CMX V6.0

Programming Applications

Edition June 2003

Comments… Suggestions… Corrections…

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

Fax forms for sending us your comments are included in the back of the manual.

There you will also find the addresses of the relevant User Documentation Department.

Certified documentation according DIN EN ISO 9001:2000

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 9001:2000.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

This manual is printed on
paper treated with
chlorine-free bleach.

Continued ▶

Indexes

Contents

1 Preface

1.1 Brief product description

The transport access system CMX (Communication Manager UNIX) is the basic product for communication software. CMX enables communication between applications in different computer systems. It is responsible jointly with the CCPs (Communication Control Programs) for communication tasks. The CMX program interface can be used to generate application programs that can communicate with other applications irrespective of the transport system.

1.2 Target group

This manual is intended for programmers who develop TS applications for communication (TS application = Transport Service application). These applications consist of application programs implemented in C.

In order to work with CMX, you must be familiar with the operating system and be proficient in using the C programming language and the C development system. Knowledge of the principles and methods of data communications will also prove helpful, especially with regard to the OSI Reference Model as standardized in ISO 7498.

1.3 Summary of contents

Two User Guides provide a comprehensive description of the CMX product:

- CMX “Operation and Administration” for system administrators and users
- CMX “Programming Applications” for programmers of TS applications

This manual describes the CMX program interfaces, i.e. all the tools you will need in order to develop TS applications of your own.

Diagnostic aids, which include trace mechanisms for libraries and a program to decode CMX messages, are included in the “Operation and Administration” manual.

Structure of the manual

The manual is divided into two parts:

Part 1 is intended to help you get acquainted with CMX and focuses on helping the first-time user to create TS applications.

This part describes the mapping of a TS application onto the process concept of your system and the allocation of transport connections to processes of the TS application. The structure of a TS application is explained, showing how it can be divided into three communication phases and how the functions of the program interfaces are used within these phases. In addition, you will learn how you can use the Transport Name Service (TNSX) to request names and addresses from the address directory and pass them on to CMX, and how to obtain diagnostic information from CMX in the case of errors. To explain the individual programming steps, program fragments have been provided as examples.

Part 2 consists of chapters 8 and 9. Each of these chapters describes one of the CMX program interfaces. Every function call of the respective program interface is explained in detail along with its parameters. The description is arranged in alphabetical order. Each chapter begins with a summary of all the information you will need to use the functions.

The description takes into account all the various ways of connecting a computer to a network (LAN and WAN). The ways in which your own computer can be incorporated into a network will depend on the CCs (Communication Controller) and CCPs (Communication Control Program) that have been installed on your system.

The program interfaces are described independent of which operating system is used. Details specific to the operating system version are given in the Release Notice for CMX.

References to other publications

The text contains references to other publications in the form "see [n]", where n is a number. The Reference Section contains a list of the corresponding publi-cations by number, together with a brief overview of the contents.

1.4 Sample programs

CMX comes with a number of sample programs for TS applications on ICMX(L) and ICMX(NEA). The C source code for these programs is stored in the directory *opt/lib/cmx/demo* (for Reliant UNIX) or */opt/SMAW/SMAWcmx/lib/cmx/demo* (for Solaris). Here you will also find a sample script for making entries in the TS directory, as well as sample scripts for calling the programs.

You will find descriptive information concerning the purpose, compilation, and invocation of these programs in comments at the start of the C source code programs.

1.5 Readme files

Functional changes and supplements of the current product version to this manual are also included in the product-specific README files. These README files are located in the relevant product directories under *opt/readme* (Reliant UNIX) or */opt/SMAW/documents* (Solaris), provided your system administrator has installed them. The README files are displayed using an editor, and are printed on a standard printer.

2 The CMX transport access system

2.1 Communication between TS applications

Any application that wishes to exchange data with another application in some other end system requires the services of a transport system. The transport system performs all the necessary tasks to set up the connection and to transport data over the physical media (lines, computers). Applications that use the services of a transport system are called TS applications.

A TS application should be capable of setting up connections and exchanging data using different transport systems. As far as possible, the TS application should be independent of the underlying transport system used. Transport systems may differ in various respects, e.g. the size of the data unit that each transport system can handle, the format of the partner application’s transport address to be passed, and the format of the TS application’s address in the local system. For this reason, CMX provides TS applications with a uniform interface to the various transport services. This interface is the program interface ICMX(L). It provides TS applications with access to the services of transport systems that conform to the standards laid down in the OSI Reference Model for open systems. CMX is thus a transport access system.

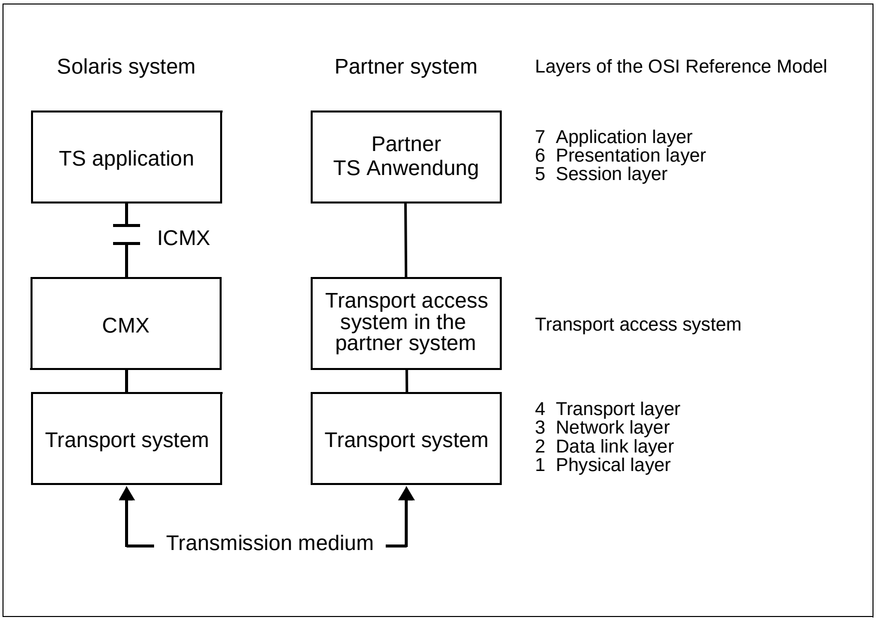


Figure 1: The CMX transport access system

A TS application that uses CMX functions can thus communicate in a uniform way with:

– other TS applications in the same computer (local communication),

– TS applications in other computers which use the functions of the transport access system CMX,

– TS applications in host computers running BS2000/OSD and using the functions of the transport access system DCAM or UTM,

– TS applications in communication computers running PDN and using the functions of the transport access system CAM,

– TS applications in systems of other vendors, assuming they conform to the standards prescribed in the OSI model or are connected via TCP/IP by means of the convergence protocol RFC1006.

For the programmer, the uniform program interface ICMX(L) means that he or she can develop TS applications independent of specific data transmission characteristics, i.e. only the ICMX(L) functions need to be programmed for communication. These functions can be used to:

- attach the TS application to CMX,
- set up transport connections to partner applications,
- send and receive data,
- control the data flow,
- disconnect transport connections,
- detach TS applications from CMX.

If a TS application is to communicate with a partner application in end systems (running BS2000/OSD or PDN) which have not yet been adapted to conform to OSI conventions and which therefore need the expanded functionality of the NEA transport services, the migration service NEABX must be used. NEABX builds on the ICMX(L) functions and is implemented via the ICMX(NEA) program interface of CMX.

TS applications that use the functions of CMX interfaces are also called CMX applications in the description below. This term is always used whenever it is necessary to make a distinction between TS applications running under ICMX and other TS applications.

2.2 The CMX program interfaces - an overview

CMX provides the programmer of TS applications with the following two function complexes:

- The functions for connection-oriented communication.

  These functions cover local services, connection handling and data exchange.They are available via the ICMX(L) program interface.

- The NEABX migration service.

  The functions of NEABX permit CMX applications to communicate with TS applications in communication computers and host computers of the TRANSDATA family that do not conform to OSI conventions.

  The NEABX migration service is available via the ICMX(NEA) program interface.

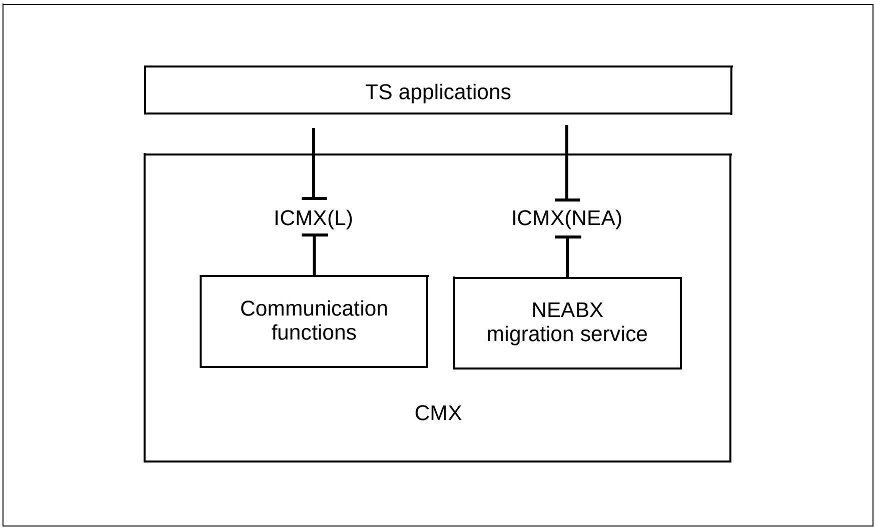


Figure 2: CMX program interfaces

The program interfaces of CMX are library interfaces, i.e. the functions of CMX are provided in the form of modules in a library. These modules are linked into the programs of the TS application.

All modules are stored in the library */usr/lib/libcmx.so* (a shared object).

2.2.1 CMX functions for communication (ICMX(L))

The program interface ICMX(L) includes all functions which are used by a TS application for communication.

The following function groups are provided at the ICMX(L) interface:

– Functions for attaching to and detaching from CMX

– Functions to establish a connection

– Functions to close down a connection

– Functions to redirect a connection

– Functions for the exchange of data

– Functions for flow control

– Functions to request information

Functions for attaching to and detaching from CMX

When a TS application attaches itself to CMX, it passes its LOCAL NAME, i.e. its own address within the local system, to CMX. Only then is the TS application addressable. After communication, the TS application must detach itself from CMX.

Functions to establish a connection

This includes functions for

– active connection setup:

  The two functions in this group are used to request a connection with a remote TS application (connection request) and to set up the connection after receipt of a positive response from the remote TS application (connection confirmation).

– passive connection: The two functions in this group serve to accept a connection setup request from a remote TS application (connection indication) and to respond to this request (connection response).

Functions to close down a connection

The two functions in this group are used to actively close down a connection (disconnection request) or to accept a disconnection request (disconnection indication).

Functions to redirect a connection

Within a TS application a connection may be passed on (redirected) to another process of the same TS application. The two functions in this group can be used to redirect a connection and to accept a connection from another process (redirect indication).

Functions for the exchange of data

This group of functions allows you to:

– send (data request) and receive (data indication) normal data.

– send (expedited data request) and receive (expedited data indication) expedited data.

  Expedited data refers to small amounts of data that can be transmitted to a communication partner with priority over the main data stream. These functions are optional.

Functions for flow control

If you currently cannot or do not wish to receive any data, you can have the data flow stopped by informing CMX. CMX will then stop signaling incoming data. The communication partner is (usually) notified and will not be permitted to send you any further data until you release the data flow. The data flow can be controlled separately for normal and expedited data (datastop, datago, xdatstop, xdatgo).

Functions to request information

This group includes functions that can be used to:

– await or fetch an event (event).

  A typical example of an event is a disconnection request from the communication partner.

– request information on errors (error).

– request information on CMX parameters (information).

– query LOCAL and GLOBAL NAMES, and TRANSPORT ADDRESSES (get local name, get name, get address).

Chapters 4 to 7 explain the use of the functions in programs of a TS application.

2.2.2 CMX functions for migration (ICMX(NEA))

The CMX functions for migration are grouped together in the NEABX migration service. The functions of NEABX are available at the ICMX(NEA) interface.

The NEABX migration service supports communication between CMX applications and TS applications in communication computers running PDN and host computers running BS2000 when such partners use functions which used to be available in the NEA transport protocols but are no longer provided in ISO transport systems conforming to ISO standard 8072. Such functions are:

- password at connection setup
- user data at connection setup exceeding 32 bytes
- message structuring with ETX/ETB
- indication of the message code used
- requesting transport acknowledgments
- sequence numbers in message exchange

This expanded functionality is provided through the NEABX protocol. The formation and interpretation of this protocol takes place through the NEABX functions. The function calls are issued in a way analogous to that for the CMX functions for communication, described in section 2.2.1.

The NEABX migration service enables an existing TS application that does not conform to OSI conventions to communicate with a CMX application. Such a TS application can therefore communicate with a CMX application without modification of its communication interface. Conversely, the NEABX migration service enables a CMX application to support the expanded functionality of such a TS application.

The migration service NEABX will be needed until the communication interfaces of these TS applications in the network are adapted to the functionality of the ISO transport system.

Criteria for deciding on use

The decision to use the NEABX migration service must be made in the CMX application before it is attached to CMX. NEABX must always be used when the desired communication partner requires functions that are provided in the NEABX protocol but are not available in a transport service conforming to ISO standard 8072.

Use of the NEABX migration service is particularly required when you wish to communicate with current BS2000 applications via DCAM, TIAM or UTM.

CMX functions with NEABX migration service

The CMX calls for migration and the process of communication are essentially the same as for the CMX functions for communication at the ICMX(L) interface. The ICMX(NEA) functions call the ICMX(L) functions internally. They can thus be classified in a similar manner to the function groups described in section 2.2.1. Chapters 4 to 7 explain how the functions are used in the programs of a TS application.

2.2.3 System and user options

The functions of CMX consist of mandatory and optional functions with mandatory and optional parameters.

For communication with partners via CMX, the mandatory functions with the mandatory parameters are always available for all transport systems.

Depending on the type of connection to the network, i.e. depending essentially on the transport system, optional functions are also available, as well as optional parameters for the mandatory functions.

Option	Optional function	Optional parameter	System option	User option
User data at connection setup	no	yes	yes	yes
User data at disconnection	no	yes	yes	yes
Expedited data	yes	yes	yes	yes
Monitoring of inactive time	no	yes	yes	yes
User data at connection setup	no	yes	yes	yes
User data at disconnection	no	yes	yes	yes
Expedited data	yes	yes	yes	yes

Table 1: CMX options

Option	**Optional function**	**Optional parameter**	**System option**	**User option**
Monitoring of inactive time	no	yes	yes	yes
Connection limit active/passive mode	no	yes	no	yes
User reference of attachment	no	yes	no	yes
User reference of connection	no	yes	no	yes

Table 1: CMX options

The system options are oriented to the functionality of the transport system. If options are used that the transport system or the communication interface of the partner application does not provide, the connection will not be established, or a disconnect indication will be issued by CMX. Given an appropriate transport system, CMX guarantees error-free execution of your CMX application.

If communication is to be error-free, the user options must also be correct, i.e. the partners must have a common understanding of how they are used.

This means that CMX does *not* compensate for the difference between the functionality expected in the TS application and that actually provided by the transport system. This applies particularly to the system options shown above.

Refer to the manuals for the individual CCP products for a description of which system options are offered by a particular transport system.

3 TS applications

This chapter outlines the characteristics of TS applications that use the functions of the CMX program interfaces.

The following points are covered in the sections of this chapter:

– Name and properties of a TS application

  Every TS application has a GLOBAL NAME, with which it can be uniquely identified within the network. To communicate with other TS applications in the network, a TS application must be addressable. For this reason, a TS application is assigned the properties TRANSPORT ADDRESS and LOCAL NAME in addition to other properties.

– Structure of a TS application

  A TS application is a C program or a system of C programs that calls CMX functions.

  This section describes what is required when writing TS application programs, how such C programs are compiled, and which libraries must be linked into the source code.

– Association between a TS application, processes, and connection

  This section deals with the question of how a TS application can be mapped onto a system’s process concept, and illustrates the association between a process and a connection.

– Threads and Multithreading

  This section gives an overview of threads and multithreading and the related connections and processes. It also lists the necessary CMX library functions and describes compiling and linking.

3.1 Names and addresses of TS applications

Every TS application has a GLOBAL NAME. This name identifies the TS application uniquely in the network, i.e. different TS applications have different GLOBAL NAMES. The GLOBAL NAME specifies which TS application is involved.

The GLOBAL NAMES of all TS applications in the local system and those of all TS applications in remote systems with which the local TS applications wish to communicate are recorded in a name and address directory. This directory is known as the TS directory. The properties of a TS application are stored in the TS directory along with its GLOBAL NAME. A property refers to any information on the communication partners that may be required by the respective transport system in order to set up a connection. The transport address of a TS application is a typical example of one of its properties.

Within a TS application, only the GLOBAL NAMES of the two communication partners are used. This makes TS applications independent of the specific addressing requirements of the transport system and of changes within the network. All that is needed is the addition or modification of the relevant properties in the TS directory. The TS application reads the properties from the TS directory with the aid of certain ICMX(L) function calls and passes them on to CMX directly (i.e unseen).

Properties must be managed and GLOBAL NAMES must be assigned by the TNSX administration. It must ensure that among the GLOBAL NAMES of all TS applications no two are the same, i.e. different TS applications must have unique GLOBAL NAMES.

For an overview of the TNSX, please refer to the relevant “CMX, Operation and Administration” manual [1] or [2].

3.1.1 The GLOBAL NAME of a TS application

The GLOBAL NAME of a TS application is a hierarchically structured name consisting of up to 5 name parts: name part[1] through name part[5]. Of these, name part[1] is the highest in the hierarchy, name part[5] the lowest. All levels of the hierarchy need not be present in a GLOBAL NAME; it is possible to omit name parts. A GLOBAL NAME can also consist of a single name part at any hierarchy level. Apart from the hierarchical order, the TNSX makes no further specifications regarding the meanings of the name parts within the GLOBAL NAME.

An application program that is being executed in more than one computer must run under a different GLOBAL NAME in each computer. The program name and the GLOBAL NAME of the TS application must not be confused. A TS application has a GLOBAL NAME that is unique within the network. This name is assigned by the network administrator, as required. Nevertheless, the TS application may functionally consist of the same programs in the various end systems.

3.1.2 Properties of a TS application

The properties of a TS application constitute all the information on a TS application that is required by the transport system in order to set up the connection and manage the actual transmission of data. Properties are assigned to the GLOBAL NAME in the TS directory. The following tables illustrate which properties can be assigned to a TS application in the local system or to the communication partners (i.e. the remote TS applications).

Some of these properties must be queried from the TNSX and passed to CMX when attaching the local TS application to CMX or setting up the connection.

Properties of a local TS application

Property	**Meaning of the property**
LOCAL NAME	This property is needed to attach the TS application to CMX in the local end system. The LOCAL NAME consists of the addresses of the TS application in the local system for the various transport systems. It is made up of a hexadecimal string with non-printing characters.
USER1 USER2 USER3	User-specific properties. Up to three user-specific properties may be assigned to each TS application in a freely-selectable format. These properties can be used to store information that is relevant for your application. They are not used by CMX or the TNSX.

Table 2: Properties of a local TS application

Properties of a remote TS application

Property	**Meaning of the property**
TRANSPORT ADDRESS	The value of this property is the communication partner’s transport address that is expected by CMX at connection setup. It is made up of a hexadecimal string with non-printing characters.
TRANSPORT SYSTEM	The value of this property is the type of transport system used for communicating with a remote communication partner. When writing a TS application you need not concern yourself with this property. CMX uses it internally.
USER1 USER2 USER3	Up to three user-specific properties may be assigned to each TS application in a freely-selectable format. These properties can be used to store information that is relevant for your application. They are not used by CMX or the TNSX.

Table 3: Properties of a remote application

The properties TRANSPORT ADDRESS and LOCAL NAME, and their meanings, are described in more detail in the following section.

3.1.3 The properties LOCAL NAME and TRANSPORT ADDRESS

Every TS application is assigned a unique Transport Service Access Point (TSAP) when it is attached to CMX. The TSAP is identified by means of the LOCAL NAME that is specified by the TS application when it attaches itself to CMX.

The TS application can access the services of the transport system via the TSAP. Which transport systems, i.e. network connections, can be accessed by the TS application will depend on the T-selectors contained in the LOCAL

NAME of the TS application. The LOCAL NAME contains one or more T-selectors. A single T-selector can be valid for multiple network connections, provided these are of the same type.

The TS application can be addressed from the network via the T-selector, since the T-selector is a component of its TRANSPORT ADDRESS for the respective network. The TRANSPORT ADDRESS provides a means of uniquely addressing the TS application in the entire network. The TRANSPORT ADDRESS of a TS application consists of the network address of the end system in which the TS application is located and the T-selector of the TS application for this network connection. The TRANSPORT ADDRESS is thus structured as follows:

TRANSPORT ADDRESS =
    end system network address + (locally unique) T-selector

The following diagram illustrates the relationship between the LOCAL NAME, TSAP, and TRANSPORT ADDRESS.

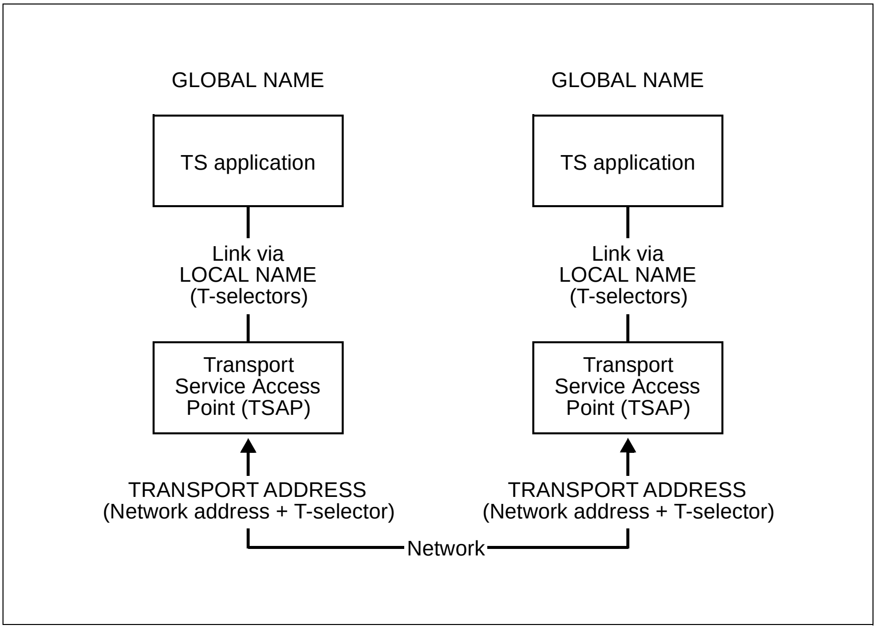

GLOBAL NAME
GLOBAL NAME
TS application
TS application
Link via LOCAL NAME (T-selectors)
Link via LOCAL NAME (T-selectors)
Transport Service Access Point (TSAP)
Transport Service Access Point (TSAP)
TRANSPORT ADDRESS (Network address + T-selector)
TRANSPORT ADDRESS (Network address + T-selector)
Network

Figure 3: TRANSPORT ADDRESS and LOCAL NAME

The *t_getloc()*, *t_getaddr()*, and *t_getname()* calls provided at the ICMX(L) interface can be used to determine the LOCAL NAME or TRANSPORT ADDRESS for a given GLOBAL NAME, and the GLOBAL NAME corresponding to a given TRANSPORT ADDRESS. In other words, all information required in ICMX(L) for a TS application can be queried from the TS directory with the help of these calls.

