604
- communication partner 92
- complex ID 58
- confirmation 352
  - handshake 148
- confirmation job 58
  - creating 258

- connecting
  - with database systems 102
- control code 117
- control field 113, 454
- conv\_enclev 322
- conversion tables 608
- conversion, lowercase letter 345
- convtac\_enclev 322
- CTRL 219
- curr\_ccs 319
- cursor 107
  - positioning 109
- D**
- DADM 224
- data declaration 488
- data structure
  - C/C++ 494
  - COBOL 549
  - version number 316, 425
- data that is local to the node 74
- data transfer phase 154
- DATABASE 102
- database error 106
- database system 101
  - connecting 102
  - coordinating multiple 101
  - error processing 106
- database transaction 103
- DATA-PERMITTED 148
- date and time 286
  - request 292
- dattim\_info 316
- DB transaction 103
- dead letter queue 62
- DEBUG function
  - C/C++ 501
- declaring areas
  - C/C++ 489
  - COBOL 546
- delete
  - secondary storage area 438
  - temporary queue 404
- destructor 496
- dev\_cap 319
- DGET 233
- dialog
  - distributed 124
  - LU6.1 (example) 143
  - OSI TP (example) 164
  - via LU6.1 132
  - via OSI TP 147
- dialog confirmation (OSI TP) 352
- dialog message
  - read 334
  - write 351
- dialog program unit
  - structure 37
- dialog service
  - structure 40
- dialog step 32
- dialogue (OSI TP) 147
- different field names 603
- DIN 66 265 204
- distributed processing 121
  - asynchronous message 190
  - controlling communication 124
  - dialog message 124
  - LU6.1 (example) 143
  - MGET 348
  - OSI TP (example) 164
  - via LU6.1 132
  - via OSI TP 147
- documentation
  - summary 14
- double-step addressing 208
- DPUT 244
  - C/C++ example 512
  - COBOL example 566
  - influence of generation parameters 255
  - job complex 258
  - order of the calls 265
  - with distributed processing 256
  - without job complex 245
- DPUTLIMIT1 255
- DPUTLIMIT2 255
- dynamic loading 540

### E

EDIT 118  
edit profile 118  
encv\_info 316  
end of service 154, 155  
    requesting 150  
end of transaction 154, 155  
    requesting 150  
endta 320  
environment variable  
    COBOL in Unix systems 558  
error  
    with connected database 106  
error handling  
    after service restart 128  
    by program unit 125  
    programming (C/C++) 503  
    with connected database 106  
error routine  
    programming 89  
ESQL program unit 105  
event exit  
    C/C++ 502  
    COBOL 553  
    FORMAT 461  
    INPUT 449  
    INPUT, C/C++ example 516  
    INPUT, COBOL example 570  
    SHUT 458  
    SHUT, C/C++ example 533  
    SHUT, COBOL example 584  
    START 457  
    START, C/C++ example 533  
    START, COBOL example 584  
    VORGANG 459  
event functions 447  
event service 472  
    BADTACS 472  
    BADTACS, C/C++ example 531  
    BADTACS, COBOL example 588  
    MSGTAC 473  
    MSGTAC, C/C++ example 519  
    MSGTAC, COBOL example 572  
    SIGNON 477

event-driven service see event service  
exit 485  
EXIT PROGRAM 543  
external 39

### F

FGET 266  
FHS 108  
field name  
    differences C/C++ - COBOL 603  
fill character  
    communication area 81  
    SPAB 78  
flag  
    message segment at socket 199  
fork() 508  
form 107  
FORMAT 447, 461  
-format 109  
format 107  
format identifier 108  
format name 462, 505, 555  
format type 109  
formatting control area 465  
formatting system 108  
formatting user area 464  
formfeed 98  
FORMIO area 116  
FORMUSR area 116  
FPUT 272  
    influence of generation parameters 281  
    with distributed processing 281  
fuchn 319  
fucom 319  
fuhsh 319  
functional unit 147  
    chained transactions 148  
    commit 148  
    dialogue 147  
    handshake 148  
    polarized control 147  
    recovery 149  
fupol 319



**G**

gen\_nb\_lth 316  
 gen\_spab\_lth 316  
 generation  
   C/C++ example 537  
 global secondary storage area, see GSSB  
 GSSB 83, 432  
   unlock 443  
 GTDA 282