3.2 Structure of a TS application

A TS application is a C program or a system of C programs that call CMX functions. This chapter describes what should be observed when creating such a program.

The following example illustrates the structure of a program of this type. The specified function calls are part of the ICMX(L) interface. A program that uses the functions of ICMX(NEA) is structured analogously, except that it uses the corresponding x_...() calls instead of the t_...() calls. The calls *t_getloc()*, *t_getaddr()*, and *t_getname()* are exceptions; they can be used in both programs.

```
#include <cmx.h>
#include <tnsx.h>
 .
 .
main(argc, argv)
int argc;
char *argv[];
{
    .
    .
    /* 1st communication phase */
    t_getloc();             /* Ascertain LOCAL NAME */
    t_attach();             /* Attach to CMX */
    /* 2nd communication phase */
    t_getaddr();            /* Ascertain TRANSPORT ADDRESS */
                            /* of partner */
    t_conrq();              /* Set up connection */
    .
    .
    t_concf();              /* Accept connection */
                            /* confirmation */
    /* 3rd communication phase */
    t_datarq();             /* Send data to partner */
    .
    .
    t_datain();             /* Receive data from partner */
    .
    .
    t_disrq();              /* Close down connection */
    t_detach();             /* Detach from CMX */
    .
    .
    exit();
}
```

Header files

Every TS application program must contain an include statement for the file <*cmx.h*>. <*cmx.h*> contains the definitions of the parameters for the functions of the ICMX(L) interface.

If the TS application is to communicate via the migration interface ICMX(NEA), it must also contain an include statement for the file <*neabx.h*>. <*neabx.h*> defines all additional parameters required by the migration service.

All these files are located in the directory */usr/include*.

Permissible order for CMX function calls

TS application programs must call CMX communication functions in a certain order. The process of communication can be divided into three phases. A TS application must pass through each phase successfully before it can enter the next phase.

- 1st communication phase:
  The TS application must attach itself to CMX. Only when the TS application is known to CMX can it make use of the services of CMX. The processes carried out in this communication phase are described in chapter 5.
- 2nd communication phase:
  In this phase the TS application sets up the connection to its communication partner. During connection setup the two partners must reach an agreement as to how the subsequent exchange of data is to take place and what form the data is to have. Both partners determine, for example, whether they wish to exchange expedited data. The processes carried out in this communication phase are described in chapter 6.
- 3rd communication phase:
  In the third phase the data is exchanged between the partners. Both communication partners can send and receive data. The processes carried out in this communication phase are described in chapter 7.

This is the order in which a TS application program may call CMX functions. In addition, note that some calls may be issued only after certain responses from the other communication partner have arrived and been received by the TS application (see section “Receiving events” on page 37).

One might say that a TS application assumes various states during the course of communication. Several states are possible within each communication phase. Only certain transitions are possible between the states within a given phase and between states of different phases.

A TS application can shift from one state to the next only by calling certain CMX functions or when certain events arrive for it from the network.

Sections “States of TS applications and permissible state transitions” on page 96 and “Finite-state automata” on page 226 contain flow charts illustrating the possible states and state transitions. These flow charts are designed to simplify the development of your own TS application programs.

Communication of a TS application via ICMX(L) and ICMX(NEA)

ICMX(L) and ICMX(NEA) calls may not be mixed within the same communication phase of a TS application program. The ICMX(L) functions *t_getaddr()*, *t_getloc()*, and *t_getname()* to query names and addresses from the TS directory are exceptions; they can also be used by TS applications that make use of the migration service. Please note that a TS application can only communicate via one of the interfaces, ICMX(L) or ICMX(NEA), at a given time. If a process of a TS application wishes to communicate simultaneously in both modes, it must attach itself to CMX as two different TS applications, i.e. with two different LOCAL NAMES.

Reaching an agreement as to the form of transferred data

Two TS applications wishing to communicate with each other must also reach an agreement as to the form of the data to be transferred. Of importance here is the character set in use in each system. In Solaris systems, this is the ISO 7-bit code; in BS2000/OSD and PDN systems it is the EBCDIC code. Any necessary conversion of the data must be performed by the TS applications themselves, since the transfer through the transport systems and CMX is code-transparent.

Parameter passing and storage allocation

In TS applications parameters are passed to CMX functions as values or pointers; for options, unions are defined. All structures are declared in the header files.

In your program you must always provide all storage areas used to pass values to CMX or in which CMX is to return anything. You allocate such storage areas either at compile time (statically) or at runtime (dynamically), e.g. with *malloc()* (see the reference manual for the C Development System). In the CMX parameter structures, length fields are defined for areas of variable length. Before calling CMX, enter in these fields the lengths of the areas provided. Then, upon return, you can usually read from these fields the lengths of the data returned by CMX.

3.3 Compiling and linking TS application programs

After a C program *prog.c* for a TS application has been edited, it must be compiled, and the CMX functions from the CMX library *libcmx.so* must be linked. The C Development System is required for this purpose.

The C compiler, with included link phase, is called as follows:

```
cc -o prog prog.c ... -lcmx -lsocket -lnsl
```

Please refer to the Release Notice for possible deviations from this syntax.

3.4 TS applications, processes, connections

The two following sections describe the relationships between TS applications and processes and between processes and connections.

3.4.1 TS applications and processes

In the simplest case a TS application is a single process. However, there are additional possibilities for structuring a TS application.

A TS application can work with multiple processes, which need not be related to one another. Each individual process of a TS application must attach to CMX separately. Processes belong to the same TS application when they have attached themselves to CMX using the same LOCAL NAME. The first process to attach itself creates the TS application.

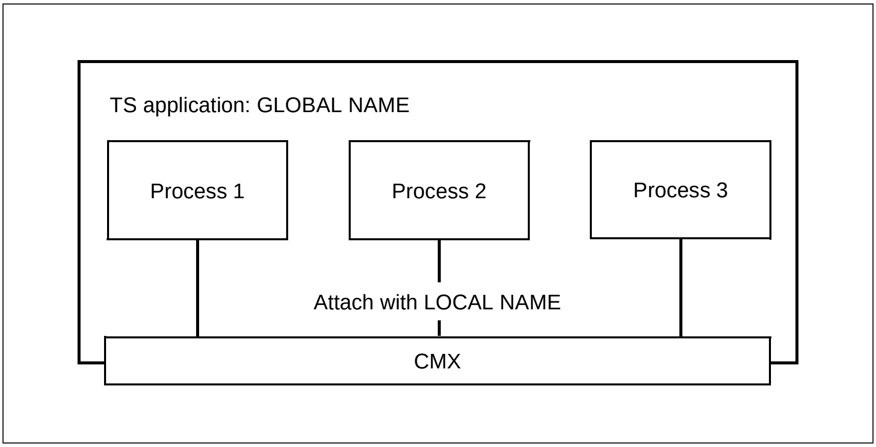


Figure 4: One TS application - multiple processes

On the other hand, one process may control multiple TS applications. To achieve this, you attach the process to CMX using different LOCAL NAMES.

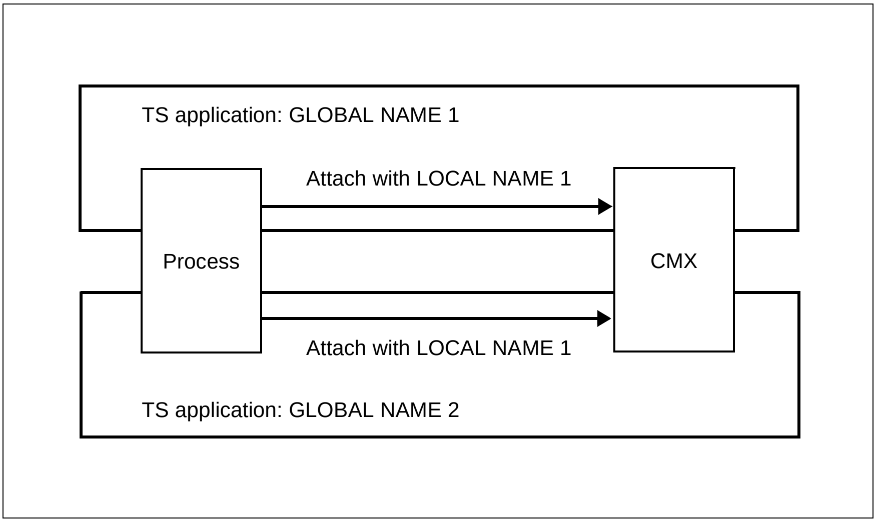


Figure 5: One process - multiple TS applications

The process distinguishes the various TS applications it controls by means of the different LOCAL NAMES or by means of a freely chosen user reference.

3.4.2 Connections and processes

The processes of a TS application can set up connections to other TS applications independently of one another, and individual processes of the TS application may maintain multiple connections simultaneously. If the process is attached to more than one TS application, the connections may also belong to different TS applications. When the connection is set up, a Transport Connection Endpoint (TCEP) is created for each connection. In other words, a single process can serve a number of TCEPs, but the same TCEP may not be simultaneously assigned to multiple processes. Each TCEP is assigned to **exactly** one process at a given time. It **cannot** be "inherited" via *fork()*.

CMX assigns each connection an identifier. This is the transport reference. This alone enables the process to address a specific connection.

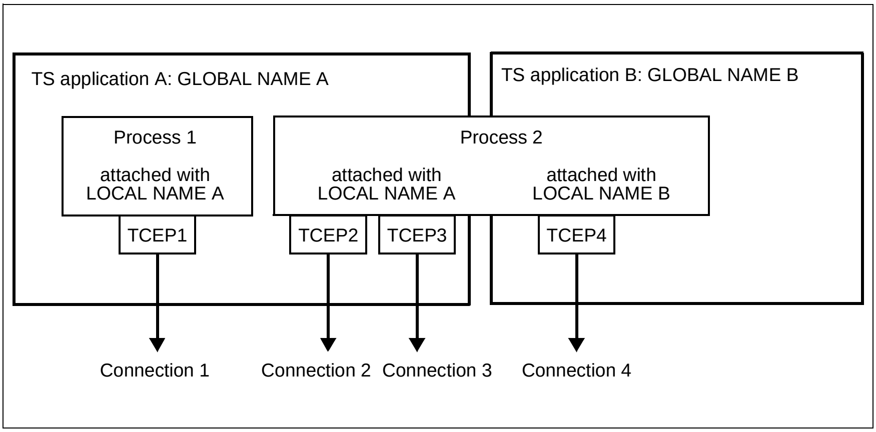

TS application A: GLOBAL NAME A
TS application B: GLOBAL NAME B
Process 1
attached with LOCAL NAME A
Process 2
attached with LOCAL NAME A
attached with LOCAL NAME B
TCEP1
TCEP2
TCEP3
TCEP4
Connection 1
Connection 2
Connection 3
Connection 4

Figure 6: Connections and processes

A process may, however, redirect a connection to another process that has attached itself in the same TS application. The connection will then no longer be recognized in the process that redirects it. In this way it is possible to handle connections to various partners in various processes. A central distribution process may, for example, receive all connections and then redirect them to appropriate subordinate processes. In the above figure, for example, process 2 could redirect connection 2 or connection 3 to process 1.

3.5 Threads and Multithreading

A thread is part of a program which is executed sequentially. A purely sequential program is described as being single-threaded. If a program consists of two or more independent parts which are executed concurrently then these parts are described as being multithreaded.

A multithreading (MT) operating system makes it possible to execute the various parts (threads) of single processes in parallel. On single processor machines, execution of these threads is pseudo-parallel whereas on multiprocessor machines the threads really are executed in parallel.

CMX V6.0 provides a library for multithreaded applications which complies with POSIX 1003.1c and ISO/IEC 9945-11 standards. To enable the use of this library, the function calls of the ICMX(L) program interface have been made multithread capable.

The ICMX(NEA) program interface is not available in a multithread version.

Connections and processes

Each thread handles its own connections but cannot access connections owned by other threads. Connections to other processes (i.e. threads in other processes) and connections to other threads in the same process can be redirected with the aid of the *t_redrq* and *t_redin* functions.

i This manual describes primary CMX applications based on the conventional UNIX process model. In this manual, the statements made by processes to threads are intended for multithread applications. For more information on the creation of multithreaded CMX applications you should refer in particular to the function calls *t_redin* and *t_redrq* (see section “t_redin - Accept redirected connection (redirection indication)” on page 185 and section “t_redrq - Redirect connection (redirection request)” on page 189).

Example 1: TS application with one process and multiple threads

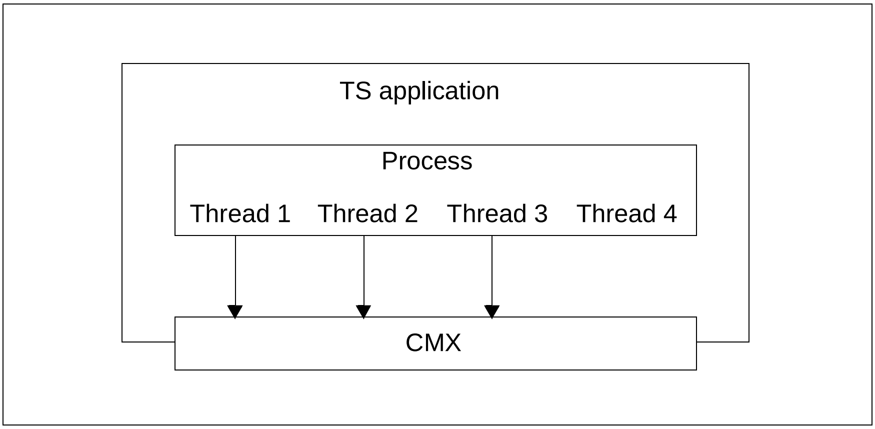


Figure 7: Structure of a multithreaded TS application with a single process

The process contains four threads. Threads 1, 2 and 3 are currently attached to CMX. Thread 4 is not attached to CMX.

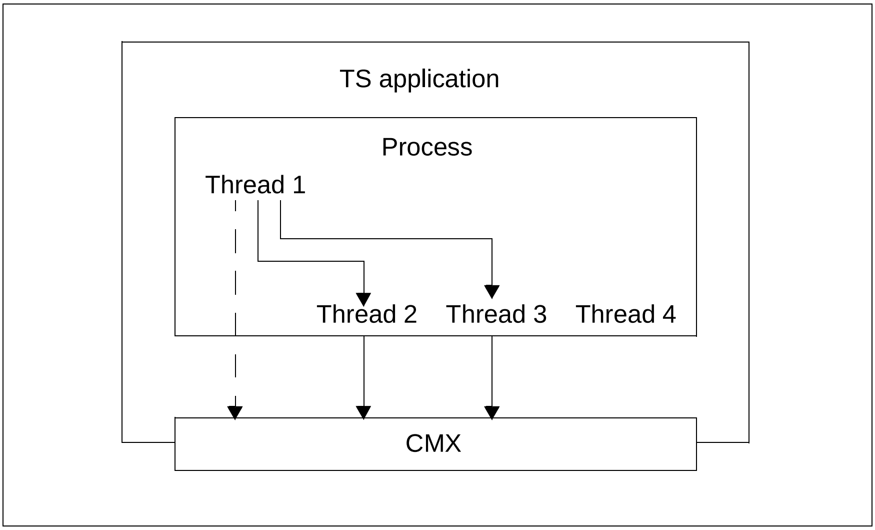


Figure 8: Connection redirection inside a multithreaded process

In this example, thread 1 acts as the distributor thread. It is attached to the CMX and directs incoming requests to establish a connection to other threads inside the process. These threads independently execute the data transfer phase. Thread 1 can also establish additional connections.

Example 2: TS application with several processes each with multiple threads

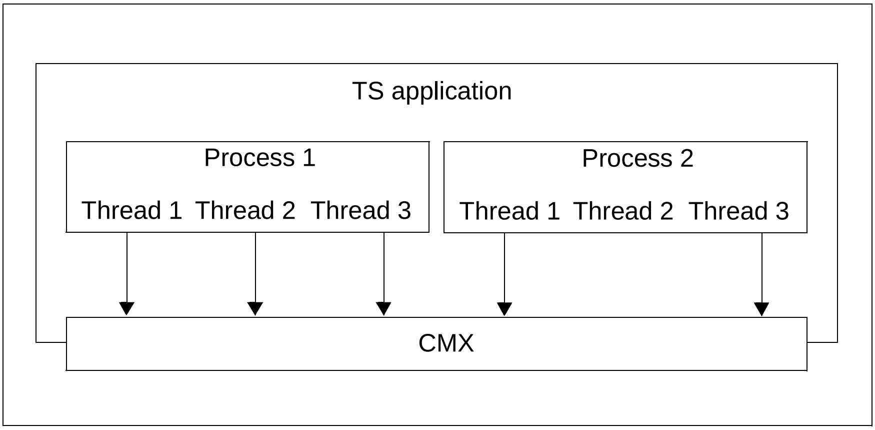


Figure 9: Structure of a multithreaded TS application with several processes

The application works with two processes each of which contains multiple threads. All the threads, with the exception of thread 2 in process 2 are attached to the CMX.

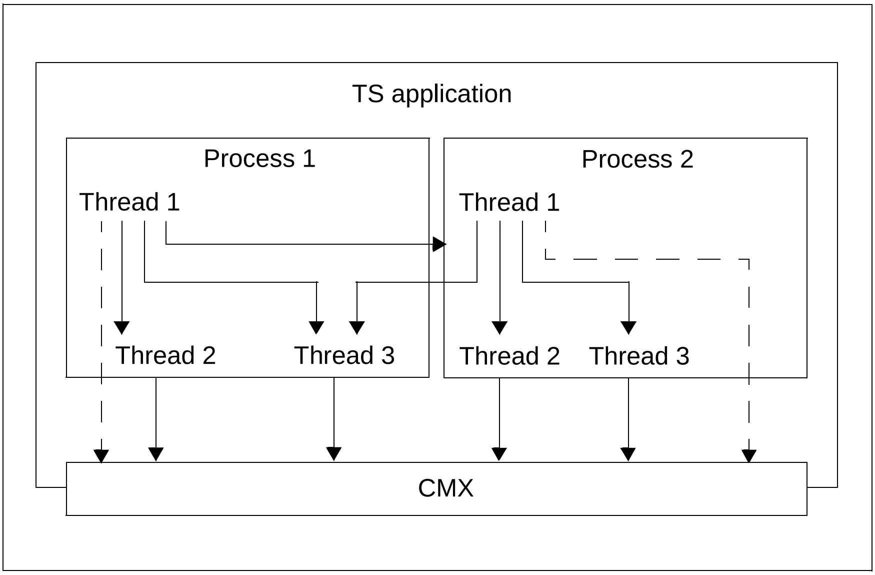


Figure 10: Connection redirection between multiple processes with multiple threads

In this example, the threads 1 in processes 1 and 2 act as the distributor threads. They are attached to the CMX and direct incoming requests to establish a connection to other threads inside their own processes and to a thread in any other process. They can also process requests to establish a connection.

Include files

POSIX threads require the include files *<pthread.h>*, *<errno.h>*, *<limits.h>*, *<signal.h>*, *<types.h>* and *<unist.h>*. The include files are in the directory */usr/include*.

CMX library functions

t_attach()

Each thread to be used with CMX must be attached to CMX with the *t_attach()* function. All other connection-specific CMX calls such as *t_datarq()* can only be executed from threads which have been attached with *t_attach()* and which have established (or contain) a connection made with *t_conrq()*, *t_conin()* or *t_redin()*.

The CMX connections of the individual threads are strictly separated from each other. A thread cannot access the CMX connections of other threads.

t_redin(), t_redrq()

These two functions use the process ID to identify the process to which a connection is to be redirected or to identify the process from which a connection is to be obtained. If the connection is to be redirected to a specified thread in the receiver process then the thread ID must also be stated.

These are the possible types of connection redirection:

– to any other thread in the same process
– to a specified other thread in the same process
– to any other thread in another process
– to a specified other thread in another process

In the case of redirection to any thread, CMX will select the thread. The thread must have the following characteristics:

– It must be able to receive redirected connections (T_REDIN set with *t_attach()*).

– It must be attached to the same TS application (LOCAL NAME set with *t_attach()*)

– Its connection limit has not yet been exceeded (parameter *t_conlim* set with *t_attach()*).

t_setopt()

The *t_setopt()* function can only be used to change thread-specific data. The thread trace can be switched on and off. When the trace is switched on, the trace range is set with the options -s, -S and -D. Parameters which refer to the entire process cannot be changed.

Compiling and linking

- In order to ensure full compatibility with POSIX 1003.1c when compiling with MT programs, you should use the switch _POSIX_C_SOURCE=199506L.
- To compile multithreaded code, set the _REENTRANT switch.
- Use the compile option -mt to switch on all the options to be used with multi-threading.

For linking you can continue to use the Solaris standard linker.

- Link multithreaded applications to the *libpthreadcmx.so* library (option -lpthreadcmx) instead of the *libcmx.so* library (option -lcmx).
- Link programs with -lpthread. This means you must access the definitions in *<pthread.h>* where you should enter the compile instruction -lpthread as the last switch.
- For explicit links, place link *libpthread.so* before *libc.so* (*libc* has predefined libpthread stubs, i.e. dummy functions).
- Place the -lpthread switch in the link statement (ld) before the -lc switch.