**H**

handshake (OSI TP) 148, 212  
   confirmation 339  
   request 339  
 hardcopy mode 94  
 heterogeneous coupling via OSI TP 162  
 heuristic decision 153  
 hostm 318  
 HP-UX 13

**I**

ID card  
   signing on 99  
 ID card reader, see card reader  
 identifier of the communication partner 311  
 if\_ver 316  
 if\_version 425  
 IFG 108, 505, 555  
 ILCS  
   event handling 470  
 indicator  
   client protocol 311  
 INFO 286  
 INFO CD 291  
 INFO CK 298  
 INFO DT 292  
 INFO LO 293, 295  
   BS2000/OSD 293  
   Unix system 295  
 INFO PC 296  
 INFO SI 297  
 information on reset 311

information, request  
   INFO 286  
   INIT PU 303  
 INFORMIX 102  
 INIT 302  
   with distributed processing 313  
 INIT PU  
   structure of the message area 316, 425  
 initiate  
   program unit 302  
 INPUT 447, 449  
   C example 516  
   COBOL example 570  
   errors 456  
   generation notes 456  
   second parameter 454  
 input  
   partial formats 114  
 input messages  
   socket partners 196  
 input/output indicator 465  
 inputmsg\_enclev 322  
 IUTMDB 101  
 ivariant 318  
 iversio 318

**J**

job  
   asynchronous 51  
   background 51  
   output 52  
 job complex 58, 258, 329  
   C/C++ example 513  
   COBOL example 567  
   define 329  
   MCOM 329  
 job ID 224  
   ascertain 232  
 JOB VARIABLE LINK 501  
 job-receiving service  
   INIT 313  
   MPUT 362  
   multiple 124