Example:

```
cc -mt [flags] file ... -lpthreadcmx -lpthread -lc
cc -mt [flags] file ... -lpthreadcmx -D_POSIX_C_SOURCE=199506L \
                        -D_EXTENSIONS_ -lpthread -lc
```

For more details on compiling and linking multithreaded applications, see the “Multithreaded Programming Guide” published by Sun Microsystems.

i The -mt switch corresponds to the -D_REENTRANT plus -lthread. Non-threaded and single-threaded applications are compiled without the _REENTRANT, _POSIX_C_SOURCE and __EXTENSIONS__ (or mt) flags.

Signals

Single-threaded CMX supports any signal (e.g. SIGIO) used to inform the application of the presence of an event. This means, for example, that an application reading *stdin* can be blocked and interrupted as soon as a CMX event is present.

Other information

Thread ID

The thread ID is a pthread_t data type and not an integer type.

errno variable

Make the *errno* variable thread-specific by using the *<errno.h>* include file.

Thread generation and termination

CMX functions cannot generate or terminate threads.

Detach status

Use the *pthread_attr_setdetachstate()* or *pthread_attr_getdetachstate()* functions to define if the thread resources are to be used again.

Threads attached to CMX can only be detached using the *t_detach()* function. If you try to detach a thread using thread library functions (e.g. *pthread_detach()*, *pthread_exit()* or *pthread_cancel()*), this can cause the loss of thread-specific CMX library resources.

Stack handling

The standard size is 1 Mbyte.
You can use the thread functions *pthread_attr_setstacksize()* and *pthread_attr_setstackaddr()* to define the stack size. The minimum recommended stack size for CMX is 32 kByte.

Library trace

The library trace output also gives the thread ID. This enables thread-specific post processing with cmxl prepared ASCII trace data. The prolog of the ASCII trace file indicates if the file was generated by a single-threaded or a multi-threaded application. See also the manual “CMX, Operation and Administration“ [1].

4 Event processing and error handling

4.1 Receiving events

The operations involved during communications between TS applications are asynchronous, i.e. a wide variety of events can occur independently of the behavior of a TS application. Events are requests and responses received by CMX from other TS applications in the network or messages from the transport systems involved.

Examples of such events are:

– The connection request of a communication partner (the "calling application")

– The arrival of data via an existing connection

– Flow control events (set and released send locks)

– Disconnection by the communication partner or CMX

CMX forwards these events to the TS application when the *t_event()* function is called by the TS application. Exactly one event is passed by CMX for each *t_event()* call, possibly with the identification of the connection involved (transport reference). The TS application must then directly process the received event as required, e.g. by calling the corresponding "fetch" function.

A routine to be called instead of the internal "waiting for/checking events" routine can be passed to CMX using the *t_callback()* call. The program waits in this routine for CMX and program-specific events.

The CMX functions are designed in a manner that allows, but does not compel, the TS application to wait for a possible answer from the network after issuing a call. There are three ways in which a TS application can process events:

– Synchronous processing

– Asynchronous processing

– Event processing in the program

Synchronous processing

The TS application calls *t_event()* with the parameter *cmode* = T_WAIT. As long as no event is waiting, the process sleeps and consumes no CPU time. When there is an event (T_CONIN in Fig. 11), CMX awakens the process, and *t_event()* returns the code of the event and, when appropriate, the transport reference of the connection involved.

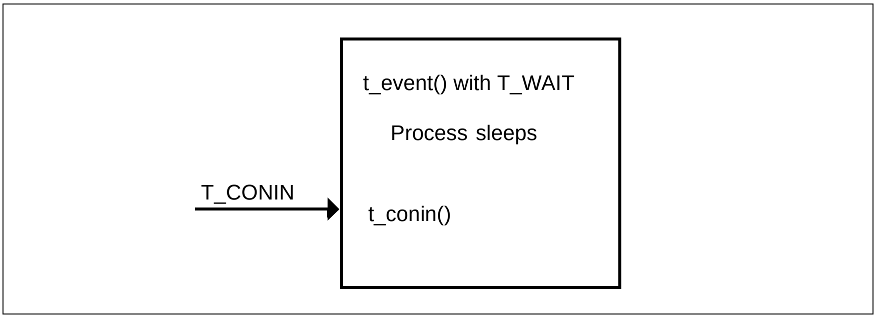


Figure 11: Synchronous processing

Even when the process is sleeping in *t_event()*, it can be awakened using signals. CMX will then resume it with T_NOEVENT, if a handler is defined for the signal.

When *t_event()* is called it is also possible to limit the waiting time. Simply specify how long the process is to wait for an event. If no event arrives within this time, CMX will resume the process with T_NOEVENT.

Asynchronous processing

Call *t_event()* with the parameter *cmode* = T_CHECK. If no event is waiting, the call will immediately return with T_NOEVENT. You may continue with any processing and subsequently call *t_event()* again to check for a possible event.

However, it is not wise to just have *t_event()* run in a continuous loop; it is better to use synchronous event processing (*cmode* = T_WAIT), and wake up the process periodically by using *alarm()* if required.

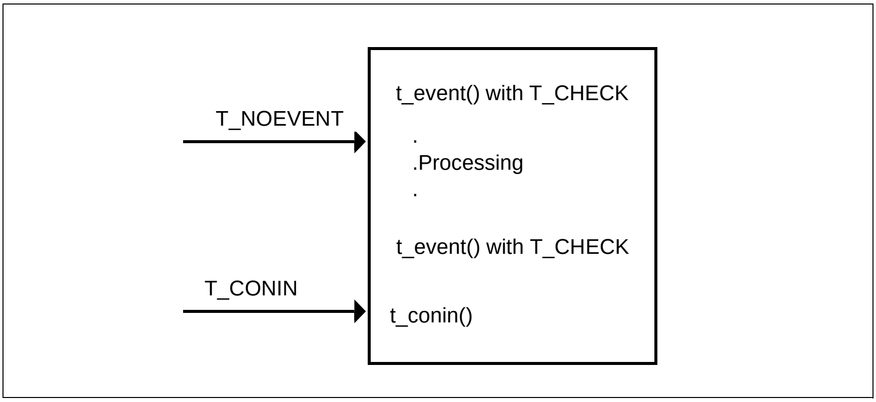


Figure 12: Asynchronous processing

CMX expects a particular reaction, depending on which event was reported. Since program execution is determined by what events occur, the program logic can be largely encapsulated in a switch construction whose cases are the various events (as in the sample programs). If a TS application is to communicate via the migration interface ICMX(NEA) it must fetch events using the call *x_event()*.

Event processing in the program

The *callback()* call can be used to insert your own callback routine. This routine is called instead of the internal "event waiting point" routine during *t_event()*. In the callback routine the program must wait for/check CMX events and can also wait for/check its own events.

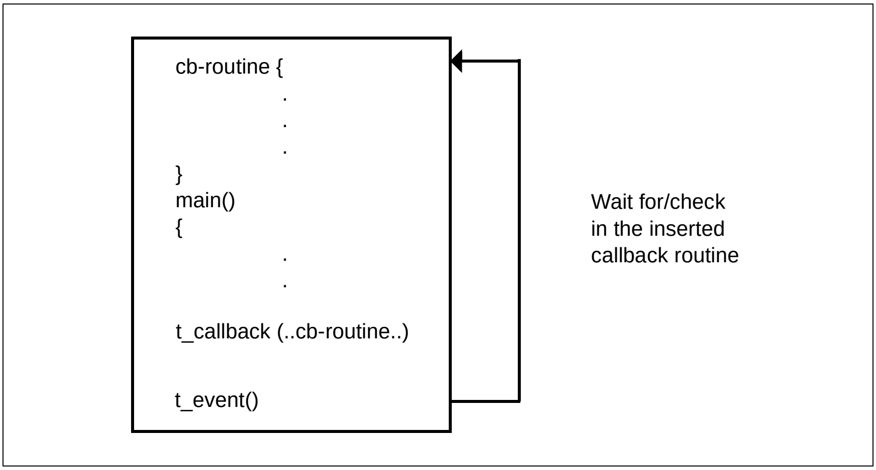


Figure 13: Inserting and activating a callback routine

4.2 Error handling

4.2.1 Error checking functions

A function call resulting in an error always returns with a global error indicator. A more precise value is obtained by calling the error checking function. The following table shows which function calls and error indicators are applicable to the individual CMX program interfaces:

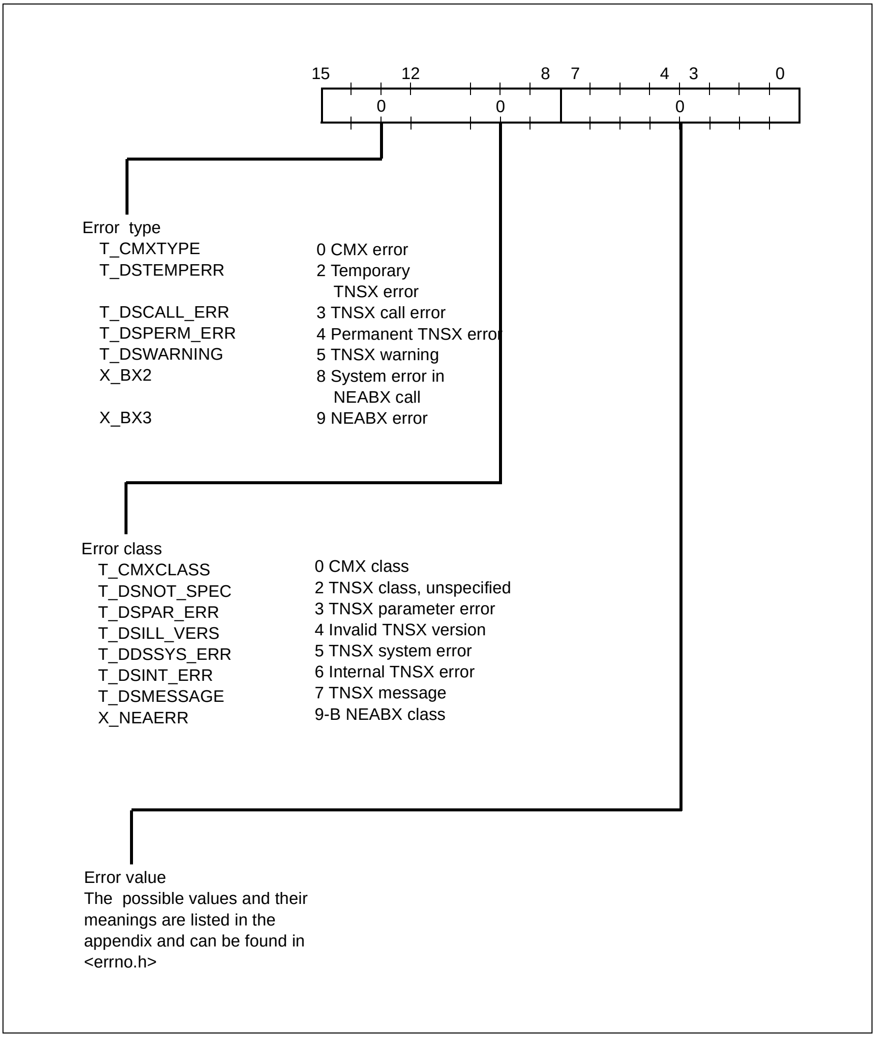
Interface	**Function calls**	**Global error indicator**	**Error checking function**
ICMX(L)	t_....	T_ERROR	t_error()
ICMX(NEA)	x_....	X_ERROR	x_error()

Table 4: Error checking functions

The values returned by *t_error()* or *x_error()* are in hexadecimal form.

4.2.2 Format of CMX error messages

Every error message at ICMX(L) and ICMX(NEA) is passed in the form 0x%x, where %x is an error code with a length of 16 bits. The error code is structured as follows:



Error type

T_CMXTYPE	0 CMX error
T_DSTEMPERR	2 Temporary TNSX error
T_DSCALL_ERR	3 TNSX call error
T_DSPERM_ERR	4 Permanent TNSX error
T_DSWARNING	5 TNSX warning
X_BX2	8 System error in NEABX call
X_BX3	9 NEABX error

Error class

T_CMXCLASS	0 CMX class
T_DSNOT_SPEC	2 TNSX class, unspecified
T_DSPAR_ERR	3 TNSX parameter error
T_DSILL_VERS	4 Invalid TNSX version
T_DDSSYS_ERR	5 TNSX system error
T_DSINT_ERR	6 Internal TNSX error
T_DSMESSAGE	7 TNSX message
X_NEAERR	9-B NEABX class

Error value
The possible values and their meanings are listed in the appendix and can be found in <errno.h>

Figure 14: Format of CMX error messages

Error messages are interpreted beginning on the left (bit 15).

The error values for the individual functions of ICMX(L) and ICMX(NEA) are listed in the appendix in as far as the errors are generated in CMX. Other error values can be obtained from <*errno.h*>.

4.2.3 Decoding error messages

In addition to the diagnostic functions *t_error()* and *x_error()*, special function calls are provided at the ICMX(L) and ICMX(NEA) interfaces in order to convert error codes into plain English.

The program *cmxdec* can be used at the command level to decode ICMX error messages and the reasons for disconnection.

Reasons for disconnection are the values that are returned in the *reason* parameter when *t_disin()*, or *x_disin()* is called. They specify why a connection was closed down or rejected.

The error code or the value of *reason* is decoded by *cmxdec*. The symbolic value defined in the appropriate header file is written to *stderr*. If a corresponding message catalog exists, an explanatory text is also output.

The program *cmxdec* is described in the relevant “CMX, Operation and Administration” manual [1] or [2].

5 Attaching to/detaching from CMX

A TS application comes into existence as soon as a process attaches itself to CMX using the application’s LOCAL NAME. Each further process wishing to operate within this TS application must also attach itself to CMX for this TS application, i.e. by using the same LOCAL NAME.

Before a process terminates it must detach itself from CMX. When the last process of a TS application has detached itself from CMX, the TS application no longer exists for CMX.

5.1 Attaching to CMX

A process attaches itself to CMX via the ICMX(L) interface by calling *t_attach()*.

When doing this the process must pass the LOCAL NAME of the TS application for which it wishes to attach itself to CMX.

The process must read the LOCAL NAME from the TS directory prior to attachment, i.e. before the *t_attach()* call. To do this, it calls the ICMX(L) function *t_getloc()* and passes to *t_getloc()* a parameter with the GLOBAL NAME of the TS application for which it wishes to attach itself. *t_getloc()* returns a pointer to a structure in which the LOCAL NAME is stored. This pointer is passed as a parameter in *t_attach()*.

Thus, the *t_getloc()* call must precede the *t_attach()* call.

When the first process of a TS application attaches itself, a Transport Service Access Point (TSAP) is created for the TS application. The TSAP is the point at which the transport service is accessible. It is assigned the LOCAL NAME of the TS application.

When attaching itself, each process of a TS application specifies:

– whether it wishes to actively set up connections for the TS application. The TS application can then assume the role of the "calling TS application" in the subsequent connection setup phase.

– whether it wishes to wait passively on behalf of the TS application for connection requests from other TS applications in the network. The TS application can then assume the role of the "called TS application" during the course of communication.

– whether it will accept connections that another process of the same TS application wishes to pass to it (i.e. whether it will accept connection redirection). A process of the same TS application means a process that has attached itself to CMX using the same LOCAL NAME.

A process may attach itself to CMX for all three of these possibilities, or for only one or two of them.

The same process can also be attached to several different TS applications. To do this, it must call *t_attach()* and *t_getloc()* for each of these TS application.

CMX accepts connection requests from remote TS applications on behalf of a TS application as soon as a process of the TS application has attached itself to CMX for passive connection setup. Incoming connection requests are initially forwarded by CMX to the process that was the first in the TS application to attach itself for passive connection setup.

Only after successful attachment can a process call other CMX functions, i.e. issue other *t_...()* calls.

Notes on attaching via ICMX(NEA)

The procedure for attaching a TS application at the ICMX(NEA) program interface is analogous to the one above; the only difference is that the call *x_attach()* must be used instead of *t_attach()*. The LOCAL NAME can be likewise read from the TS directory with the help of *t_getloc()*.

5.2 Detaching from CMX

Before a process terminates, it calls *t_detach()*. *t_detach()* detaches the process from CMX for that TS application. First, however, all TS connections maintained by the process must be closed down (siehe chapter “Managing connections between TS applications” on page 49). If the process does not do this, CMX implicitly closes down all TS connections itself. This is, however, provided only for exceptional situations, for example when a process is terminated prematurely.

When the last process of a TS application has detached itself, the TS application no longer exists for CMX. Connection requests from remote TS applications will no longer be accepted for that TS application.

Notes on detaching via ICMX(NEA)

The procedure for detaching a TS application at the ICMX(NEA) program interface is analogous to the one above; the only difference is that the function *x_detach()* must be used instead of *t_detach()*.

5.3 Examples of attaching and detaching a process

5.3.1 Example of attaching and detaching a process at ICMX(L)

The following program fragment shows the program execution sequence when a process is attached and detached at the ICMX(L) interface.

A process attaches itself to CMX for the TS application "Test_application_ACT" and then detaches itself. In the option structure *t_opta1* it specifies that it only wishes to actively set up connections in this TS application (T_ACTIVE), and that no more than one connection is to be simultaneously maintained.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
.
.
#define ERROR   1
 .
 .
struct  t_opta1 t_opta1 = { T_OPTA1, T_ACTIVE, 1 };
                                             /* t_attach () */
 .
 .
 /* Structures for addressing */
#define MYNAME  "Test_application_ACT"
char myname[TS_LPN+1] = { MYNAME } ;
struct t_myname t_myname, *p_myname;
   .
   .
/* Attach active application to CMX */
if ((p_myname = t_getloc(myname, NULL)) != NULL)
        t_myname = *p_myname;
else {
       fprintf(stderr, ">>> ERROR 0x%x in t_getloc\n",
t_error());
       exit(ERROR);
}
if (t_attach(&t_myname, &t_opta1) == T_ERROR) {
       fprintf(stderr, ">>> ERROR 0x%x in t_attach\n",
               t_error());
       exit(ERROR);
}
```

```
fprintf(stderr, "Application ’%s’ attached.\n", myname);
  .
  .
/* Detach TS application from CMX */
if (t_detach(&t_myname) == T_ERROR)
        fprintf(stderr, ">>> ERROR 0x%x in t_detach\n",
                             t_error());
fprintf(stderr, "Application ’%s’ detached.\n", myname);
   .
   .
```

5.3.2 Example of attaching and detaching a process at ICMX(NEA)

The following program fragment shows the program execution sequence when a process is attached and detached at the ICMX(NEA) interface.

A process attaches itself to CMX for the TS application "NEA_application_ACT" and then detaches itself. In the option structure *t_opta1* it specifies that it only wishes to actively set up connections in this TS application (X_ACTIVE), and that no more than one connection is to be simultaneously maintained.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
#include        <neabx.h>
.
.
#define ERROR   1
 .
 .
struct  x_opta1 x_opta1 = { X_OPTA1, X_ACTIVE, 1 };
                                          /* x_attach () */
 .
 .
 /* Structures for addressing */
#define MYNAME  "NEA_application_ACT"
char myname[TS_LPN+1] = { MYNAME } ;
struct x_myname x_myname, *p_myname;
   .
   .
/* Attach active application to ICMX(NEA) */
if ((p_myname = t_getloc(myname, NULL)) != NULL)
        x_myname = *p_myname;
else {
       fprintf(stderr, ">>> ERROR 0x%x in t_getloc\n",
                            t_error());
       exit(ERROR);
}
if (x_attach(&x_myname, &x_opta1) == X_ERROR) {
       fprintf(stderr, ">>> ERROR 0x%x in x_attach\n",
                        x_error());
       exit(ERROR);
}
fprintf(stderr, "Application '%s' attached.\n", myname);
  .
  .
/* Detach TS application from ICMX(NEA) */
if (x_detach(&x_myname) == X_ERROR)
        fprintf(stderr, ">>> ERROR 0x%x in x_detach\n",
                             x_error());
fprintf(stderr, "Application '%s' detached.\n", myname);
   .
   .
```

6 Managing connections between TS applications

Connection setup and disconnection involve two TS applications. One is the calling TS application; it initiates connection setup. The other is the called TS application, with whom the calling TS application wishes to establish a connection. The following sections elucidate the relationships and sequences.

The fact that CMX is displayed only once in the diagrams is just a simplification of the presentation. Actually, each partner uses "his" CMX in his processor, and in between stand the network and the transport systems.

6.1 Establishing a connection

The processing sequence in the course of setting up a connection at ICMX(L) is explained first. The following figure illustrates the chronological sequence of ICMX(L) calls in the programs of the calling and called TS application.

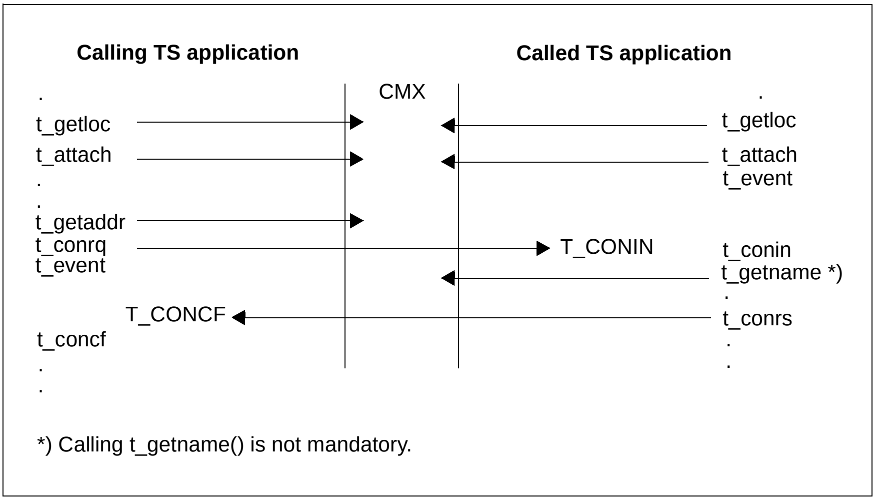


Figure 15: Establishing a connection (ICMX(L))

Course of connection setup in the calling TS application

The process of the calling TS application must inform CMX when attaching itself to it that it intends to actively set up a connection. The calling TS application first obtains its LOCAL NAME and then attaches itself to CMX. Thereafter it ascertains the TRANSPORT ADDRESS of the called TS application and requests a connection using *t_conrq()*.

It then waits with *t_event()* for confirmation of the called TS application, i.e. for the TS event T_CONCF. When *t_event()* has reported the TS event, the calling TS application establishes the connection with the call *t_concf()*.