job-submitting service

INIT 313

MPUT 362

PGWT 393

## K

K071 106

KB 310

KB header 80, 310

field names 604

KB program area 80, 310, 312

adjust size 302

maximum length 311

KB return area 80

field names 605

KB, see communication area

kc\_pa

field names 606

KCACKCID 370

KCADAY 316, 425

KCADOY 317

KCAHOUR 317, 425

KCALARM 111, 117

KCAMIN 317, 425

KCAMONTH 316, 425

KCAPCCSN 294

KCAPLANG 294, 295

KCAPNLSL 295

KCAPPL 316

KCAPPLNM 297, 318

kcapro.h 209, 494

KCAPROA 549

KCAPROC 209

KCAPTERR 294, 295

KCASEAS 317

KCASEC 317, 425

kcat.h 494

KCATC 549

KCAUSWEIS 311, 313

KCAUSWEIS field 100

KCAYEAR 316

KCBCAPNM 297, 318

KCBENID 310, 313

kcca.h 494

KCCARD 117

kccard 100, 311

KCCCSNO 294

kccf.h 454, 494

KCCFC 454, 549

KCCFCFLD 455

KCCFCREM 455

KCCFFLD 455

KCCFFNAM 455

KCCFLOFL 455

KCCFNOCF 455

KCCFREM 455

KCCFS 455

KCCCLIENT 321

KCCLNODE 425

KCCNVTAC 322

KCCON 371

KCCONV 322

KCCCP 311

KCCSCURR 319

kccv\_day 310

kccv\_doy 310

kccv\_hour 310

kccv\_minute 310

kccv\_month 310

kccv\_second 310

kccv\_status 310

kccv\_tac 310

kccv\_year 310

kccv\_year4 311

kcdad.h 225, 231, 494

KCDADC 225, 231, 549

KCDADPID 231

KCDAGDOY 231

KCDAGHR 231

KCDAGMIN 231

KCDAGSEC 231

KCDAGTIM 231

KCDAGUS 231

KCDANMSG 231

KCDAPMSG 231

KCDASDOY 231

KCDASHR 231

KCDASMIN 231

KCDASSEC 231  
KCDASTIM 231  
KCDATAK 292  
KCDATAS 292  
KCDATE 316  
KCDEFCCS 294  
KCDEVCAP 319  
kcdf.h 111, 494  
KCDFC 111, 549  
KCDPMSGs 371  
KCDPUTID 370  
KCDSTA 311  
KCENCR 316, 320  
KCENDTA 150, 320  
KCERAS 111  
KCEXTEND 117  
KCFPMSGs 371  
KCFUCHN 213, 319  
KCFUCOM 213, 319  
KCFUHS 213, 319  
KCFUPOL 212, 319  
KCGENDOY 370  
KCGENHR 370  
KCGENMIN 370  
KCGENSEC 370  
KCGENTIM 370  
KCGENUID 370  
KCGNB 316  
KCGPAB 316, 425  
KCHOSTNM 297, 318  
KCHSET1 294  
KCHSTA 311  
KCICCD 453  
KCICFINF 452  
KCICUT 453  
KCICVST 452  
KCICVTAC 452  
KCIERRCD 453  
KCIFCH 451  
KCIFKEY 452  
kcifn 451  
KCIKKEY 452  
KCILTERM 452  
KCIMF 451  
KCINCMD 453  
kcinf.h 494  
KCINFC 549  
kcini.h 316, 425, 494  
KCINIC 316, 425, 549  
kcinp.h 494  
KCINPC 549  
KCINPMSG 322  
KCINTAC 453  
KCIUSER 452  
KCIVAR 297, 318  
KCIVER 297, 318  
KCJHRAK 292  
KCJHRAS 292  
KCJHRVG 310  
KCKBC 549  
KCKBKOPF 310  
KCKNZVG 310, 313  
KCLANG 297  
KCLKBPB 311  
KCLocale 316  
KCLogTER 311, 313  
kclpa 311  
KCLSTSGN 425  
KCLTCCSN 294  
KCLTLANG 294, 295  
KCLTNLSL 295  
KCLTRMNM 371  
KCLTTERR 294, 295  
kcmac.h 495, 497  
KCMINAK 292  
KCMINAL 311  
KCMINAS 292  
KCMINVG 310  
KCMISC 323  
KCMONAK 292  
KCMONAS 292  
KCMONVG 310  
kcmmsg.h 495  
KCMMSGC 549  
KCNEGMSG 370  
KCNOdf 111  
KCOF1 150, 311  
KCOPC 549



- KCTARB [311](#), [394](#)
- KCTAST [133](#)
- KCTERMN [311](#), [313](#)
- KCTJHAK [292](#)
- KCTJHAS [292](#)
- KCTJHVG [310](#)
- KCTMZONE [317](#)
- KCUHRAK [292](#)
- KCUHRAS [292](#)
- KCUIDLTH [214](#)
- KCUIDTYP [213](#)
- KCUSCCSN [319](#)
- KCUSERID [214](#)
- kcuserid [310](#), [313](#)
- KCUSLANG [318](#)
- KCUSNLSL [318](#)
- KCUSTERR [318](#)
- KCVER [316](#), [425](#)
- KCVERS [212](#), [297](#), [318](#)
- KCVGST [133](#), [152](#), [153](#)
- KCYEARVG [311](#)
- KDCAPLI [449](#)
- KDCATTR [504](#)
- KDCDEF
  - C/C++ example [537](#)
- KDCDISP
  - internal [100](#)
- KDCDLETQ [62](#)
- KDCFOR [112](#)
- kdcinp.h [451](#)
- KDCINPC [451](#)
- KDCPADM [364](#)
- KDCROOT [30](#)
- KDCS [204](#)
  - storage areas [74](#)
- KDCS attribute [504](#)
- KDCS call [31](#)
  - APRO [208](#)
  - COBOL [552](#)
  - CTRL [219](#)
  - DADM [224](#)
  - DGET [233](#)
  - DPUT [244](#)
  - FGET [266](#)
  - FPUT [272](#)
  - GTDA [282](#)
  - INFO [286](#)
  - INIT [302](#)
  - LPUT [326](#)
  - MCOM [329](#)
  - MGET [334](#)
  - MPUT [351](#)
  - PADM [364](#)
  - PEND [372](#)
  - PGWT [385](#)
  - PTDA [395](#)
  - QCRE [399](#)
  - QREL [404](#)
  - RSET [407](#)
  - SGET [411](#)
  - SIGN [417](#)
  - SPUT [432](#)
  - SREL [438](#)
  - UNLK [443](#)
- KDCS calls [203](#)
  - C/C++ [496](#)
  - check [298](#)
  - COBOL [552](#)
  - comments on the description [207](#)
  - extensions [204](#)
  - function groups [206](#)
  - overview [204](#), [593](#)
- KDCS communication area
  - see communication area
- KDCS interface
  - characteristics [30](#)
  - MQ calls [52](#)
  - MQ calls without automatic processing [64](#)
- KDCS macro, see C/C++ macro
- KDCS parameter area, see parameter area
- KDCS storage areas [74](#)
  - locked [88](#)
- KDCS\_C\_DEBUG [501](#)
- KDCS\_SPACES [499](#)
- KDCSCDB [501](#)
- KDCSIGN [519](#)

### L

LANG variable  
  read with INFO LO 295

language environment  
  request information 293

language identifier 318  
  request with INFO LO 295

LD\_LIBRARY\_PATH 558

LEASY 102

length conflict  
  MGET 345

lifetime  
  service-controlled queue 64

line mode 117  
  extended 506, 557

link and load module 505

LINKAGE SECTION 543, 545

Linux distribution 13

LLM (link and load module) 505, 555

local classes 496

local secondary storage area, see LSSB

locale  
  changing the location of the user ID 428

locale\_info 316

logical control code 117

long messages  
  exchanging with socket partners 198

lowercase letter  
  conversion 345

LPAP name 122

LPUT 326

LSES name 313

LSSB 82, 432  
  length 82

LTAC name 122

LU6.1 132

### M

macro, see C/C++ macro

main routine KDCROOT 30

MCOM 329  
  C/C++ example 513  
  COBOL example 567

message  
  of length 0 253, 346  
  redelivery 60, 65  
  to service-controlled queue 51

message destination 473

message flow, BS2000/OSD 116

message format 466

message queuing 34, 50  
  administration 95, 224  
  DADM 224  
  DPUT 244  
  FGET 266, 399, 404  
  FPUT 272  
  MCOM 329

message segment 90  
  DPUT 254  
  flag at socket 199  
  FPUT 280  
  MPUT 360

message-oriented 196

MFHSROUT 116

MGET 334  
  C/C++ example 511  
  COBOL example 563  
  length conflict 345  
  reset message 349  
  status information 349  
  with distributed processing 348

MPUT 351  
  C/C++ example 512  
  COBOL example 565  
  with distributed processing 362

MPUT EM 363

MPUT HM 362

MSCF 619

MSGTAC 448, 473  
  C/C++ example 519  
  COBOL example 572  
  example 475

multi-step service 40

multi-step transaction 104

### N

notational conventions 27

**O**

- octet string [216](#)
- online auction
  - example [73](#)
- openUTM version
  - query [297](#)
- Oracle [102](#)
- OSI TP [147](#)
  - basic functions [150, 212](#)
  - commit functions [212](#)
  - control dialog [219](#)
  - example [164](#)
  - handshake functions [212](#)
  - heterogeneous coupling [162](#)
  - information on reset [311](#)
  - information on selected functions [311](#)
  - protocol element [147](#)
  - select function combinations [209](#)
- OSI TP example
  - more complex dialog trees [179](#)
  - multiple job receivers [177](#)
  - one job receiver [165](#)
  - terminating via CTRL AB [188](#)
- OSI-LPAP name [122](#)
- ositp\_info [316](#)
- output
  - format mode [97](#)
  - line mode [97](#)
  - partial formats [114](#)
  - printer [94](#)
- output job [51, 244, 272](#)
- output messages
  - socket partners [197](#)
- overview
  - KDCS calls [204, 593](#)
- overview information
  - reading [225](#)

**P**

- PADM [364](#)
- PADM AI
  - return [370](#)
- PADM PI
  - return [371](#)
- parameter
  - program unit (C/C++) [487](#)
  - program unit (COBOL) [551](#)
- parameter area [31](#)
  - field names [606](#)
- parameter list
  - RSO printer [248](#)
- partial format [113](#)
  - read [347](#)
  - write [360](#)
- partner identification [123](#)
- partner status [137](#)
- passing addresses [551](#)
- PEND [372](#)
  - by distributed processing (LU6.1) [134, 137, 139](#)
  - by distributed processing (OSI TP) [156](#)
  - variants [378](#)
- PEND ER
  - by distributed processing [127](#)
- PEND PS [477](#)
- PEND RS
  - by distributed processing [125](#)
- PGWT [385](#)
  - in distributed processing (OSI TP) [156](#)
  - in OSI TP service [394](#)
  - with distributed processing [393](#)
- physical input/output area [464](#)
- polarized control [147](#)
- prepare
  - end dialogue [219](#)
  - to commit [219](#)
- prepare to commit [153](#)
  - MGET [342](#)
- print job [95](#)
- print options
  - RSO printer [244, 254, 272, 280](#)
- printable string [216](#)
- printer, administration [364](#)
- printer control [95](#)
- printing
  - avoiding bottlenecks [96](#)
  - format mode [97](#)
  - line mode [97](#)