Course of connection setup in the called TS application

Each process of the called TS application must inform CMX when attaching itself to it that it intends to passively set up a connection. After being attached, the called TS application initially waits for a TS event with *t_event()*. The TS event T_CONIN indicates the connection request of the calling TS application. The called TS application accepts this connection indication with the call *t_conin()*. It can then ascertain from the TRANSPORT ADDRESS of the calling TS application its GLOBAL NAME, and answers the connection request with *t_conrs()*.

Exchanging user data during connection setup

The reason the calls *t_conin()* (connect indication) and *t_concf()* (connect confirmation) are required is that both TS applications can already exchange user data while the connection is being set up, if the transport system supports this option (see section “System and user options” on page 12).

With *t_conrq()* the calling TS application may pass user data, i.e. a small quantity of data that the called TS application receives with *t_conin()*. If the called TS application then answers the connection request with *t_conrs()*, it in turn may also pass information. This is received by the calling TS application with *t_concf()*.

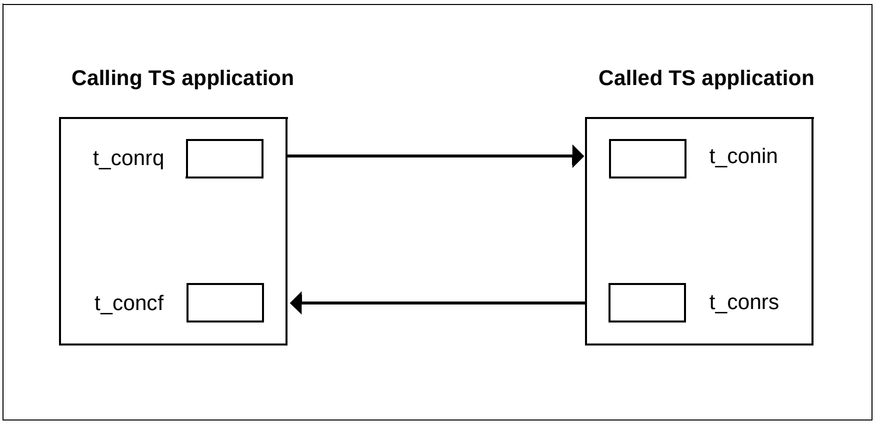


Figure 16: Exchange of user data during connection setup

Rejecting a connection request

The called TS application may also reject the connection request. The sequence is the same. The event T_CONIN must first be accepted with *t_conin()*, but instead of the call *t_conrs()* the call *t_disrq()* is issued (see also section “Closing down a connection” on page 55).

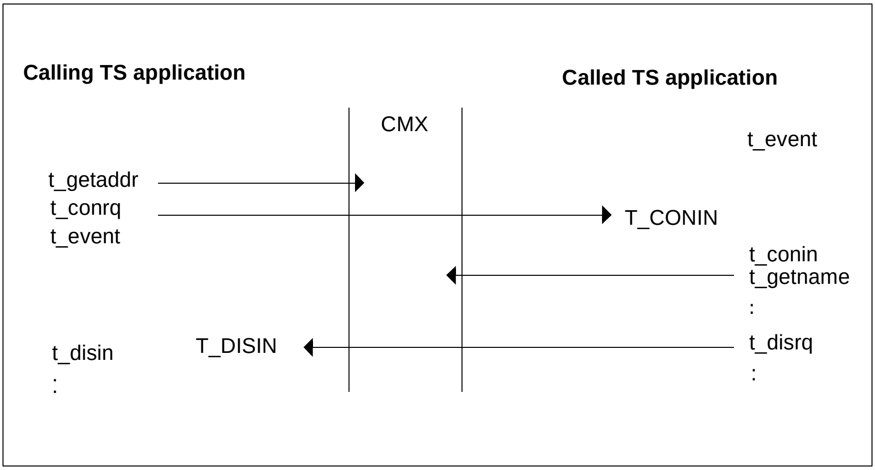


Figure 17: Rejecting a connection request

Notes on connection setup at ICMX(NEA)

Functions of the ICMX(NEA) program interface call ICMX(L) functions internally. The operations described above are thus also applicable here, provided the user data to be exchanged can be directly transmitted by CMX. TS applications using ICMX(NEA) are, however, required to pass the NEABV protocol in the user data. The length of this user data may exceed the user data length permitted by the transport system. As a result, user data passed with *x_conrq()* cannot be transmitted by CMX by using a *t_conrq()* call.

The chronological sequence of the resulting operations is illustrated in the figure below:

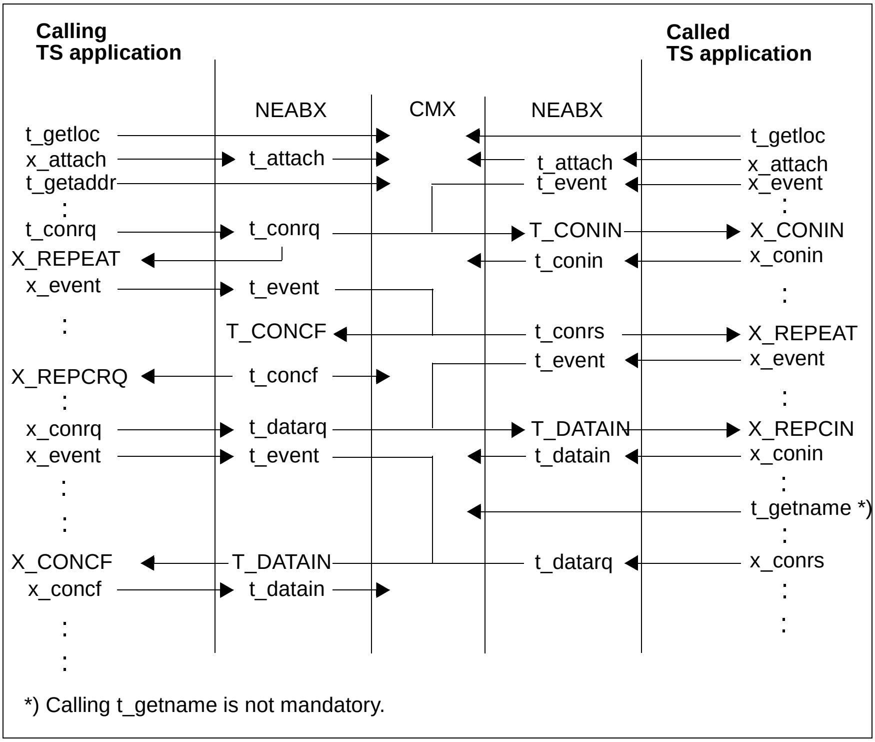


Figure 18: Connection setup with ICMX(NEA)

On calling *x_conrq()*, the **calling TS application** receives the return value X_REPEAT. It must now call *x_event()* and wait until NEABX reports the event X_REPCRQ. Following this, it must repeat *x_conrq()* with the same parameters. It is only when the value T_OK is returned to the TS application on calling *x_conrq()* and when the following *x_event()* call reports the event X_CONCF that the connection to the called TS application can be set up with *x_concf()*.

After attaching itself to CMX, the **called TS application** calls *x_event()*. If the connect indication X_CONIN arrives, the called TS application calls *x_conin()*. The value X_REPEAT is returned as the result. The TS application must now call *x_event()* again and wait for the event X_REPCIN. After this event arrives, the TS application accepts the user data by repeating the *x_conin()* call and then confirms the connection request with *x_conrs()*.

In this connection setup type, CMX establishes an internal connection with the partner CMX and transmits the user data with the help of the ICMX(L) calls for data transmission (*t_datarq(), t_datain()*).

If the called TS application wishes to reject the connection, it must call *x_disrq()* instead of *x_conrs()*.

Agreeing on expedited data

If the transport system provides the expedited data option, the TS applications may agree on its use during connection setup. This takes place as follows:

With the connection request with *t_conrq()* the calling TS application makes a proposal, which the called TS application can only "negotiate down". This means: If the calling TS application proposes not using any expedited data, then this is settled for the connection. If on the other hand it proposes that expedited data be exchanged, the called TS application may accept or reject this in its connection response with *t_conrs()*. In both cases the answer is binding.

If one of the two TS applications does not agree with the result of the expedited data negotiation, it may close down the connection.

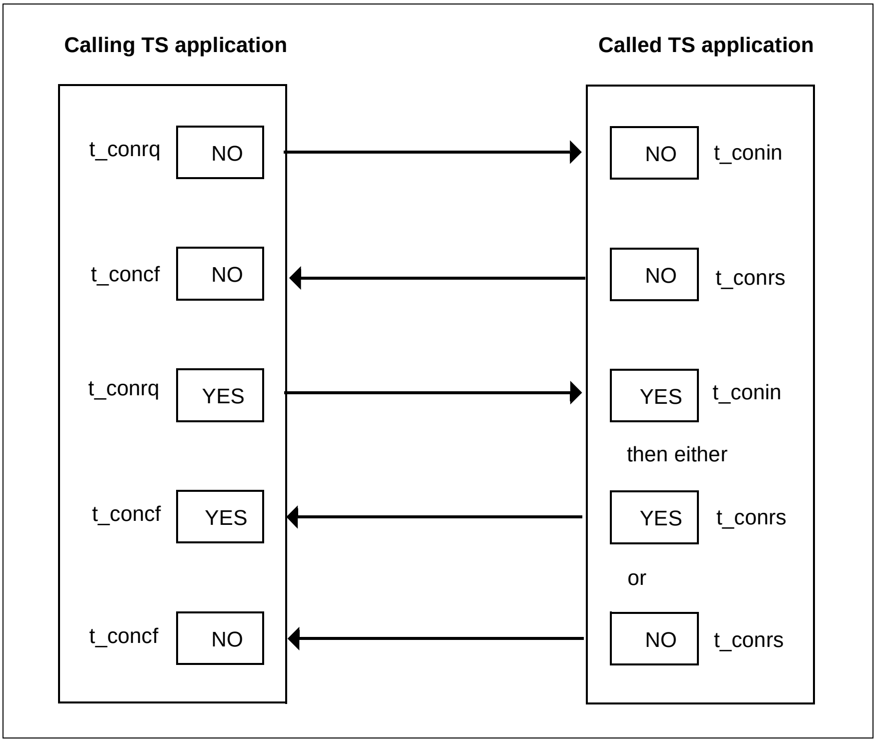


Figure 19: Negotiation regarding expedited data during connection setup

If the TS applications are to communicate via the migration interface ICMX(NEA), the prefix x_ must be used instead of t_ for the calls given above, and the prefix X_ must replace T_ for the events.

6.2 Closing down a connection

Either of the two communicating TS applications may call *t_disrq()* in order to close down the connection. The partner TS application then receives the event T_DISIN.

By calling *t_disin()* it accepts the disconnection. With this call it obtains the reason for the disconnection.

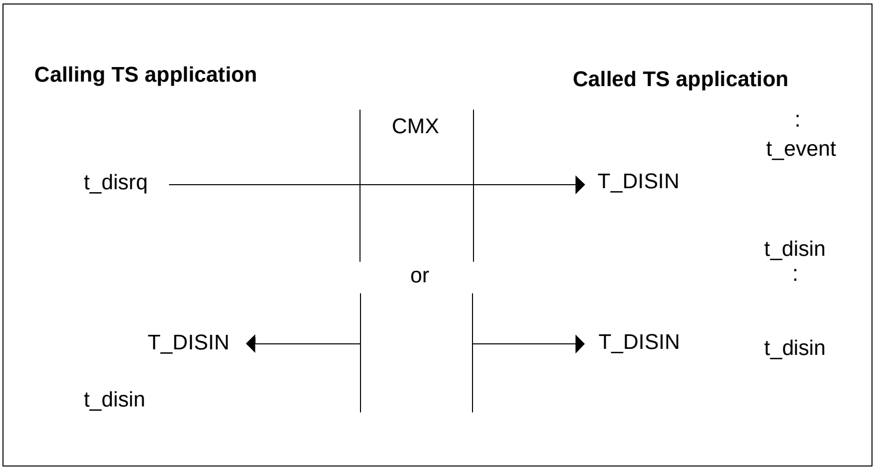


Figure 20: Closing down a connection

If the transport system provides for it, the TS application that closes down the connection may include user data with *t_disrq()*. The partner TS application receives this with *t_disin()*.

The connection may also be closed down by CMX. In this case, both TS applications receive the event T_DISIN, which they must fetch with *t_disin()*. Based on the reason given for the disconnection each TS application can ascertain whether the connection was closed down by the other TS application or by CMX.

If the CMX application is to communicate via the migration interface ICMX(NEA), the calls *x_disrq()* and *x_disin()* must be used instead of *t_disrq()* and *t_disin()*.

6.3 Example of setting up and closing down a connection with ICMX(L)

6.3.1 Examples of establishing a connection with ICMX(L)

The two following program fragments show how a connection is set up.

Example 1 shows the program structure for the calling TS application.

Example 2 shows the program structure for the called TS application.

Example 1:

The TS application actively sets up a connection to the TS application “Test_application_PAS” and then closes it down.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
   .
   .
#define ERROR   1
   .
   .
int     tref;                    /* Transport reference */
int     reason;                  /* Reason for disconnection */
 /* Structures for addressing */
#define PNAME   "Test_applicaton_PAS"
char pname[TS_LPN+1] = { PNAME } ;
struct t_partaddr t_partaddr, p_partaddr;
   .
   .
/* set up the connection of the passive partner */
if ((p_partaddr = t_getaddr(pname, NULL)) != NULL)
         t_partaddr = *p_partaddr;
else {
  fprintf(stderr, ">>> ERROR 0x%x at t_getaddr\n", t_error());
  exit(ERROR);
}
if (t_conrq(&tref, (union x_address *)&t_partaddr,
          (union x_address *)&t_myname, NULL) == T_ERROR) {
  fprintf(stderr, ">>> ERROR 0x%x at t_conrq, tref 0x%x\n",
          t_error(), tref);
  exit(ERROR);
}
/* event-driven processing */
* t_event() waits synchronously (T_WAIT) */
/*
for (;;) {
        switch (event = t_event(&tref, T_WAIT, NULL)) {
        case T_CONCF:
           /*
            * Connection setup successfull?
            */
           if (t_concf(&tref, NULL) == T_ERROR) {
                  fprintf(stderr, ">>> ERROR 0x%x at t_concf
                          tref 0x%x\n",
                          t_error(), tref);
                  exit(ERROR);
           }
           fprintf(stderr, "Connection established
                             to’%s’ .\n", pname);
           .
           .
        case T_DISIN:
            /* Disconnection by partner or system */
             if (t_disin(&tref, &reason, NULL) == T_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x at t_disin
                tref 0x%x\n",
                        t_error(), tref);
                exit(ERROR);
             }
             fprintf(stderr, "Received disconnect indication,
                     tref          0x%x,
                     reason %d\n", tref, reason);
            .
            .
        }
}
 /* Disconnection */
 if (t_disrq(&tref, NULL) == T_ERROR){
      fprintf(stderr, ">>> ERROR 0x%x at t_disrq tref 0x%x\n",
                    t_error(), tref);
      exit(ERROR);
 }
fprintf(stderr, "Connection tref 0x%x actively closed down.\n",
                 tref);
    .
    .
```

Example 2

The TS application waits passively for an incoming connection request, accepts the connection, and then closes it down.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
   .
   .
#define ERROR   1
   .
   .
int     tref;                   /* Transport reference */
int     reason;                 /* Reason for disconnection */
/*
 * Structures for addressing
 */
struct t_myname t_myname, *p_myname;
struct t_partaddr t_partaddr;
   .
   .
   .
/* Event-driven processing:
 * t_event() waits synchronously (T_WAIT)
 */
for (;;) {
        switch (event = t_event(&tref, T_WAIT, NULL)) {
        case T_CONIN:
          /* Accept connection request */
          if (t_conin(&tref, (union x_address *)&t_myname,
                   (union x_address *)&t_partaddr, NULL) ==
                    T_ERROR) {
            fprintf(stderr, ">>> ERROR 0x%x at t_conin
                    tref 0x%x\n",
                    t_error(), tref);
            exit(ERROR);
          }
          if (t_conrs(&tref, NULL) == T_ERROR) {
             fprintf(stderr, ">>> ERROR 0x%x at t_conrs tref
                     0x%x\n",
                     t_error(), tref);
             exit(ERROR);
             }
             .
             .
        case T_DISIN:
            /*
```

```
             * Disconnection by partner or system
             */
            if (t_disin(&tref, &reason, NULL) == T_ERROR) {
             fprintf(stderr, ">>> ERROR 0x%x at t_disin tref
             0x%x\n",
                      t_error(), tref);
             exit(ERROR);
           }
           fprintf(stderr, "Received disconnect indication, tref
           0x%x,
                   reason %d\n", tref, reason);
                        .
                        .
        }
}
/*
 * Disconnection
 */
if (t_disrq(&tref, NULL) == T_ERROR){
    fprintf(stderr, ">>> ERROR 0x%x at t_disrq tref 0x%x\n",
            t_error(), tref);
    exit (ERROR);
}
fprintf(stderr, "Connection tref 0x%x actively closed down
            .\n", tref);
    .
    .
```

6.3.2 Examples of establishing a connection with ICMX(NEA)

Example 1 shows the program structure for the calling TS application.

Example 2 shows the program structure for the called TS application.

Example 1:

The TS application actively sets up a connection to the TS application “NEA_application_PAS and then closes it down.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
#include        <neabx.h>
   .
   .
#define ERROR   1
   .
   .
int     tref;                   /* Transport reference */
int     reason;                 /* Reason for disconnection */
 /* structures for addressing */
#define PNAME   "NEA_application_PAS"
char pname[TS_LPN+1] = { PNAME } ;
struct x_partaddr x_partaddr, *p_partaddr;
struct x_optc1 x_optc1 ;
char   *udatap = "User connection message, exceeding 32
                  characters" ;
int     retval ;
char    answer[X_MSG_SIZE] ;
                     /* User message received with x_conf  */
   .
   .
/* Set up connection to the passive partner */
if ((p_partaddr = t_getaddr(pname, NULL)) != NULL)
         x_partaddr = *p_partaddr;
else {
      fprintf(stderr, ">>> ERROR 0x%x at t_getaddr\n",
      t_error());
      exit(ERROR);
}
x_optc1.x_optnr = X_OPTC3 ;
x_optc1.x_xdata = X_YES ;
x_optc1.x_timeout = T_NOLIMIT ;
x_optc1.x_prot = X_NEABX ;
```

```
x_optc1.x_udatap = udatap ;
x_optc1.x_udatal = strlen(udatap) ;
if ((retval=x_conrq(&tref, (union x_address *)&x_partaddr,
          (union x_address *)&x_myname, &x_optc1)) == X_ERROR) {
     fprintf(stderr, ">>> ERROR 0x%x at x_conrq, tref 0x%x\n",
           x_error(), tref);
     exit(ERROR);
}
/* Event-driven processing :
 * x_event() waits synchronously (X_WAIT)
 */
for (;;) {
        switch (event = x_event(&tref, X_WAIT, NULL)) {
        case X_REPCRQ :
        if (x_conrq(&tref, (union x_address *)&x_partaddr,
              (union x_address *)&x_myname, &x_optc1) ==
               X_ERROR) {
      fprintf(stderr, ">>> ERROR 0x%x on repeating
      x_conrq, tref 0x%x\n",
              x_error(), tref);
          exit(ERROR);
           }
        break ;
        case X_CONCF:
           /*
            * Connection setup sucessfull?
            */
           x_optc1.x_udatap = answer ;
           x_optc1.x_udatal = sizeof(answer) ;
           if ((retval=x_concf(&tref, &x_optc1)) == X_ERROR) {
                  fprintf(stderr, ">>> ERROR 0x%x at x_concf
                  tref 0x%x\n",
                          x_error(), tref);
                  exit(ERROR);
           }
           if ( retval == X_REPEAT )
              break ;
           else
               fprintf(stderr, "Connection established to ’%s
               .\n", pname);
           .
           .
        case X_REPCCF :
              /*
               * fetch confirmation again
               */
           if (x_concf(&tref, &x_optc1) == X_ERROR) {
      fprintf(stderr, ">>> ERROR 0x%x on repeating x_concf
      tref 0x%x\n",
          x_error(), tref);
                  exit(ERROR);
           }
           fprintf(stderr, "Connection established to %s’ .\n",
           pname);
           .
           .
        case X_DISIN:
            /* Disconnection by partner or system */
            if (x_disin(&tref, &reason, NULL) == X_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x at x_disin
                tref 0x%x\n",
                        x_error(), tref);
                exit(ERROR);
            }
            fprintf(stderr, "Received disconnect indication,
                              tref
            0x%x,
                   reason %d\n", tref, reason);
            :
        }
}
 /* Disconnection */
 if (x_disrq(&tref, NULL) == X_ERROR){
            fprintf(stderr, ">>> ERROR 0x%x at x_disrq tref
            0x%x\n",
                    x_error(), tref);
            exit(ERROR);
 }
 fprintf(stderr, "Connection tref 0x%x actively closed down.\n",
 tref);
    .
    .
```

Example 2:

The TS application waits passively for an incoming connection request, accepts the connection, and then closes it down.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
#include        <neabx.h>
   .
   .
#define ERROR   1
   .
   .
int     tref;                   /* Transport reference */
int     reason;                 /* Reason for disconnection */
/*
 * structures for addressing
 */
struct x_myname x_myname, *p_myname;
struct x_partaddr x_partaddr;
struct x_optc1 x_optc1 ;
char   *answer = "User connection message, exceeding 32
                  characters" ;
int     retval ;
char    udatap[X_MSG_SIZE] ;
                 /* User message received with x_conin */
   .
   .
   .
/* Event-driven processing:
 * x_event() waits synchronously (X_WAIT)
 */
for (;;) {
        switch (event = x_event(&tref, X_WAIT, NULL)) {
        case X_CONIN:
        case X_REPCIN :
            /* Accept connection request */
            x_optc1.x_optnr = X_OPTC3 ;
            x_optc1.x_udatap = udatap ;
            x_optc1.x_udatal = sizeof(udatap) ;
            if ((retval=x_conin(&tref, (union x_address
            *)&x_myname,
                (union x_address *)&x_partaddr, &x_optc1))
            == X_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x at x_conin
                tref 0x%x\n",
                        x_error(), tref);
                exit(ERROR);
            }
            if ( retval == X_REPEAT )
              break;
              /* Wait for X_REPCIN */
            x_optc1.x_udatap = answer ;
            x_optc1.x_udatal = strlen(answer) ;
            if (x_conrs(&tref, &x_optc1) == X_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x at x_conrs
                tref 0x%x\n",
                        x_error(), tref);
                exit(ERROR);
                }
                .
                .
        case X_DISIN:
            /*
             * Disconnection by partner or system
             */
            if (x_disin(&tref, &reason, NULL) == X_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x at x_disin
                tref 0x%x\n",
                        x_error(), tref);
                exit(ERROR);
             }
             fprintf(stderr, "Received disconnect indication,
                              tref
             0x%x,
                    reason %d\n", tref, reason);
                .
                .
        }
}
/*
 * Disconnection
 */
if (x_disrq(&tref, NULL) == X_ERROR){
    fprintf(stderr, ">>> ERROR 0x%x at x_disrq tref 0x%x\n",
            x_error(), tref);
    exit (ERROR);
}
fprintf(stderr,"Connection tref 0x%x rejected or actively
 closed down.\n", tref);
    .
    .
```

6.4 Redirecting connections

Incoming connections for a local TS application are initially received by the process that first attached itself for that TS application. Now in order e.g. to be able to associate particular connections with particular processes, a connection may be redirected to another process. Of course, actively set up connections may also be redirected.

Both processes must belong to the same TS application, i.e. they must have attached themselves with the same LOCAL NAME. However, they do not have to be related. The receiving process must indicate its readiness to accept a connection redirection when attaching itself to CMX.

Sequence in redirecting a connection

Process A specifies the process ID of process B when calling *t_redrq()*. Process B receives the event T_REDIN and must initially accept the connection, with the call *t_redin()*. With this call process B is informed of the process ID of process A. If process B does not wish to have the connection, it may close it down or redirect it further, e.g. back to process A.

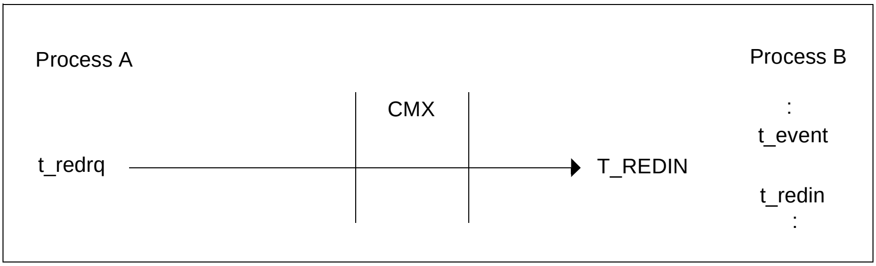


Figure 21: Redirecting a connection

With *t_redrq()* it is also possible to include user data, which process B receives when it calls *t_redin()*.

Notes on redirecting a connection with ICMX(NEA)

If the TS application is to communicate via the migration interface ICMX(NEA), the calls *x_redrq()*, *x_redin()* and *x_event()* must be used instead of *t_redrq()*, *t_redin()* and *t_event()*. Process B receives the event X_REDIN. It is essential to specify a storage area for the transmission of user data, since the migration service must create a message to pass an internal NEABX protocol.

6.4.1 Example of redirecting a connection

The following program fragments show how a connection can be redirected and how a redirected connection is accepted.

6.4.1.1 Example of redirecting a connection at ICMX (L)

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
  .
#define ERROR   1
  .
int     tref;     /* Transport reference */
int     cpid;     /* ID of process to receive connection */
int     rpid;     /* ID of process wanting to relinquish
                     connection */
  .
  .
/* Actively redirect connection */
if (t_redrq(&tref, &cpid, NULL) == T_ERROR) {
    fprintf(stderr, ">>> ERROR 0x%x at t_redrq tref 0x%x\n",
            t_error(), tref);
    exit(ERROR);
}
fprintf(stderr, "Connection redirected to #%d \n", cpid);
 .
 .
/* Accept connection redirection */
for (;;) {
        switch (event = t_event(&tref, T_CHECK, NULL)) {
        case T_REDIN:
             if (t_redin(&tref, &rpid, NULL) == T_ERROR) {
                 fprintf(stderr, ">>> ERROR 0x%x at t_redin
                 tref 0x%x\n",
                         t_error(), tref);
                 exit(ERROR);
             }
             fprintf(stderr, "Connection received from #%d .\n",
                                  rpid);
             .
             .
         }
}
```

6.4.1.2 Example of redirecting a connection at ICMX(NEA)

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
#include        <neabx.h>
  .
#define ERROR   1
  .
int     tref;   /* Transport reference */
int     cpid;   /* ID of process to receive connection */
int     rpid;   /* ID of process wanting to relinquish
                    connection */
struct x_optc2 x_optc2 ;
char message[X_RED_SIZE] ;
  .
/* actively redirect connection */
strcpy(message,"private");
x_optc2.x_optnr = X_OPTC2 ;
x_optc2.x_udatap = message ;
x_optc2.x_udatal = strlen(message) + X_RED_PL ;
if (x_redrq(&tref, &cpid, &x_optc2) == X_ERROR) {
    fprintf(stderr, ">>> ERROR 0x%x at x_redrq tref 0x%x\n",
            x_error(), tref);
    exit(ERROR);
}
fprintf(stderr, "Connection redirected to #%d \n", cpid);
 .
/* Accept connection redirection */
for (;;) {
        switch (event = x_event(&tref, X_WAIT, NULL)) {
        case X_REDIN:
            x_optc2.x_optnr = X_OPTC2 ;
            x_optc2.x_udatap = message ;
            x_optc2.x_udatal = sizeof(message) ;
             if (x_redin(&tref, &rpid, &x_optc2) == X_ERROR) {
                 fprintf(stderr, ">>> ERROR 0x%x at x_redin
                 tref 0x%x\n",
                         x_error(), tref);
                 exit(ERROR);
             }
             fprintf(stderr, "Connection received from #%d .\n",
             rpid);
             .
             .
         }
}
```

7 Transmitting data

Once a connection has been set up, the two TS applications can exchange data. Either TS application may initiate the data exchange regardless of whether it is the calling or the called TS application.

The amount of data forming a logical unit from the point of view of the TS applications is referred to as a message, or TSDU (Transport Service Data Unit). A TSDU may be any length (but see also section “Transport system specific features” on page 103).

However, CMX can accept only a limited amount of data at any one time. This is referred to as a data unit or TIDU (Transport Interface Data Unit). The maximum length of a TIDU depends on the transport system. This length must be queried for every connection using the call *t_info()*, or *x_info()* in the case of ICMX(NEA).

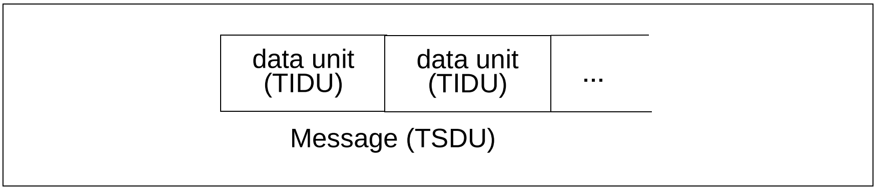


Figure 22: TIDU and TSDU

The logical linkage of TIDUs to form a TSDU is controlled by means of a parameter, which specifies for each TIDU in a message whether it is followed by a further TIDU or is the last one in the TSDU.

If the transport system provides the option, and both TS applications agree to it when the connection is set up, they may also exchange expedited data. Expedited data is a small quantity of data that is given priority over normal data, i.e. expedited data never arrives later than normal data sent subsequently to the expedited data.

Expedited data must always be transmitted all at once. A unit of expedited data is called an ETSDU (Expedited Transport Service Data Unit).

7.1 Sending and receiving normal data

Normal data is sent with one of the calls *t_datarq()* or *t_vdatarq()*.

Each such call sends at most one TIDU. *t_datarq()* is called when the TIDU to be sent is contained in one contiguous storage area. *t_vdatarq()* is called when the TIDU to be sent is located in several different storage areas.

In the simplest case data transfer proceeds as follows:

– The sending TS application passes one TIDU to CMX with each call.

– The receiving TS application receives the event T_DATAIN. This indicates that data has arrived.

– The receiving TS application must accept the data with the call *t_datain()* or the call *t_vdatain()*.

t_datain() and *t_vdatain()* differ in that with *t_datain()* the data is placed into one contiguous storage area while with *t_vdatain()* the data is placed into several different storage areas.

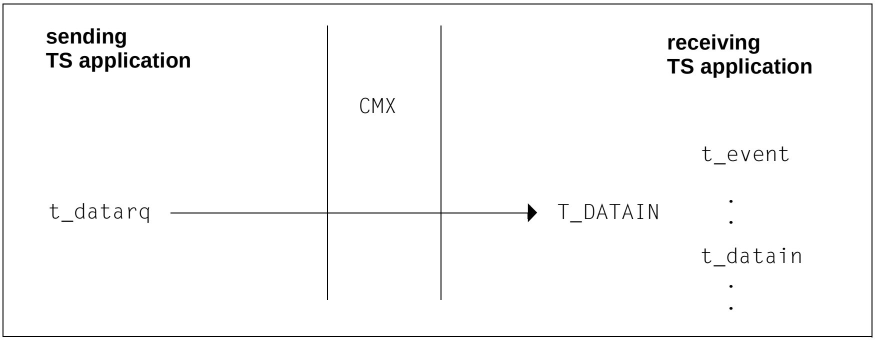


Figure 23: Transmitting normal data

If the TSDU is longer than one TIDU ...

it must be broken down into multiple TIDUs. This is done as follows:

- The sending TS application determines, as sender, when the TSDU is ended. Each time a TIDU is sent with *t_datarq()* or *t_vdatarq()*, this TS application indicates in the *chain* parameter whether a further TIDU of the current TSDU is to follow (*chain* = T_MORE) or the TIDU being sent is the last one (*chain* = T_END).
- In the same way, the receiving TS application is informed with each *t_datain()* or *t_vdatain()* call by *chain* as to whether there is another TIDU to come in the current TSDU.

Each TIDU is announced by CMX with a T_DATAIN event. However, the length of a TIDU may be different for each of the two TS applications. Therefore it may happen that the receiving TS application will need to call *t_datain()* or *t_vdatain()* less often than the sending TS application calls *t_datarq()* or *t_vdatarq()* (or vice-versa), because the receiving TS application reads TIDUs in "its" length. The situation then looks like this:

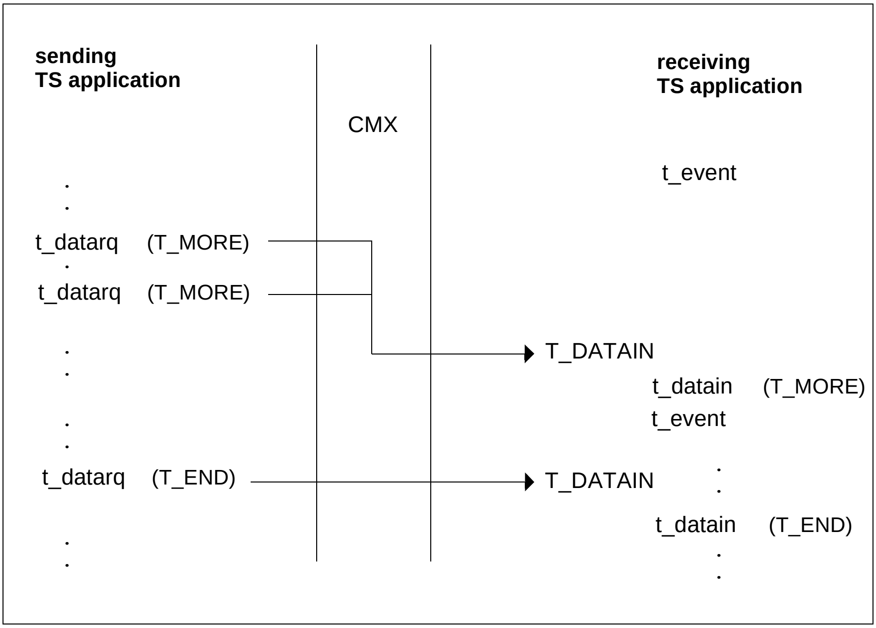


Figure 24: TSDU in multiple TIDUs

The value returned by t_datain() and t_vdatain()

With *t_datain()* and *t_vdatain()* you must specify a length for the incoming data to be read. If the length specified is less than the size of the TIDU for the sender, the value returned by *t_datain()* or *t_vdatain()* will indicate the excess length of the data in the waiting TIDU.

If a TIDU has not yet been completely read, *t_datain()* or *t_vdatain()* must be called repeatedly until the TIDU has been completely read. During this time, *t_event()* may not be called, the connection may not be redirected nor may the data flow be controlled.

Note that CMX does not guarantee that at the receiving TS application all TIDUs of a message will be completely filled, even when the size of a TIDU is the same for both the sending and the receiving TS application and the sending TS application sends only completely filled TIDUs.

Notes on transmitting data via ICMX(NEA)

If the TS application is to communicate via the migration interface ICMX(NEA), the calls *x_datarq()*, *x_datain()*, and *x_event()* must be used instead of the calls *t_datarq()*, *t_datain()*, and *t_event()*. Calls corresponding to *t_vdatarq()* and *t_vdatain()* are not provided in ICMX(NEA). The event indicated is X_DATAIN; the parameter values for message length are X_MORE and X_END. Note that with ICMX(NEA) there are limitations regarding data length; in particular, reading data units in piecemeal fashion is not possible.

Furthermore, when setting up the connection, the TS applications can agree on transmitting the NEABX protocol in the form of user data. An option structure is used for controlling and interpreting the NEABX protocol.

7.2 Examples of transmitting normal data

The following program fragments illustrate the program execution sequence when transmitting normal data via ICMX(L) and ICMX(NEA).

7.2.1 Example of transmitting normal data via ICMX(L)

The TS application receives and sends data. The length of the data is limited here to one TIDU.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
.
.
#define ERROR   1
.
.
/* Send and receive buffers */
char    e_bufpt[8000];          /* Receive buffer */
int     e_bufl;                 /* Transfer length */
char    s_bufpt[8000];          /* Send buffer */
int     s_bufl;                 /* Transfer length */
int     chain;                  /* TSDU indicator for */
                                /* t_datarq(), t_datain() */
int     tref;                   /* Transport reference */
 .
 .
/* Event-driven processing: */
 * t_event() waits synchronously (T_WAIT) */
for (;;) {
        switch (event = t_event(&tref, T_WAIT, NULL)) {
        .
        .
        /* Receive data; e_bufl is the TIDU length (t_info()) */
        case T_DATAIN:
            if ((rc = t_datain(&tref,e_bufpt,&e_bufl,&chain)) ==
               T_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x in t_datain
                 tref 0x%x\n",t_error(), tref);
                exit (ERROR);
             }
             :
        }
}
```

```
/* Send data; s_bufl is maximum TIDU length */
if ((rc = t_datarq(&tref, s_bufpt, &s_bufl, &chain)) ==
           T_ERROR) {
        fprintf(stderr, ">>> ERROR 0x%x in t_datarq tref
                 0x%x\n",t_error(), tref);
        exit(ERROR);
}
```

7.2.2 Example of transmitting normal data via ICMX(NEA)

The following program fragments illustrate the program execution sequence when transmitting normal data via ICMX(NEA). It is assumed that, when calling *x_conrq()* or *x_conrs()*, the TS application has agreed the use of the NEABX protocol in the data phase.