- printout
    - administration 364
  - PROCEDURE DIVISION 546
  - processing
    - multiple job receivers 124
  - processing acknowledgment (OSI TP)
    - request 351
  - processing step 32
    - for differing actions 42
    - multiple in a single program unit 46
    - multiple program units 44
  - processor name
    - request 297
  - program framework 35
  - program indicator 311, 460
  - program name
    - C/C++ 485
    - COBOL 544
  - program unit 29
    - asynchronous 53
    - C example 515
    - command section (C/C++) 496
    - command section (COBOL) 551
    - existing as LU6.1 job receiver 141
    - existing as OSI TP job receiver 160
    - existing as OSI TP job submitter 161
    - initiate 302
    - name (C/C++) 486
    - name (COBOL) 544
    - parameters (C/C++) 487
    - reentrant capability 39
    - start 286
    - structure 35
    - subprogram call 47
    - terminate 372
  - program unit run 31
  - PROGRAM-ID 544
  - programming aid 124
    - LU6.1 132
    - OSI TP 149
  - programming recommendation
    - LU6.1 137
    - OSI TP 157
  - programming rule
    - LU6.1 134
    - OSI TP with commit 154
    - OSI TP without commit 154
  - pronm 318
  - protocol element
    - TP-ABORT 147
    - TP-DEFER(END-DIALOGUE) 148
    - TP-DEFER(GRANT-CONTROL) 148
    - TP-GRANT-CONTROL 147
    - TP-HANDSHAKE 148
    - TP-HANDSHAKE-AND-GRANT-CONTROL 148
    - TP-PREPARE 148
    - TP-U-ERROR 147
  - ps\_day 292
  - ps\_doy 292
  - ps\_dt\_day 317
  - ps\_dt\_doy 317
  - ps\_dt\_month 317
  - ps\_dt\_year 317
  - ps\_hour 292
  - ps\_min 292
  - ps\_mon 292
  - ps\_season 317
  - ps\_sec 292
  - ps\_tm\_hour 317
  - ps\_tm\_minute 317
  - ps\_tm\_second 317
  - ps\_year 292
  - pseudo-conversational 34
  - PTDA 395
  - pterm\_enclev 321
  - ptrnm 318
  - PTYPE
    - APPLI 195
- ## Q
- QCRE 399
  - QREL 404
  - queue
    - service-controlled 51, 61
  - queue messages
    - lifetime 64



**R**

read  
  application name (event exit) 449  
  asynchronous message 234, 399, 404  
  asynchronous messages 266  
  card information 291  
  dialog message 334  
  from secondary storage area 411  
  from TLS 282  
  ID card information 287, 291  
  partial format 347  
  status information 349  
README files 19  
receive see read  
recovery 149  
Red Hat 13  
redelivery  
  background jobs 60  
  DGET 242  
  FGET messages 60  
  queue messages 65  
  service-controlled queues 65  
reentrant capability 39  
register 39  
remote queuing 190  
reset 407  
  programmed 125  
reset message 352  
  MGET 349  
  MPUT 352  
  read 349  
resources  
  locked 393  
restart  
  by distributed processing via OSI TP 158  
  screen 119  
restart area 119, 464  
return 485, 502  
  KDCS call 598  
rlstsgn 425  
rminpw 425  
RMXA 102  
rollback 158  
ROLLBACK WORK 105

rpsword 425  
rpwval 425  
RSET 407  
  in job-receiving service 127  
  in job-submitting service 127  
  with distributed processing 410  
RSO parameter list 244, 272  
RSO printer 95  
  print options 248, 254, 280  
rtac 425  
ruser 425

**S**

screen  
  updating 114  
screen function 360  
  format mode 111  
  line mode 117  
screen restart 119  
secondary storage area  
  delete 438  
security type 213  
send  
  processing acknowledgment (OSI TP) 351  
send authorization 151, 154  
  end-of-transaction 155  
send see write  
server/server communication 121  
service 29  
  asynchronous 34  
  chaining 47  
  concept 32  
  end of 154, 155  
  requesting end of 150  
  stacking 48  
  start 32  
  start time 310  
  structure 33, 40  
service ID 123, 310  
service indicator 313, 460  
service routine 29  
service rule 136  
service stack 49