```
#include        <stdio.h>
#include        <cmx.h>
#include        <tnsx.h>
#include        <neabx.h>
.
.
#define ERROR   1
.
.
/* Send and receive buffers */
struct  x_optd1 e_optd1 ;       /* Receive structure */
char    e_bufpt[8000];          /* Receive buffer */
int     e_bufl;                 /* Transfer length */
struct  x_optd1 s_optd1 ;       /* Send structure */
char    s_bufpt[8000];          /* Send buffer */
int     s_bufl;                 /* Transfer length */
int     chain;                  /* TSDU indicator for */
                                /* x_datarq(), x_datain() */
int     tref;                   /* Transport reference */
 .
 .
/* Event-driven processing:
 * x_event() waits synchronously (X_WAIT) */
for (;;) {
        switch (event = x_event(&tref, X_WAIT, NULL)) {
        .
        .
        /* Receive data; e_bufl is the TIDU length (x_info()) */
        case X_DATAIN:
            e_optd1.x_optnr = X_OPTD2 ;
            e_bufl = sizeof (e_bufpt);
            if ((rc = x_datain(&tref,e_bufpt,&e_bufl,&chain,
            (x_optd *)&e_optd1))== T_ERROR) {
                fprintf(stderr, ">>> ERROR 0x%x in x_datain
                tref 0x%x\n",x_error(), tref);
                exit (ERROR);
             }
             .
             .
        }
}
/* Send data; s_bufl is maximum TIDU length */
s_optd1.x_optnr = X_OPTD2 ; /* Output length in s_bufl is net */
s_optd1.x_code = X_ASCII;
s_optd1.x_strukt = X_ETXEOT;
if ((rc = x_datarq(&tref, s_bufpt, &s_bufl, &chain, (x_optd
*)&s_optd1))== X_ERROR) {
        fprintf(stderr, ">>> ERROR 0x%x in x_datarq tref
                 0x%x\n",x_error(), tref);
        exit(ERROR);
}
```

7.3 Sending and receiving expedited data

If the exchange of expedited data was agreed at connection setup (see section section “Establishing a connection” on page 49), the TS applications may do so as follows:

Expedited data is sent with the call *t_xdatrq()*. In the simplest case the sequence is as follows:

– The sending TS application sends expedited data with a call.

– The receiving TS application receives the event T_XDATIN. This indicates that expedited data has arrived.

– The receiving TS application must accept the data with the call *t_xdatin()*.

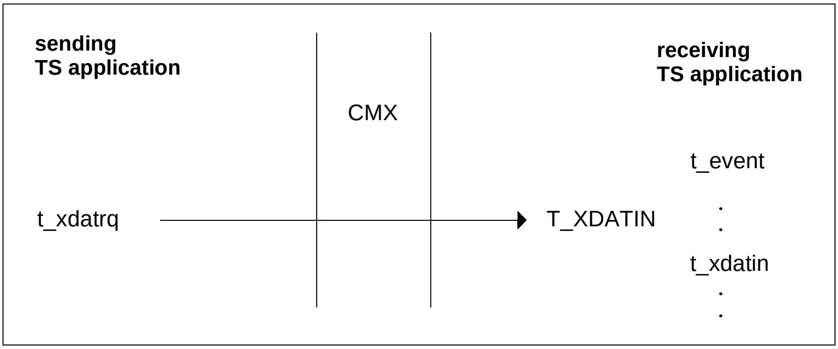


Figure 25: Transmitting expedited data

The value returned by t_xdatin()

With *t_xdatin()* a length must be specified for the incoming expedited data to be read. If the length specified is less than the amount of expedited data that has arrived, the value returned by *t_xdatin()* will then give the excess length of the waiting expedited data.

If the expedited data has not yet been completely read, *t_xdatin()* must be called repeatedly until the data has been completely read. During this time *t_event()* may not be called, the connection may not be redirected nor may the data flow be controlled.

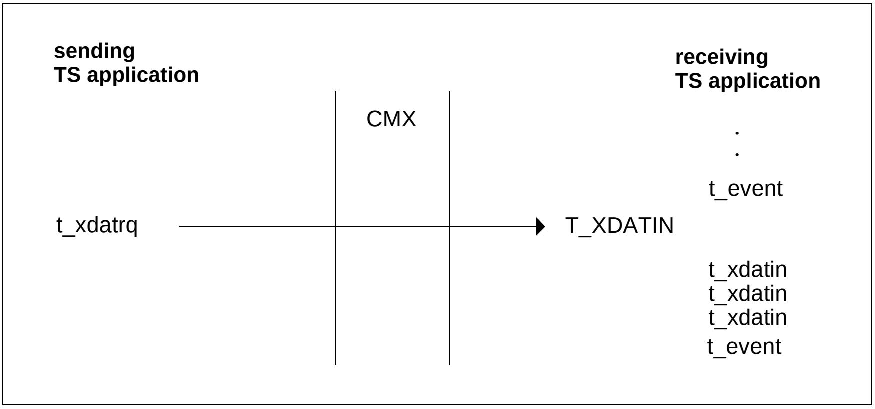


Figure 26: Reading expedited data in piecemeal fashion

Note on transmitting expedited data via ICMX(NEA)

If the TS application is to communicate via the migration interface ICMX(NEA), the calls *x_xdatrq()*, *x_xdatin()*, and *x_event()* must be used instead of the calls *t_xdatrq()*, *t_xdatin()*, and *t_event()*. The event indicated is X_XDATIN. Note that with ICMX(NEA) there are limitations regarding data length. In particular, reading expedited data in piecemeal fashion is not possible.

Furthermore, when setting up the connection, the TS applications can agree on transmitting the NEABX protocol in the form of user data. An option structure is used for controlling and interpreting the NEABX protocol. The relevant stipulations are described in the section “NEABV protocol” on page 232.

7.4 Flow control of normal and expedited data

If a TS application is not ready to receive data over a connection, it informs CMX of this with the call *t_datastop()*. CMX immediately stops delivering the event T_DATAIN for that connection. For *t_datarq()* the communication partner will receive the return value T_DATASTOP from CMX and may not send any more data.

As soon as the TS application is again ready to receive data over the connection it calls *t_datago()*. The communication partner will receive the event T_DATAGO and the TS application may again receive data from it. It again receives the event T_DATAIN.

Flow control for expedited data takes place in the same way. Here the calls *t_xdatstop()* and *t_xdatgo()* are used. The corresponding events are T_XDATIN and T_XDATGO.

Note however:

When the flow of expedited data is stopped (with *t_xdatstop()*), CMX also implicitly stops the flow of normal data. When the flow of expedited data is then released again (with *t_xdatgo()*), the flow of normal data remains blocked. It must be expressly released (with *t_datago()*).

When the flow of normal data is released CMX implicitly also releases the flow of expedited data again. Thus, after calling *t_xdatstop()*, calling *t_datago()* releases both the flow of normal data and the flow of expedited data.

What do you gain by preventing T_DATAIN or T_XDATIN from being received?

During this time the TS application can issue other CMX calls, e.g. to set up a further connection. This would not be possible if a T_DATAIN event were waiting. If this were the case, and the TS application did not fetch the data, every *t_event()* call would again return the event T_DATAIN and the TS application would not be able to receive the event T_CONCF, required to set up a connection.

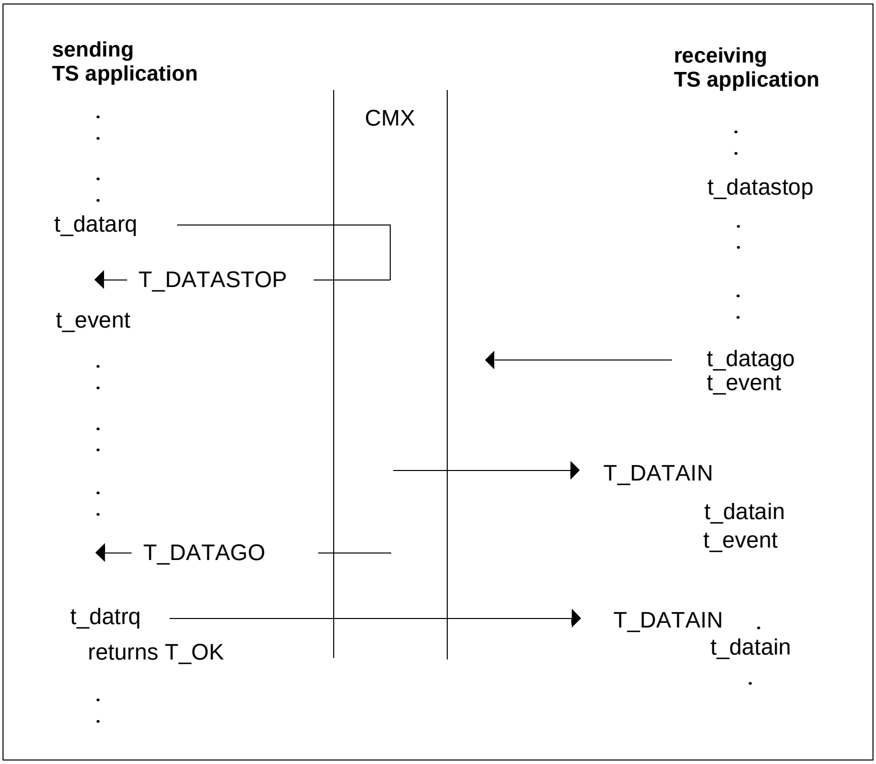


Figure 27: Data flow control at the sending end

The sending TS application receives T_DATASTOP in response to the call *t_datarq()* or *t_vdatarq()*, because the receiving TS application has stopped the data flow or because there is a temporary resource bottleneck in CMX. The data was sent, but no longer indicated to the receiving TS application. The sending TS application must now wait with *t_event()* for the event T_DATAGO, in order to be able to send data again.

Notes on data flow control at ICMX(NEA)

If the TS application is to communicate via the migration interface ICMX(NEA), the calls with the prefix x_ must be used instead of the calls with the prefix t_. The same applies for the events, where the prefix T_ is replaced by X_. Note that with ICMX(NEA) the value X_DATASTOP may also be returned for the calls *x_datain()* and *x_xdatin()*. In this case, before the receiving TS application can send data it must wait for the event X_DATAGO.

8 The ICMX(L) program interface

This chapter describes the ICMX(L) program interface to the communication manager CMX. It contains:

– A summary of the functions of the ICMX(L) interface, with details on the communication phases,
– Notes on the correct use of the functions (finite-state automata),
– A description of the features specific to transport systems,
– Notes on the availability of the system options for the transport systems,
– Precise descriptions of the ICMX(L) function calls, with all parameters, in alphabetical order.

8.1 Overview of the program interface

Transport Service ISO 8072

With ICMX(L), the present version of CMX provides a program interface to the connection-oriented transport service (TS) as defined in ISO 8072 within the framework of the OSI Reference Model for open systems. Therefore in ICMX(L) the services T-CONNECT (connection setup), T-DISCONNECT (disconnection), T-DATA (data exchange), and T-EXPEDITED-DATA (exchange of expedited data) are defined with the primitives:

T-CONNECT.request
T-DISCONNECT.request
T-CONNECT.indication
T-DISCONNECT.indication
T-CONNECT.response
T-CONNECT.confirmation
T-DATA.request
T-DATA.indication
T-EXPEDITED-DATA.request
T-EXPEDITED-DATA.indication

Table 5: Service primitives in ICMX (L)

In addition, ICMX(L) provides local services that simplify the implementation of TS applications. These are:

T-ATTACH	Attach a TS application to CMX
T-DETACH	Detach a TS application from CMX
T-ERROR	Query errors
T-REDIRECT	Redirect a connection to another process
T-FLOWCONTROL	Flow control for normal data
T-EXPEDITED-FLOWCONTROL	Flow control for expedited data
T-EVENT	TS event check
T-INFO	Information
T-GETADDR	Query TRANSPORT ADDRESS
T-GETLOC	Query LOCAL NAME
T-GETNAME	Query GLOBAL NAME
T-CALLBACK	Register callback routine
T-SETOPT	Set options

The TS permits two TS applications to exchange messages over a transport connection (TC). This connection-oriented communication provides for the exchange of messages without loss or duplication while maintaining the message sequence. Furthermore, by means of connection identification the connection-oriented TS makes it possible to dispense with transferring and processing addresses in the data phase. An established TC is uniquely identified (in both end systems) by a transport reference (tref) between CMX and the TS application. Certain parameters that influence message transport on a TC can be negotiated between the TS applications at connection setup. For the correct functioning of communication certain rules must be observed, which are described in the following.

ICMX(L) is implemented as a set of C functions, which make communication between TS applications independent of the specific characteristics of the transport systems used (layers 1 - 4 in the OSI Reference Model) with regard to profile, protocol classes, etc.

Depending on which TS is used, each TC is assigned one or more special files, which are visible to the TS application only in that they consume the corresponding number of the available file descriptors. These special files simplify the cleanup measures taken in CMX following premature termination of the TS application.

Names and addresses

Every TS application has a GLOBAL NAME. This name uniquely identifies the TS application in the network. GLOBAL NAMES are assigned by the TNSX administration. It must ensure that the names of all TS applications are different from one another.

A TS application works exclusively with GLOBAL NAMES. It obtains information from its GLOBAL NAME using CMX calls, e.g. the LOCAL NAME it must specify when attaching to CMX. It can use the GLOBAL NAME of a remote TS application to ascertain the TRANSPORT ADDRESS it must pass to CMX at connection setup.

The LOCAL NAME links the local TS application to a Transport Service Access Point (TSAP). The TRANSPORT ADDRESS of the remote TS application is required to address the Transport Service Access Point (i.e. the TS application linked to it) in the partner system. The LOCAL NAME and TRANSPORT ADDRESS are read from the TS directory.

ICMX(L) functions for querying information from the TS directory are:

t_getaddr()
: Given the GLOBAL NAME of a TS application, returns its TRANSPORT ADDRESS. The TRANSPORT ADDRESS must be passed through as a parameter to the relevant ICMX(L) call.

t_getname()
: Given a TRANSPORT ADDRESS, returns the GLOBAL NAME of the TS application.

t_getloc()
: Given the GLOBAL NAME of a TS application, returns its LOCAL NAME in the current end system. The LOCAL NAME must be forwarded as a parameter to the relevant ICMX(L) call.

t_getaddrpart() and **t_setaddrpart()**
: Analyzes or modifies a TRANSPORT ADDRESS.

t_getlocpart() and **t_setlocpart()**
: Analyzes or modifies a LOCAL NAME.

<*cmx.h*> defines the structures *t_myname* and *t_partaddr*. *t_myname* is used by a TS application to receive (pass) its LOCAL NAME from (to) the TNSX; *t_partaddr* is used for the TRANSPORT ADDRESS.

The contents of these structures are as follows:

```
struct t_myname {
   char t_mnmode;        /* = T_MNMODE */
   char t_mnres;         /* = 0 */
   short t_mnlng;        /* Length of the filled-in part of
                            the t_myname structure */
   char t_mn[T_MNSIZE];  /* Field for the T-selectors of the
                            LOCAL NAME */
}
struct t_partaddr {
   char t_pamode;        /* = T_PAMODE */
   char t_pares;         /* = 0 */
   short t_palng;        /* Length of the filled-in part of
                            the */
                         /* t_partaddr structure */
   char t_pa[T_PASIZE];  /* Field for the partner address */
}
```

The meanings of members in the structure *t_myname* are shown below:

t_mnmode = T_MNMODE
: specifies that the field *t_mn* contains a LOCAL NAME.

t_mnres, *t_mn*[T_MNSIZE]
: are of no relevance to you. The contents of these members are simply taken from the TNSX and passed on to CMX.

t_mnlng
: specifies the length of all data passed in the structure *t_myname*.

The meanings of members in the structure *t_partaddr* are as follows:

t_pamode = T_PAMODE
: specifies that the field *t_pa* contains a TRANSPORT ADDRESS.

t_pares, *t_pa*[T_PASIZE]
: are of no relevance to you. The contents of these members are simply taken from the TNSX and passed on to CMX.

t_palng
: specifies the length of all data passed in the structure *t_partaddr*.

The LOCAL NAME and TRANSPORT ADDRESS are passed to CMX or received from CMX in the union *t_address*.

```
union t_address {
   struct t_myname tmyname;
   struct t_partaddr tpartaddr;
}
```

Error handling and diagnosis

All function calls return a return code. This is either T_OK, to indicate successful completion, or T_ERROR to generally indicate that an error occurred. The error check function *t_error()*, called immediately following an error, returns more detailed diagnostic information. All errors detected by CMX as violations of the communications rules by the TS application have specific error codes and are defined in <*cmx.h*>. Other errors result from failures in calling functions in the operating system environment in CMX; they can be identified from <*errno.h*>. The transport systems used generate no error messages; any errors result in disconnection with a corresponding reason. The reason for disconnection is obtained by the TS application when *t_disin()* is called.

The following functions return the text version of an error code returned by *t_error()*:

t_strerror()
Returns a pointer to the text string for an error code received from ICMX(L).

t_perror()
Calls *t_strerror()* to ascertain the text string for an error code received from ICMX(L) and writes the string to *stderr*.

The following functions return the text for a disconnection reason returned by *t_disin()*:

t_strreason()
Returns a pointer to the text string for a disconnection reason that has been received. The reason for disconnection is passed to the TS application when *t_disin()* is called.

t_preason()
Calls *t_strreason()* to ascertain the text string for a disconnection reason that has been received with *disin()* and writes the string to *stderr*.

The error decoding program *cmxdec* (see the “Operation and Administration” manual [1] or [2]) provides mechanisms for obtaining these texts at the command line.

For diagnostic purposes ICMX(L) provides a trace facility. It can be flexibly controlled via the environment variable CMXTRACE. The trace mechanism logs the calls with their arguments in compressed form in temporary files. The editing program *cmxl* then converts the log to readable form in a separate step (see the “Operation and Administration” manual [1] or [2]).

TS applications, transport connections and processes

A TS application is a system of programs that uses the TS, i.e. the services of CMX. The mapping of a TS application to the process concept of the system is left up to the implementor. A TS application may organize itself into one or more (not necessarily related) processes. The processes may, essentially independently from one another, maintain TCs to remote TS applications. The processes of a TS application may exchange their TCs among one another. However, at any point in time the transport reference of a TC is assigned to exactly one process. It therefore cannot be inherited by child processes. In CMX there is a separate local service, REDIRECT, for redirecting a TC to another process.

One process may also simultaneously control multiple TS applications. In this case, the implementation must provide for suitable coordination of the execution of the various TS applications. CMX supports this through its asynchronous processing mode.

Synchronicity and asynchronicity; TS events

Communications operations are by nature asynchronous: A wide variety of TS events can occur independently of the activity of a TS application. For example, a TS application may be sending data over one TC when, asynchronously, a disconnection indication for another TC arrives, of which the TS application must be informed immediately.

In principle, the functions of CMX are asynchronous: This means, after issuing a call a TS application need not wait for a possible answer from the network. Any answer will be accepted by CMX when it arrives and sent to the TS application as a TS event at the next opportunity when requested.

For this, CMX provides the TS application with a query mechanism in two forms: Synchronous (waiting) and asynchronous (checking). This query mechanism must be appropriately used by the TS application if it wishes to react quickly and properly to TS events.

With synchronous execution, the calling process is suspended until a TS event arrives. This wakes up the process, so that it can immediately process the TS event. Waiting can be limited by specifying a waiting period or it can be cut short via signals such as SIGALRM. The synchronous mechanism is useful for TS applications that maintain several TCs at a time, so that they need not poll them.

With asynchronous execution, at convenient times, such as at the end of a processing step, the process can check whether a TS event has arrived, and handle it before continuing with the next processing step. This is useful for processes that expect longer delays between TS events, during which times they can or must attend to other operations.

The corresponding function in CMX is

t_event()
If the parameter value T_WAIT is passed, *t_event()* suspends the process until a TS event arrives, the time limit expires, or a signal arrives. If a TS event is already waiting, or there is an error, the function returns immediately with the code for the event, or T_ERROR. The suspended process is awakened when a signal arrives, and *t_event()* returns with T_NOEVENT or T_ERROR. When the time limit expires the process

resumes with the TS event T_NOEVENT. With the parameter value T_CHECK, *t_event()* always returns immediately and returns either the code of the TS event encountered or T_NOEVENT or T_ERROR.

The following asynchronous TS events are defined in CMX:

T_NOEVENT
: In the asynchronous case: No TS event present

  In the synchronous case: Abort by signal or waiting time elapsed

T_CONIN
: Arrival of a connection indication from a calling TS application

T_CONCF
: Arrival of a connection confirmation from a called TS application

T_DISIN
: Arrival of a disconnect indication from a remote TS application or from CMX

T_REDIN
: Arrival of a redirection indication from another process of the same TS application (this TS event is local; it is an extension to the TS to make implementation of TS applications more flexible)

T_DATAIN
: Arrival of normal data from a remote TS application

T_XDATIN
: Arrival of expedited data from a remote TS application

T_DATAGO
: Removal of a block on the sending of normal data and expedited data set through flow control

T_XDATGO
: Removal of a block on the sending of expedited data set through flow control

T_ERROR
: Fatal error; more detailed information is provided by the query function *t_error()*.

With each TS event, except for T_NOEVENT and T_ERROR, the TS application is also given the transport reference, so that it can react for that TC specifically to the TS event.

Some TS events must be accepted by the TS application by calling corresponding functions. Exceptions are: T_ERROR, T_DATAGO, T_XDATGO. Such function calls return additional information on the TS events. The following table lists the TS events and the corresponding functions.

TS event	**Function for fetching**
T_CONCF	t_concf()
T_CONIN	t_conin()
T_DATAGO	t_event()
T_DATAIN	t_datain() or t_vdatain()
T_DISIN	t_disin()
T_REDIN	t_redin()
T_XDATGO	t_event()
T_XDATIN	t_xdatin()

Table 6: TS events and the corresponding functions

As a rule, TS events are delivered in the order in which they occur. Of course, the TS event T_XDATIN may overtake the TS event T_DATAIN, and T_DISIN may overtake T_DATAIN and T_XDATIN. In the latter case the overtaken TS events on that TC are dropped.

Signaling for asynchronous event processing

For asynchronous event processing, CMX provides an optional signaling mechanism to prevent unnecessary *t_event()* calls which return T_NOEVENT. If the signaling mechanism has been activated, every event that occurs is indicated to the process by a signal. This takes place asynchronously to the execution of the process. Following the arrival of a signal, one of your own signal routines or a CMX-internal signal routine is executed. The CMX-internal routine causes the signals to be logged in the CMX tracer. After the signal routine has been run, the process should call *t_event()* to check for the existence of a waiting event and then process it as required.

The signaling mechanism is disabled by default. It can be activated and controlled via the environment variable CMXINIT as shown below:

CMXINIT="-s" activates signaling with signal 22 (SIGIO).

CMXINIT="-S n" activates signaling with signal n (where n = a decimal number).

The value for *n* should be selected appropriately. Not all signals can be intercepted with signal routines.

CMXINIT can be set in C programs as follows:

```
putenv("CMXINIT='-s'"); or putenv("CMXINIT='-S n'";)
```

Within a process, CMXINIT is evaluated once before the first call to ICMX(L).

Signalling is not available for multithreaded applications. The related options in CMXINIT will be ignored.

Attaching/detaching

Communication by a process via CMX is activated when the process attaches itself to CMX. A special file is opened for the process the first time this is done. This special file is used for exchanging jobs between the CMX library functions and the operating system. A TS application is generated when the first process attaches itself for that TS application. When this is done, a Transport Service Access Point (TSAP) is created, at which the TS is accessible. When the first process is attached the TS application is linked to this TSAP. The TSAP is assigned the LOCAL NAME of the TS application. It thereby becomes addressable from the network. When the TS application is detached, any TCs still in existence are closed down, along with the TSAP; the process environment is dissolved and assigned resources are released for future use.

One and the same process may attach itself for several TS applications at once (i.e. manage multiple TSAPs) and in each of these TS applications maintain multiple Transport Connection Endpoints (TCEP). Also, several processes may attach themselves for the same TS application (use the same TSAP) and actively set up TCs or passively wait for connection indications without interfering with one another. Of course, each TCEP is assigned to exactly one process.

The following functions are used for attaching and detaching. They perform primarily local tasks. If no implicit disconnection must be performed, no information is passed to the network.

t_attach()

Attaches (the current process of) a TS application to CMX. When attached, the process may specify its future behavior in the TS application. The first time a process is attached CMX begins accepting connection indications for the TS application.

t_detach()

Detaches (the current process of) a TS application from CMX. Any existing TCs of the process in the TS application are closed down by CMX. If no more processes of the TS application are attached, the TS application is thereafter no longer known to CMX.

Connection setup, disconnection and redirection

Before two TS applications can exchange data, a TC must be set up between them. One of the two TS applications is viewed as the calling TS application; it initiates connection setup. The other is the called TS application; it waits for requests from calling TS applications.

The calling TS application issues a connection request and receives an answer from the called TS application. The called TS application waits for a connection indication (indication of a connection request) and accepts it or rejects it. During connection setup, the TS applications negotiate certain attributes of the TC for the data transmission and may exchange user data.

The TC may be closed down at any time by either of the TS applications or by CMX. This is not negotiated between the TS applications, but instead is immediately carried out by CMX. The other TS application (or both, if CMX closes down the TC) receives a disconnect indication, which may be neither answered nor averted. CMX indicates all errors in the transport systems by closing down the TCs involved. CMX does not guarantee that data still in transit at the time of the disconnection request will be delivered.

Connection redirection is a local service in CMX that simplifies organizing a TS application into processes. A process holding a completely established TC may redirect it (depending, of course, on the state; see figure “States of TS applications and permissible state transitions” on page 97) to another process of the same TS application. The TSAP and the TCEP remain unchanged. The redirecting process loses the transport reference for the TC, whereupon the TC is no longer available to the process.

The relevant functions are:

t_conrq()

Requests connection setup to the called TS application with the specified TRANSPORT ADDRESS. Reference to the TSAP is established via the LOCAL NAME used when the calling TS application was attached. The function returns immediately after issuing the request; the calling TS application receives a transport reference. It must then wait synchronously or asynchronously for the answer of the called TS application (see above).

t_concf()
: Accepts from CMX the answer of the called TS application, indicated with T_CONCF; connection setup is now complete.

t_conin()
: Receives from CMX a connection request, indicated with T_CONIN, from the calling TS application, along with that TS application's TRANSPORT ADDRESS. Reference to the TSAP is established for the called TS application through provision of the LOCAL NAME specified when it was attached.

t_conrs()
: Answers (accepts) a connection request after it has been indicated with T_CONIN and received by the TS application.

t_disrq()
: Requests that a connection be closed down; this function may be called at any time by either of the TS applications; it is also used to reject a connection request (instead of accepting it) after the request has been indicated by CMX and received by the TS application.

t_disin()
: Accepts from CMX the disconnect indication indicated with T_DISIN. The reason for disconnection is also passed to the TS application with this function call.

t_redrq()
: Redirects a TC to a process of the same TS application; the TC is then no longer available for the redirecting process.

t_redin()
: Accepts from CMX a connection redirection indicated with T_REDIN; the receiving process must accept it, but may immediately pass it on (return it) or close the TC down.

Data exchange and flow control

Once a connection has been set up, the initiative rests with the TS application (not with CMX). It may:

– send normal data and (if agreed) expedited data, or

– indicate, with *t_event()*, that it is ready to receive normal data or (if agreed) expedited data.

Data transfer is message-oriented: The TS applications exchange Transport Service Data Units (TSDU) - messages of any length - or Expedited Transport Service Data Units (ETSDU) - expedited data of limited length. Expedited data is limited to a few bytes; when transferred it is given priority over the stream of normal data and placed into separate queues. CMX guarantees only that expedited data will never arrive at the receiving TS application later than normal data sent subsequently. At most one complete ETSDU may be passed to CMX per call.

A TSDU (which in principle may be any length) is passed to CMX in portions the length of one Transport Interface Data Unit (TIDU). The length of a TIDU is TC-specific and must therefore be queried by CMX for each TC (*t_info()*). Thus, a TSDU may have to be transferred using multiple send calls. A parameter in each send call indicates whether a further TIDU for that TSDU follows (T_MORE) or not (T_END). It cannot be determined from this how a TIDU is packed for transfer or delivery to the receiving TS application. CMX guarantees only that sequential joining of the TIDUs on the receiving side will reproduce the TSDU from the sending side. The TIDU length may be different for the two TS applications and depends on the TC. CMX does not guarantee that at the receiving TS application any except the last TIDU of a TSDU will be delivered completely filled.

The arrival of a TIDU of a TSDU (or the arrival of an ETSDU) is indicated to the receiving TS application by means of the TS event T_DATAIN (T_XDATIN). The TS application then fetches the TIDU (ETSDU) with a corresponding function call, either completely or in piecemeal fashion. If necessary it may or must issue several similar calls in order to take in one TIDU (ETSDU) from CMX.

The transfer of TIDUs (ETSDUs) is subject to flow control mechanisms, which can be controlled by CMX and the TS applications. The return code T_DATASTOP (T_XDATSTOP) returned when data is sent indicates to the sending TS application that the TIDU (ETSDU) was processed, but the flow of TIDUs (ETSDUs) has been blocked. No further TIDUs (ETSDUs) may be sent until the flow is released again. Release is indicated by means of the TS event T_DATAGO (T_XDATGO).

The receiving TS application stops and starts the flow of TIDUs (ETSDUs) by means of function calls to CMX, which affect the sending TS application as described above.

The following functions implement data exchange and (active) flow control:

t_datarq()
: Requests transfer of a TIDU (possibly partially filled) from a contiguous storage area. The return code T_DATASTOP signifies that the flow is blocked; further send requests are rejected with an error until the flow is released again.

t_vdatarq()
: Functions like *t_datarq*, but the TIDU can be located in multiple, non-contiguous storage areas.

t_datain()
: Accepts the data of a TIDU from CMX, placing it into a contiguous storage area, after the TIDU has been indicated with T_DATAIN. The return code specifies how much data is still contained in the current TIDU, so that a TIDU can be read in piecemeal fashion.

t_vdatain()
: Functions like *t_datain*, but the TIDU can be located in multiple, non-contiguous storage areas.

t_xdatrq()
: Requests transfer of an ETSDU (possibly partially filled); the return code T_XDATSTOP signifies that the flow is blocked; further send requests are then rejected with an error, until the flow is released again.

t_xdatin()
: Accepts the data of an ETSDU from CMX, after it has been indicated with T_XDATIN. The return code specifies how much data is still contained in the current ETSDU, so that an ETSDU can be read in piecemeal fashion.

t_datastop()
: Blocks, from the receiving side, the flow of normal data over a connection; the TS event T_DATAIN will no longer be indicated for this connection by CMX.

t_datago()
: Releases, on the receiving side, the (blocked) flow of normal data and expedited data over a connection; the TS events T_DATAIN and T_XDATIN can again be indicated for the connection by CMX.

t_xdatstop()

Blocks, on the receiving side, the flow of expedited data and normal data over a connection; CMX will no longer indicate the TS events T_XDATIN and T_DATAIN for this connection.

t_xdatgo()

Releases, on the receiving side, the (blocked) flow of expedited data over a connection; the event T_XDATIN can again be indicated by CMX for the connection.

Information service

The information service is a local service with which the TS application can query configuration-dependent parameter values from CMX. The information service is implemented with the following function:

t_info()

Returns the length of a TIDU for an established TC. The TIDU is normally only established when connection setup is completed.

Central waiting point

TS applications often expect application-specific events in addition to CMX events. In the callback routine, it is possible to wait for CMX events and application events simultaneously. A central waiting point be defined for this purpose.

t_callback()

passes a pointer to CMX indicating a routine in the application, which is called during the execution of the *t_event()* call.

Management options

Options can currently only be set in the CMX library.

t_setopt()

sets or cancels the trace options of the appropriate application.

8.2 States of TS applications and permissible state transitions

The sequences of operations at the ICMX(L) program interface are represented in the following diagram by means of finite-state automata. The diagram shows the defined states that a TS application may assume during the course of communication and the permissible transitions between these states. With the aid of the diagram it is possible to identify permissible sequences of CMX calls. The diagram shows when and how the processes of a TS application must react to certain events.

In the diagram each state is represented by a rectangle with a double border. The rectangle contains the name of the state.

The surrounding (outer) rectangles represent the three communication phases:

– 1st communication phase: Attach process
  The process exists, but is not yet or no longer attached to CMX.

– 2nd communication phase: Connection setup
  The process is attached to CMX, but no connection exists. A connection can now be set up.

– 3rd communication phase: Data transfer
  The connection has been set up. The process can now send and receive data.

The 3rd communication phase is subdivided by dotted lines into four subareas. These subareas are:

– Send normal data

– Receive normal data

– Send expedited data

– Receive expedited data

When it reaches this phase, at any given time the process is in exactly one state in each subarea. Only certain combinations of states in these subareas are permitted, i.e. a state transition within one subarea may cause a state transition in another subarea. The connections between the individual states in the various subareas can be seen by examining the conditions for state transitions (see below). If the exchange of expedited data has not been agreed for the connection, the process can only assume states of the upper two subareas.

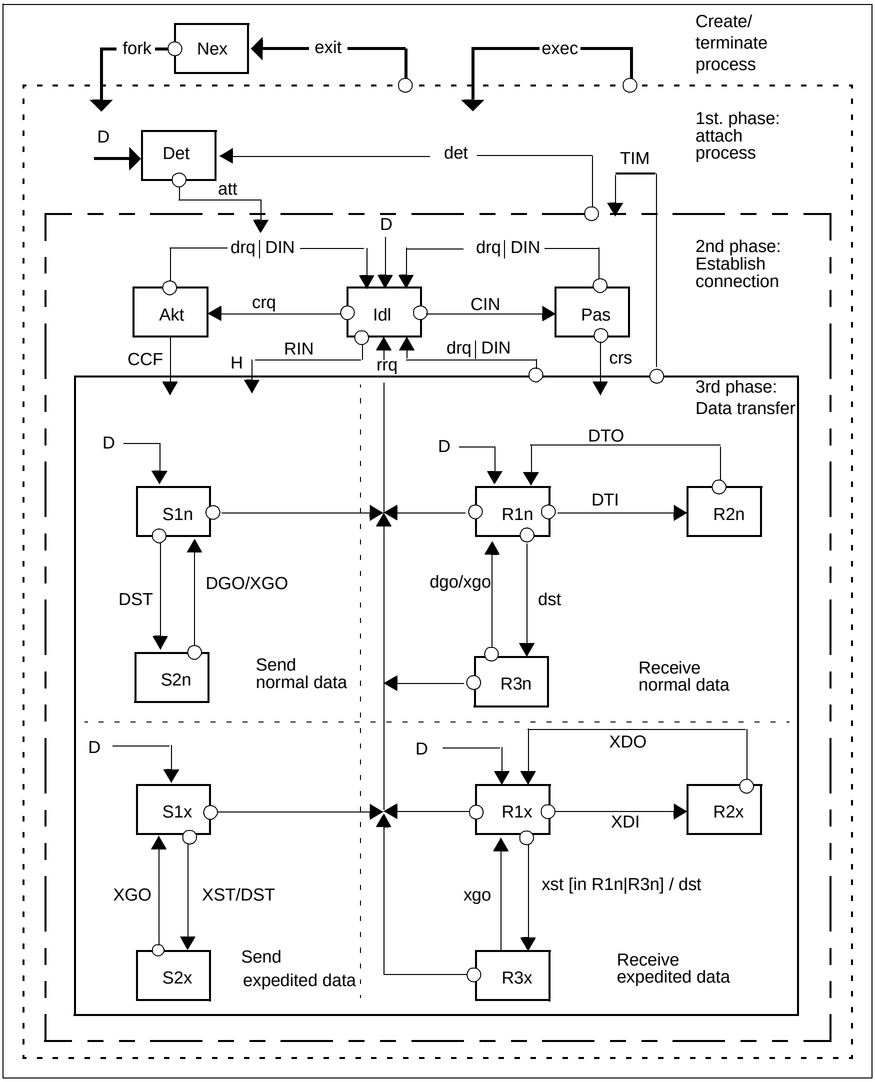


Figure 28: States of TS applications and permissible state transitions

The arrows between the rectangles indicate the possible state transitions. C indicates the condition for making the transition from an initial state to the subsequent state (initial state → subsequent state). Transitions are possible only in the directions indicated by the arrows.

Abbreviations for the states:

Nex	The process does not exist (no longer exists).
Det	The TS application is not yet attached to CMX, or the TS application has been detached from CMX.
Idl	Initial state for connection setup and for accepting a connection redirection, or a previously existing connection was closed down.
Act	Waiting for the event T_CONCF following a t_conrq() call (active connection setup).
Pas	A T_CONIN event has arrived (passive connection setup)
S1n	Initial state for t_datarq() or t_vdatarq()
S2n	Normal data flow blocked
R1n	Initial state for t_datain()
R2n	T_DATAIN indicated
R3n	T_DATAIN blocked
S1x	Initial state for t_xdatrq()
S2x	Flow of expedited data blocked
R1x	Initial state for t_xdatin()
R2x	T_XDATIN indicated
R3x	T_XDATIN blocked

Abbreviations for the state transition conditions

fork	Process created
exec	Process shift
exit	Process termination

The state transitions below occur when a CMX function is called:

att	t_attach()
det	t_detach()
crq	t_conrq()
crs	t_conrs()
drq	t_disrq()
rrq	t_redrq()
dst	t_datastop()
dgo	t_datago()
xst	t_xdatstop()
xgo	t_xdatgo()

The state transitions below occur when an event is accepted:

NET	T_NOEVENT
CIN	T_CONIN
CCF	T_CONCF
DIN	T_DISIN
RIN	T_REDIN
DTI	T_DATAIN
XDI	T_XDATIN
DGO	T_DATAGO
XGO	T_XDATGO

The following state transitions occur when certain return values are returned by CMX functions:

DST	T_DATASTOP returned by t_datarq() or T_vdatarq()
XST	T_XDATSTOP returned by t_xdatrq()
DTO	0 returned by t_datain() or t_vdatain()
	(current TIDU completely read)
XDO	0 returned by t_xdatin() (ETSDU completely read)
TIM	t_timeout (inactivity time limit for the connection reached)

8.2.1 Explanations of the possible state transitions

Arrows that terminate at a surrounding rectangle indicate that normally the process first switches to the states indicated by D→.

For example, in the transition to the 3rd communication phase (data transfer) the process initially switches to the states S1n, S1x, R1n, R1x.

An exception to this is the transition RIN H→. It means: When connection redirection occurs, the receiving process assumes the states in the 3rd phase (data transfer) that the redirecting process assumed in this phase prior to the redirection.

Arrows that begin at a surrounding rectangle indicate that a transition is possible from any given state within the rectangle.

State transitions of this kind are:

- fork
  If fork() is called in any state of the process, the child process assumes the state Det (process not yet attached to CMX). The state of the parent process remains unaffected.
- exec
  If exec() is called in any state of the process, the process switches to the state Det (process detached). It loses all attachments and connections.
- exit
  If exit() is called, the process is terminated. All connections are closed down by CMX.

– det
  If the process calls *t_detach()* in any state, it switches to the state Det. CMX closes down its connections.

– drq|DIN (drq or DIN)
  If the process calls *t_disrq()* in any state during data transfer (3rd phase) or during connection setup (2nd phase), the process switches to the state Idl. The same thing happens when CMX indicates the event T_DISIN to the process. The existing connection is closed down or the connection request of another TS application is rejected.

– TIM
  If during data transfer the inactivity time limit for the connection, specified by the parameter *t_timeout*, is exceeded, the process switches to the state Idl in the 2nd phase.

State transitions within the 3rd phase (data transfer)

The following describes the connections between state transitions in the subareas of the 3rd phase. The state assumed by a process in the subarea "Send normal data" depends on its state in the subarea "Send expedited data", and vice-versa. The state assumed by a process in the subarea "Receive normal data" depends on its state in the subarea "Receive expedited data", and vice-versa.

The following connections exist between the states of the four subareas:

DGO/XGO (DGO initiates XGO)

The event T_DATAGO initiates T_XDATGO. Along with normal data flow flow of expedited data is released, assuming it was blocked. Thus, the state transition S2n $\rightarrow$ S1n initiates the state transition S2x $\rightarrow$ S1x.

XST/DST (XST initiates DST)

The event T_XDATSTOP initiates the event T_DATASTOP. The state transition S1x $\rightarrow$ S2x brings about the state transition S1n $\rightarrow$ S2n. Blocking the expedited data flow causes blocking of normal data flow.

dgo/xgo (dgo initiates xgo)

If the process calls *t_datago()* in the state R3n (T_DATAIN blocked), *t_xdatgo()* is implicitly called. The state transition R3n $\rightarrow$ R1n initiates the state transition

R3x $\rightarrow$ R1x, if the process had previously assumed the state R3x.

xst[in R1n|R3n]/dst

If the process is in the state R1x, it may call *t_xdatstop()* only if it is in the state R1n or R3n in the subarea "Receive normal data". It thereby initiates *t_datastop()*. This means the flow of expedited data can be blocked by the process only so long as no T_DATAIN is indicated. Along with the flow of expedited data the flow of normal data is implicitly blocked (R1x $\rightarrow$ R3x initiates R1n $\rightarrow$ R3n).

8.3 Transport system specific features

This section describes the features of CMX-API which are specific to the transport system used.

Adjustable socket options for TCP/IP connections based on RFC1006

The CMXSOCKET environment variable allows you to activate the KEEPALIVE mechanism in TCP for all TCP connections to be set up in the corresponding process environment:

CMXSOCKET=-K1; export CMXSOCKET

If the KEEPALIVE mechanism is activated for a particular TCP connection but no data is transferred on this connection during the KEEPALIVE period, TCP uses test packets to ascertain whether or not the partner is still responding. If not, TCP automatically closes the connection locally and the CMX application receives a disconnect indication with the reason T_RLCONNLOST (loss of network connection).

The KEEPALIVE period is determined by the operating system. In Solaris it is defined in a system variable set to 2 hours.

In Solaris, you can change the value of the TCP variable *tcp_keepalive_interval* (specified in milliseconds) using the command
ndd -set /dev/tcp tcp_keepalive_interval <new value>.

Time monitoring of the answer to a connection setup request

All transport systems (with the exception of those working in local communications) have answer time monitoring whereby the time it takes the system to answer the protocol element making the connection request (e.g. the "call request packet" in X.25 and "CR TPDU" in IS 8073) is monitored.

The answer time is set in the protocol or on TSP-specific timers (default setting: 2 to 3 minutes). If the calling application does not receive an answer before the timer setting has elapsed, it will receive a CMX event T_DISIN with the connection disconnection indication T_RLNORESP. The protocol element requesting the disconnection will be sent to the partner system.

On some transport systems, the response time to a connection setup indication (CMX event T_CONIN) is also monitored. If the application does not reply with *t_conrs()* or *t_disrq()* within a preset time period, it will receive a T_DISIN with the disconnection indication T_RUNKNOWN or T_USER; this is irrespective of the transport system used and does not take into account whether or not a shorter timer was set on the local system or on the partner system.

Features influenced by the NEA protocol

In the case of NEA-TSP, if expedited data is sent three times but is not collected by the partner, the connection will be closed down.

8.4 System options and message length

It is important to note when creating TS applications that the system options to "exchange user data when setting up and closing down a connection" and to "exchange expedited data" are not supported by every transport system (CCP profile). Moreover, in transport systems that do support these system options, the permitted length of the user data or the expedited data unit are different.

The Release Notes provide information on which CCP profiles support these system options and details on the supported length for user data, expedited data and messages.

8.4.1 Programming notes

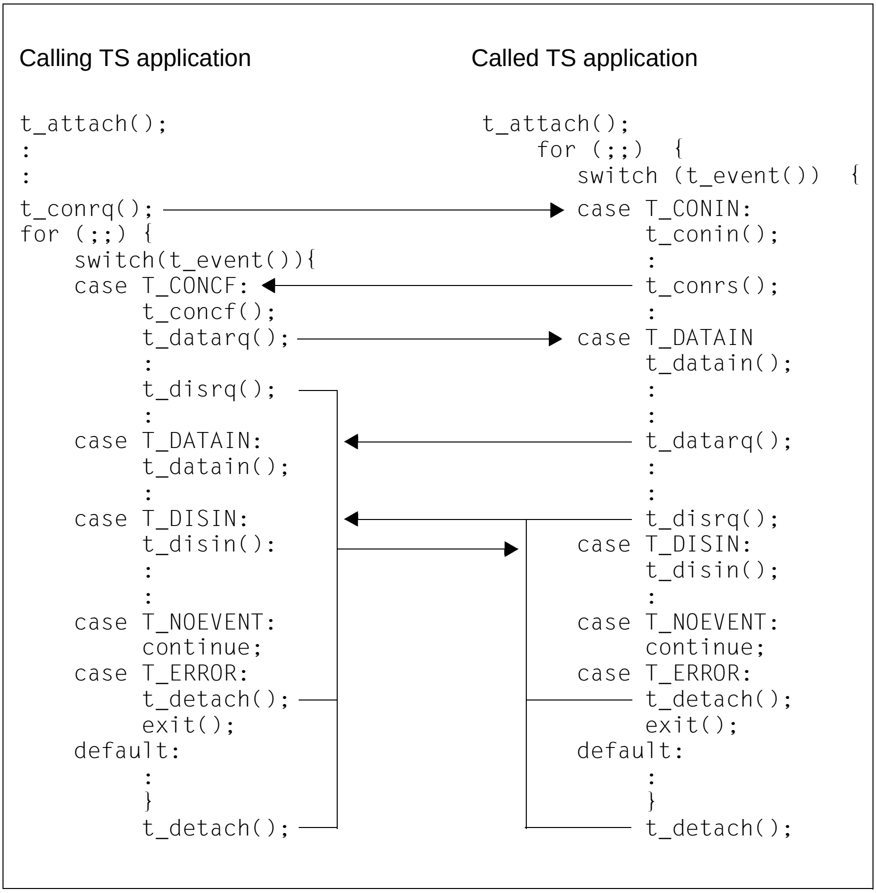
The primary purpose of ICMX(L) is to make TS applications independent of the transport systems used. This allows TS applications to execute in a variety of network environments. ICMX(L) supports this independence for TS applications that adhere to the following rules:

1. The application should make no explicit assumptions regarding the length of a TIDU or regarding the way TIDUs are packed for communication.
2. The limits defined in <*cmx.h*> for the options must never be exceeded. It is to be noted that some transport systems do not provide certain options.
3. The TS application should handle addressing exclusively with the aid of the TNSX; it should not construct any physical transport addresses in the programs.
4. CMX functions should not be called in signal handling routines; signal handling is not suitable for performing asynchronous processing outside the current context.
5. Function prototyping is supported by CMX. Thus, the program is notified whether or not the transferred parameter structure is correct during compilation.

ICMX is designed such that program runs can be event-driven. It is specially designed to allow an event loop to be programmed where special consideration is given to the individual events.

Example

An event-driven design for two TS applications is shown in the following program example.



Calling TS application

```
t_attach();
:
:
t_conrq();
for (;;) {
    switch(t_event()){
    case T_CONCF:
         t_concf();
         t_datarq();
         :
         t_disrq();
         :
    case T_DATAIN:
         t_datain();
         :
    case T_DISIN:
         t_disin():
         :
         :
    case T_NOEVENT:
         continue;
    case T_ERROR:
         t_detach();
         exit();
    default:
         :
         }
         t_detach();
```

Called TS application

```
t_attach();
    for (;;)  {
       switch (t_event())  {
       case T_CONIN:
            t_conin();
            :
            t_conrs();
            :
       case T_DATAIN
            t_datain();
            :
            :
            t_datarq();
            :
            :
            t_disrq();
       case T_DISIN:
            t_disin();
            :
       case T_NOEVENT:
            continue;
       case T_ERROR:
            t_detach();
            exit();
       default:
            :
            }
            t_detach();
```

Figure 29: Event-driven design of two TS applications

8.4.2 Additional functionality "Operation without TNS/Creation of templates"

Using the functions *t_getloc()* and *t_getaddr()* you may generate so-called templates, i.e. LOCAL NAMES and TRANSPORT ADDRESSES with an empty content, by specifying the value NULL in the parameter *globname* and the value OPTG7 in the parameter *t_optnr* within the structure *t_optg7*. Using the functions *t_getlocpart()* or *t_getaddrpart()* the application itself then assures the representation of this LOCAL NAME or this TRANSPORT ADDRESS in a data structure (struct t_addrpart) of the application.

Using the functions *t_setlocpart()* and *t_setaddrpart()* the contents of this address which was modified by the application can be changed temporarily within the memory of the application and can then be passed to the function *t_attach()* in the parameter *name* or to the function *t_conrq()* in the parameters *fromaddr* and *toaddr*.

8.4.2.1 Application scenario / Program skeleton

```
struct t_myname MN, newMN;
struct t_partaddr PA newPA;

MN=t_getloc(NULL,..);              /* Get a template of a LOCAL NAME */
        :
t_getlocpart(MN,..);                         /* Split the template into
                                              a structure t_addrpart */
/* Set program-specific information in the structure t_addrpart */

t_setlocpart(MN,newMN..);            /* Generate the LOCAL NAME newMN
                                                filled with contents */
t_attach(newMN,..)

PA=t_getaddr(NULL,..);                             /* Get template of a
                                                   TRANSPORT ADDRESS */
        :
t_getaddrpart(PA,..) ;                     /* Split the template into a
                                                structure t_addrpart */
/* Set program-specific information in the structure t_addrpart */

t_setaddrpart(PA,newPA..) ;     /* Generate the TRANSPORT ADDRESS newPA
                                                filled with contents */
t_conrq(newMN,newPA);
```

8.5 Conventions

When using ICMX(L) the following conventions must be observed:

1. All identifiers starting with "_" (underscore) are reserved for the system software.
2. All identifiers starting with "t_" or "ts" or "Ts" are reserved for CMX.
3. All preprocessor definitions starting with "T_" or "TS_" are reserved for CMX.
4. At the request of the user, signals (usually SIGIO and/or SIGTERM) are sent (by CMX components in the kernel) and intercepted in the CMX library. User-defined signal routines should therefore be programmed with caution.

8.6 ICMX(L) - function calls

The following pages describe the CMX calls in detail. Italic type in running text represents ordinary, replaceable formal parameters or the names of functions and files. Names in uppercase letters (e.g. T_MSGSIZE) represent constants that have been defined in a header file (with #define).

The following conventions are used in the parameter descriptions:

–> Indicates a parameter in which CMX expects a value provided by the caller.

<– Indicates a parameter in which CMX returns a value after the call.

<> Indicates a parameter in which the caller must provide a value, which is then modified by CMX.

Modification generally only takes place if processing was successful. If it was unsuccessful the value remains unchanged.

Of course, if a parameter involves a pointer, this marking does not refer to the pointer itself (which is always provided by the caller), but instead to the contents of the field to which the pointer points.

In all cases, for values to be returned by CMX appropriate storage space must be provided by the caller and a pointer must be passed to CMX.

8.6.1 t_attach - Attach a process to CMX (attach process)

t_attach() attaches the current process to CMX. The parameters passed in the *t_attach()* call specify:

– the TS application for which the process is being attached,

– the types of connection setup (passive, active, acceptance of a redirected connection) that are possible for the process in this TS application,

– the number of connections the process may have simultaneously in this TS application.

The TS application for which the process is being attached has a GLOBAL NAME and one or more T-selectors that are unique in the local system. The T-selectors combine to form the LOCAL NAME. The LOCAL NAME must be passed to CMX as a parameter of the *t_attach()* call. With the help of the call *t_getloc()* and the GLOBAL NAME of the TS application the LOCAL NAME can be queried from the TNSX and placed in a data area. A pointer to this data area is then passed in the *t_attach()* call.

Using repeated *t_attach()* calls, the current process may attach itself to CMX for several different TS applications.

Likewise, several different processes may attach themselves to CMX for the same TS application, i.e. using the same LOCAL NAME. The first process to attach itself for a TS application generates the TS application.

CMX accepts connection requests for a TS application from the network as soon as a process of the TS application has attached itself to CMX for the acceptance of connection indications, i.e. when T_PASSIVE is specified in *t_apmode*.

If more than one process has attached itself for a TS application with T_PASSIVE, CMX initially delivers all connection indications for the TS application to the process that first attached itself for the TS application with T_PASSIVE. Only when the maximum number of connections that this process may have for the TS application is attained are arriving connection indications delivered to one of the other processes. The order in which this is done is not defined.

Notes

At the first *t_attach()*, a file descriptor is assigned in the running process. This file descriptor remains assigned for the life of the process.

T_OK with several T-selectors means that the attachment was successful for at least one T-selector.