- service status
  - LU6.1 133
  - MGET 340
  - OSI TP 152
- service TAC 32, 310
- service transaction code 32
- service-controlled queue 50, 61
  - in distributed processing 193
  - lifetime 64
  - message to 51, 244
  - reading 233
- SESAM/SQL 102
- session name 313
- session\_enclev 321
- set wait point 385
- SGET 411
- shareable code
  - C/C++ 505
  - COBOL 555
- shared objects 507, 540
  - Unix systems 559
- short 499
- SHUT 447, 458
  - C/C++ example 533
  - COBOL example 584
- SHUTDOWN 458
- SIGN 417
- SIGN CL 428
- SIGN ON 477
- SIGN ST 477
- signal handling 507
- signing on
  - via card reader 99
- SIGNON 448, 477
- sign-on service 417, 476
  - sample program 484
- single-step addressing 208
- socket connections 196
- Solaris 13
  
- SPAB 78, 312
  - C/C++ 488
- SPUT 432
- SREL 438
  
- stack height 311
- stacked conversation
  - information 296
- stacking
  - service 48
- standalone UTM application 11
- standard primary working area, see SPAB
- START 447, 457
  - C/C++ example 533
  - COBOL example 584
  - generation notes 458
- start address
  - C/C++ 486
  - COBOL 544
- start format 112
- start of a service 32
- STARTUP 457, 458
- static 39
- status information 106, 128, 313, 349
- STOP RUN 543
- storage area
  - locked 88
- strict dialog 33
- strncmp 503
- structure
  - asynchronous service 53
  - dialog service 40
- STXIT events 471
- STXIT routines 469
- subprogram call 47
- SUSE 13
- synchronization point 101
- system name
  - request 297
  
- T**
- T61 String 216
- TAC queue 62
  - message to 272
- temporary queue 63
  - creating 399
  - deleting 404
- terminal-specific long-term storage area, see TLS

- terminate
    - program unit 372
  - territory identifier 318
    - request with INFO LO 295
  - TIAM 117
  - time
    - service start 310
  - time entry
    - limiting 255
  - time\_zone 317
  - time-driven asynchronous job 244
  - TLS 84, 282, 395
    - unlock 443
  - TLS block 285
  - TP-ABORT 147
  - TP-DEFER(END-DIALOGUE) 148
  - TP-DEFER(GRANT-CONTROL) 148
  - TP-GRANT-CONTROL 147
  - TP-HANDSHAKE 148
  - TP-HANDSHAKE-AND-GRANT-CONTROL 148
  - TP-PREPARE 148
  - TP-U-ERROR 147
  - transaction
    - end of 154, 155
    - global 121
    - requesting end of 150
    - reset 407
  - transaction code 32
    - remove 346
  - transaction indicator 311
  - transaction rule 136
  - transaction status
    - LU6.1 133
    - MGET 341
    - OSI TP 153
  - transfer syntax 361
  - TRMSGLTH 116
  - TRUNCATE-LITERAL 554
  - TS applications 195
  - two-phase-commit 101
  - type of the format 108
- U**
- UDS/SQL 102
  - ULS 85, 432
    - unlock 443
  - Unix platform 13
  - UNLK 443
  - unlock
    - GSSB 443
    - TLS 443
    - ULS 443
  - UpicB.ocx 24
  - us\_ccsname 319
  - us\_lang\_id 318
  - us\_nlslang 318
  - us\_terr\_id 318
  - user exit, see event exit
  - user ID 310
  - user information 257
    - log 245
    - reading 225
  - user log file 86, 326
  - USER queue 61
  - user-defined formatting routine 462
  - user-specific location
    - change 428
  - user-specific long-term storage are, see ULS
  - USLOG file 86
  - USP 196
  - UTM application 30
    - C example 524
    - COBOL example 577
    - communication partners 92
  - UTM application program 30
  - UTM cluster application 11
    - cluster administration journal 616
  - UTM control field 113
  - UTM Socket Protocol 196
  - UTM-controlled queue 50
    - in distributed processing 190

### V

version [318](#)  
version number  
    data structure [316, 425](#)  
version number of the data structure [425](#)  
void [485](#)  
VORGANG [447, 459](#)  
VTCSET [506](#)  
VTSU [117](#)

### W

Windows system [13](#)  
WORKING-STORAGE SECTION [544](#)  
write  
    asynchronous message [244, 272](#)  
    dialog message [351](#)  
    message segment [360](#)  
    partial format [360](#)  
    reset message [352](#)  
    to log file [326](#)  
    to secondary storage area [432](#)  
    to TLS [395](#)

### X

X/Open [101](#)  
XA interface [101](#)  
XA message [106](#)