```
#include <cmx.h>
int   t_attach (const struct t_myname *name,
                t_opta *opt);
```

-> name

For *name*, specify a pointer to a structure *t_myname* with the LOCAL NAME of the TS application. The LOCAL NAME is returned by the TNSX as a property of the GLOBAL NAME of the TS application.

<> opt

For the parameter *opt*, specify the value NULL or a pointer to a union with user options.

If *opt* = NULL is specified, CMX uses the given default values.

The following structures are defined in <*cmx.h*>:

```
      struct t_opta1 {
->       int t_optnr;    /* Option number */
->       int t_apmode;   /* Process mode */
->       int t_conlim;   /* Number of connections */
      }

      struct t_opta2 {
->       int t_optnr;    /* Option number */
->       int t_apmode;   /* Process mode */
->       int t_conlim;   /* Number of connections */
->       int t_uattid;   /* User attachment reference */
<-       int t_attid;    /* CMX attachment reference */
<-       int t_ccbits;   /* Bit list of CCs affected */
<-       int t_sptypes;  /* Address formats affected */
      }
      struct t_opta5 {
->       int t_optnr;              /* Option number */
->       int t_apmode;             /* Process mode */
->       int t_conlim;             /* Number of connections */
->       int t_uattid;             /* User attachment reference */
<-       int t_attid;              /* CMX attachment reference */
<-       int t_ccbits;             /* Bit list of CCs affected */
<-       int t_sptypes;            /* Address formats affected */
<-       int t_evref;              /* Reference point */
      }

      struct t_opta6 {
->       int t_optnr;              /* Option number */
->       int t_apmode;             /* Process mode */
->       int t_conlim;             /* Number of connections */
->       int t_uattid;             /* User attachment reference */
<-       int t_attid;              /* CMX attachment reference */
<-       struct t_cclst *t_cclist; /* Address of the CC list */
<-       int t_sptypes;            /* Address formats affected */
      }

      struct t_opta7 {
->       int t_optnr;               /* Option number */
->       int t_apmode;              /* Process mode */
->       int t_conlim;              /* Number of connections */
->       int t_uattid;              /* User attachment reference */
<-       int t_attid;               /* CMX attachment reference */
<-       struct t_cclst *t_cclist;  /* Address of the CC list */
<-       int t_sptypes;             /* Address formats affected */
<-       int t_evref;               /* Reference point */
->       char *t_hostname;         /* Host name that corresponds to
                                      the IP address of a local
                                       IP interface */
      }
```

t_optnr

Option number. Specify:

T_OPTA1 in t_opta1

T_OPTA2 in t_opta2

T_OPTA5 in t_opta5

T_OPTA6 in t_opta6

T_OPTA7 in t_opta7

t_apmode

t_apmode specifies the types of connection setup possible for the process in this TS application.

Permissible values are:

T_ACTIVE
: The process is to actively set up connections.

T_PASSIVE
: The process is to wait passively for requests to set up connections.

T_REDIRECT
: The process should accept redirected connections.

These values may be combined using bitwise OR (**|**), e.g. T_ACTIVE **|** T_PASSIVE.

Default value specifying *opt* = NULL:
T_ACTIVE **|** T_PASSIVE **|** T_REDIRECT

t_conlim

For *t_conlim*, specify the maximum number of simultaneous connections that the process can maintain in this TS application.

If *t_conlim* = T_NOLIMIT is specified, the process can maintain the installation-specific maximum number of simultaneous connections.

Default value specifying *opt* = NULL: 1

t_uattid

In the field *t_uattid* you can pass CMX any user reference desired for this application. This user reference will be subsequently returned by CMX as an option in *t_event*, i.e. when the current process queries CMX regarding the arrival of an event.

This user reference enables a process that controls multiple TS applications to more easily associate an arriving event with the appropriate attachment.

Default value specifying *opt* = NULL: 0

t_attid

This field serves trace and diagnostic purposes. It is used exclusively for logging.

In the *t_attid* field CMX returns the CMX-internal reference to the attachment.

t_ccbits

This field serves trace and diagnostic purposes. It is used exclusively for logging.

The meaning of the list may be obtained from *<cmx.h>*.

t_sptypes

This field serves trace and diagnostic purposes. It is used exclusively for logging.

In *t_sptypes* CMX returns a bit-encoded list of the address formats for which this attachment was successful.

The meaning of the list may be obtained from *<cmx.h>*.

t_evref

Event reference point. This is permitted only for compatibility with CMX on BS2000/OSD and is not supported in CMX on UNIX (Solaris and Reliant UNIX).

t_cclist

This pointer to a CC list is used for tracing and diagnostic purposes. It is used exclusively for logging.

In *t_cclist*, CMX provides a pointer to a CC list of the address formats for which this attachment was successful.

The values are explained in <cmx.h>.

t_hostname

This field is required for TS applications which run in a cluster configuration. The IP address of the corresponding IP interface is used for the „fromaddr“ parameter during active connection setup with *t_conrq()*. In the case of passive connection setup, only connection requests which arrive via this IP interface wil be displayed.

This parameter is effective only if the address format RFC1006 or LANINET is specified in the LOCAL NAME of the TS application. When a process attaches to a TS application for the first time and when it attaches to the same TS application a subsequent time using the same host name, there are no restrictions with respect to address formats other than RFC1006 and LANINET. However, if a process wishes to use a different host name, it can only attach to TS applications that use the RFC1006 and LANINET address formats exclusively. In this case, the return value is T_OK and **not** T_NOTFIRST.

Return values

T_OK
: The call was successful. The process was the first to attach itself with this name.

T_NOTFIRST
: The call was successful. However, the process was not the first to attach itself for this TS application.

T_ERROR
: Error. Error code can be queried using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_ENOENT
: All allocated resources are already occupied.

T_EFAULT
: At least one of the pointers *name* or *opt* (!= NULL) does not point to the process address space.

T_WPARAMETER
: The LOCAL NAME passed with *name* or one of the options specified in *opt* has an invalid format or contains illegal values.

T_WAPPLICATION
: The process is already attached for the TS application specified in *name*, or the LOCAL NAME that was specified in *name* is already being used by an XTI process.

T_WAPP_LIMIT
: The process has already attached itself for all applications available to it, or the maximum number of TS applications has been reached.

T_WPROC_LIMIT
: The maximum number of processes that CMX can use has been reached.

T_NOCCP
: No suitable CCP is available (at present) for the LOCAL NAME specified in *name*.

T_WLIBVERSION
: The version of the CMX library linked into the process is incompatible with the operating system version.

See also

t_detach(), t_event(), t_error(), t_getloc()

8.6.2 t_callback - Register a callback routine

Using *t_callback*, an application can attach its own function for event handling in ICMX(L), which are then called when *t_event* is executed. This function is referred to as a callback function, because it is called by an ICMX(L) function (*t_event()*) that belongs to the application.

The callback routine enables the application to use *t_event()* to simultaneously wait not only for TS events, but also for other events that are important for the application (e.g. terminal I/O). A prerequisite here is that the event sources can be described by means of file descriptors so that the callback routine can use the system functions *select()* or *poll()* (see Solaris Programmer's Reference Guide) for checking events.

The callback routine is used as follows: when *routine* is called, the function *t_event()* passes bit lists containing file descriptors, which are used internally by ICMX(L) and for which events are outstanding. The transfer is the same as for the *select()* call. The function routine can now add application-specific file descriptors and can call the function *select()* with the completed bit lists. This makes it possible to simultaneously wait for TS events and other application-specific events. If an event occurs, *routine* must check whether or not the event was application-specific. If so, *routine* ends with T_USEREVENT; otherwise, it ends with T_TSEVENT. Before *routine* terminates, the routine can continue processing an application-specific event. However, *routine* may never enter a wait state that cannot be interrupted by signals.

The behavior of *t_event* depends on the return value of *routine*. If a TS event has not occurred, *t_event()* returns to the application with T_NOEVENT. Otherwise, the TS event is displayed.

A callback routine enables the application to optimize event handling according to your own criteria. If it is not implemented correctly, the handling of TS events by *t_event()* may not be reliable. You should therefore follow the implementation guidelines listed below.

The functional sample of *routine* is similar to the function *select()* and looks like this:

```
#include <sys/select.h>
#include <cmx.h>
int   (*t_cbtype) routine (int fdsetsize,
                           fd_set *rfds,
                           fd_set *wfds,
                           fd_set *xfds,
                           struct timeval *time,
                           const void *usr);
```

-> fdsetsize
: Number of file descriptors that are used by ICMX(L) internally.

<> rfds
: Pointer to a bit list of file descriptors for which ICMX(L) expects read events internally. The return list must contain at least the file descriptors of ICMX(L) for which a read event has occurred. File descriptors that are not affected are ignored in the list.

<> wfds
: Pointer to a bit list of file descriptors for which ICMX(L) expects write permission internally. The return list must contain at least the file descriptors of ICMX(L) for which this type of event has occurred. File descriptors that are not affected are ignored in the list.

<> xfds
: Pointer to a bit list of file descriptors for which ICMX(L) expects exceptional conditions to occur. The return list must contain at least the file descriptors of ICMX(L) for which this type of event has occurred. File descriptors that are not affected are ignored in the list.

-> time

Specifies the maximum time spent waiting for an event to occur. The value 0 indicates that *routine* can check whether an event exists, but cannot enter a wait state. The value -1 means that *routine* should wait for the event for an unlimited period of time. The *time* value is derived from the *t_timeout* value when *t_event()* is called.

Please note that *routine* must support the value 0 at all times, even if the application never calls *t_event()* in T_CHECK mode.

-> usr

Pointer that was passed to *t_callback* by the application when *routine* was attached (see below). The contents are not checked by ICMX(L). The application can pass application-specific information to *routine* via *usr*.

Return value

T_NOEVENT

Neither a user event nor a TS event has occurred within *time*. The function was interrupted by a signal or an internal error occurred.

In this case, *t_event()* also terminates with T_NOEVENT.

T_TSEVENT

A TS event has occurred. *t_event()* then checks which TS event occurred and terminates with this event.

If *t_event()* cannot find a TS event, it terminates with T_NOEVENT.

T_USEREVENT

An application-specific event has occurred. *t_event()* terminates either with T_NOEVENT if *t_event()* does not detect a TS event, or it reports the TS event that *t_event()* detected outside *routine*.

Errors

If *routine* has to terminate prematurely because of an internal error, it must terminate with T_NOEVENT. You should take note of the error status internally, so that the application can take appropriate action when *t_event()* has terminated.

Implementation guidelines

- The ICMX(L)-specific file descriptors may not be distorted by *routine*. TS events can only be identified if these file descriptors are passed to *select()* or *poll()*. If the file descriptors are distorted while *routine* is running, *t_event()* is no longer reliable.
- The callback routine may only enter wait states that can be interrupted by all signals.
- The callback routine may not call *t_event()*. *t_event()* rejects the recursive call with T_CBRECURSIVE.
- The callback routine is Solaris-specific, as it explicitly uses file descriptors. The concept is therefore not provided in ICMX(L) implementations in BS2000/OSD and MS-DOS.

Information on the callback routine is reported to ICMX(L) with *t_callback*:

```
#include <cmx.h>
t_cbtype   t_callback (t_cbtype routine,
                       const void *usr,
                       const void *opt);
```

-> routine

Pointer to the callback routine that is to be called by *t_event()*. A callback routine that was attached to ICMX(L) is detached again with NULL.

-> usr

Pointer to an application-specific data area that is not checked by ICMX(L). The pointer is passed to the callback routine by *t_event()* when this is called.

-> opt

Reserved for future extensions. The value must be NULL.

Return value

T_OK

The pointer to the old callback routine is returned. The return value "NULL" means that a callback routine was not included.

T_ERROR

Error. Error code can be queried with *t_error()*.

Errors

In the event of an error the following error values are possible. They can be queried by calling *t_error()*.

The following can occur for error type T_CMXTYPE and error class T_CMXCLASS:

T_WPARAMETER
: *opt* is not NULL.

T_WSEQUENCE
: The process has not been attached in any TS application.

See also

t_event()

8.6.3 t_concf - Establish connection (connect confirmation)

t_concf() accepts a T_CONCF event from CMX previously reported with *t_event()*. T_CONCF indicates that the called TS application has positively answered a connection request (*t_conrq()* call) of the current process.

t_concf() returns:

– The user data that the called TS application included, if the transport system used provides this option.

– The answer of the called TS application if the current process proposed the exchange of expedited data when issuing the connection request *t_conrq()*.

If the *t_concf()* call is successful the connection is established for the current process. As soon as a connection is established, the TS application (not CMX) has the initiative. It may:

– send normal data and (if agreed) expedited data, or

– indicate, through *t_event()*, that it is ready to receive normal data or (if agreed) expedited data, or redirect or close down the connection.

```
#include <cmx.h>
int   t_concf (const int *tref,
               t_opt1 *opt);
```

-> tref

Pointer to a field with the transport reference of the connection, passed to the current process via *t_event()*.

<> opt

For *opt*, specify the value NULL or a pointer to a union containing a structure with system options.

This union is used to receive the user data that the called TS application included with its answer to the connection request.

If *opt* = NULL is specified, CMX discards the user data and options.

If the called TS application specified no user data and no options, CMX uses the given default values.

The following structure is defined in <*cmx.h*>:

```
   struct t_optc1 {
->    int  t_optnr;      /* Option no. */
<-    char *t_udatap;    /* Data buffer */
<>    int  t_udatal;     /* Length of the data buffer */
<-    int  t_xdata;      /* Choice for expedited data */
<-    int  t_timeout;    /* Inactive time */
   };
```

t_optnr
Option number. Specify T_OPTC1.

t_udatap
Pointer to a data area in which CMX enters the user data received from the called TS application.

Default value specifying *opt* = NULL: Undefined

t_udatal
Prior to the call 0 or the length of the data area *t_udatap* must appear here. The area must be large enough to hold the received data completely. The maximum length required depends on the transport system used.

T_MSG_SIZE is the maximum size suitable for all transport systems. T_MSG_SIZE is defined in <*cmx.h*>. After the call, CMX returns in this field the number of bytes placed in *t_udatap*.

Default value specifying *opt* = NULL: 0

t_xdata
CMX returns here the answer of the called TS application if the exchange of expedited data was proposed at connection setup. The answer is binding. Possible answers:

T_YES
The called TS application accepts the proposal.

T_NO
The called TS application rejects the proposal.

Default value specifying *opt* = NULL: T_NO

t_timeout
This field always contains T_NO.

Return values

T_OK
: The call was successful.

T_ERROR
: Error. Error code can be queried using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the pointers *opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or no T_CONCF was indicated on the connection specified by *tref*.

T_WPARAMETER
: The options specified in *opt* have an invalid format or contain illegal values, or the buffer for the data to be received is too small.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_conrq(), t_error(), t_event()

8.6.4 t_conin - Receive connection request (connect indication)

t_conin() accepts a T_CONIN event previously reported with *t_event()*. T_CONIN indicates that a calling TS application wishes to set up a connection to the current process.

The call returns:

– the TRANSPORT ADDRESS of the calling TS application,

– the LOCAL NAME of the local TS application, and

– the user data that the calling TS application included.

Subsequently the connection request may be answered (confirmed) with *t_conrs()* or rejected with *t_disrq()*.

```
#include <cmx.h>
int   t_conin (const int *tref,
               union t_address *toaddr,
               union t_address *fromaddr,
               t_opt1 *opt);
```

-> tref
: Pointer to a field with the transport reference of the connection, passed to the current process via *t_event()*.

<- toaddr
: Pointer to a union *t_address* in which CMX returns the LOCAL NAME of the called TS application that is to receive the connection. If the current process is attached for multiple TS applications, with the aid of this information the connection request can be associated with the correct TS application.

<- fromaddr
: Pointer to a union *t_address* in which CMX returns the TRANSPORT ADDRESS of the calling TS application. The TRANSPORT ADDRESS can be converted to the GLOBAL NAME of the calling TS application with the aid of the call *t_getname()*.

Notes

If an RFC1006 type TRANSPORT ADDRESS is received during communication via RFC1006 over TCP/IP, the address may not be passed in binary form to *t_conrq()* in a subsequent active connection setup, because the internal address components that are evaluated by *t_conrq()* are missing. The application must query the GLOBAL NAME using *t_getname()*, and then query the address again using *t_getaddr()*.

<> opt

For *opt*, specify the value NULL or a pointer to a union containing a structure with system options.

This union is used to fetch the user data that the calling TS application specified at connection setup.

If *opt* = NULL is specified, CMX discards the user data and options.

If the calling TS application specified no user data and no options in *t_conrq()*, CMX returns the specified default values.

The following structure is defined in <*cmx.h*>:

```
   struct t_optc1 {
->    int  t_optnr;     /* Option no. */
<-    char *t_udatap;   /* Data buffer */
<>    int  t_udatal;    /* Length of the data buffer */
<-    int  t_xdata;     /* Choice for expedited data */
<-    int  t_timeout;   /* Inactive time */
   };
```

t_optnr

Option number. Specify T_OPTC1.

t_udatap

Pointer to a data area in which CMX enters the user data received from the calling TS application.

Default value specifying *opt* = NULL: Undefined

t_udatal

Prior to the call 0 or the length of the data area *t_udatap* must appear here.

The area must be large enough that the received data completely fits. The maximum length required depends on the transport system used. T_MSG_SIZE is the maximum size suitable for all transport systems. T_MSG_SIZE is defined in <*cmx.h*>.

After the call, CMX returns in this field the number of bytes placed in *t_udatap*.

Default value specifying *opt* = NULL: 0

t_xdata

In this field CMX returns the proposal of the calling TS application regarding expedited data.

Possible answers:

T_YES

The calling TS application proposes exchanging expedited data.

T_NO

The exchange of expedited data is ruled out by the calling TS application.

If the calling TS application proposes exchanging expedited data (T_YES), the answer of the current process in the subsequent *t_conrs()* is final.

If the calling TS application desires no expedited data (T_NO), none can be requested by the current process in the subsequent *t_conrs()*. It may then be necessary for the current process to reject the connection request with *t_disrq()*.

Default value specifying *opt* = NULL: T_NO

t_timeout

This field always contains T_NO.

Return values

T_OK

The call was successful.

T_ERROR

Error. The error code can be queried using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT

At least one of the pointers *toaddr, fromaddr, opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or no T_CONIN was indicated on the connection specified by *tref*.

T_WPARAMETER
: The options specified in *opt* have an invalid format or contain illegal values, or the buffer for the data to be received is too small.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_attach(), t_conrs(), t_conrq(), t_disrq(), t_error(), t_event(), t_getname()

8.6.5 t_conrq - Request connection (connection request)

t_conrq() requests the establishment of a transport connection from the local TS application to a called TS application (active connection setup).

More specifically, the effects of *t_conrq()* are:

– The called TS application receives the event T_CONIN as a connection indication, to which it must respond.

  The answer of the called TS application is later indicated to the current process by CMX in a *t_event()* call as event T_CONCF or T_DISIN.

– The called TS application may be sent user data along with the connection request, if the transport system used provides this option.

```
#include <cmx.h>
int   t_conrq (int *tref,
               const union t_address *toaddr,
               const union t_address *fromaddr,
               const t_opt13 *opt);
```

<- tref

Pointer to a field in which CMX returns the connection-specific transport reference. This uniquely identifies the connection in the subsequent communication phases. It must therefore be specified with all calls that involve this connection.

-> toaddr

Pointer to a union *t_address* with the TRANSPORT ADDRESS of the called TS application. The TRANSPORT ADDRESS is returned by the TNSX as a property of the GLOBAL NAME of the called TS application. It can be obtained from the TNSX using a *t_getaddr()* call.

-> fromaddr

Pointer to a union *t_address* with the LOCAL NAME of the calling TS application. The same LOCAL NAME must be specified here as was specified in *t_attach()* for this TS application.

-> opt

For *opt*, specify the value NULL or a pointer to a union with system options. This is used to specify the user data and options that the called TS application is to receive with the connection indication.

If *opt* = NULL is specified, CMX uses the given default values.

The following structures are defined in <*cmx.h*>:

```
struct t_optc1 {
->     int  t_optnr;      /* Option no. */
->     char *t_udatap;    /* Data buffer */
->     int  t_udatal;     /* Length of the data buffer */
->     int  t_xdata;      /* Choice for expedited data */
->     int  t_timeout;    /* Inactive time */
   };
   struct t_optc3 {
->     int  t_optnr;      /* Option no. */
->     char *t_udatap;    /* Data buffer */
->     int  t_udatal;     /* Length of the data buffer */
->     int  t_xdata;      /* Choice for expedited data */
->     int  t_timeout;    /* Inactive time */
->     int  t_ucepid;     /* User connection reference */
   };
```

t_optnr

Option number. Specify:

T_OPTC1 in *t_optc1*

T_OPTC3 in *t_optc3*

t_udatap

Pointer to a storage area containing user data that the called TS application is to receive with the connection indication.
Default value specifying *opt* = NULL: Undefined

t_udatal

Length of the user data, in bytes, to be transferred from the area *t_udatap*. If 0 is specified for *t_udatal*, *t_udatap* is ignored. The maximum value for *t_udatal* depends on the transport system (see the Release Notes).

Default value specifying *opt* = NULL: : 0

t_xdata

In the *t_xdata* parameter the current process informs the called TS application as to whether it is ready to exchange expedited data. Permissible values are:

T_YES

Exchange of expedited data proposed.

T_NO

Exchange of expedited data ruled out.

Default value specifying *opt* = NULL: T_NO t_timeout

For *t_timeout*, specify the inactive time for the connection. The inactive time specifies how long the connection may be inactive before it will be closed down by CMX.
It begins only when all data has been retrieved.

Possible specifications:

T_NO

The inactive time of the connection will not be monitored.

n > 0

The connection may be inactive for n seconds. Thereafter CMX will close it down.

Default value specifying *opt* = NULL: : T_NO.

t_ucepid

This field can be used to pass a freely-selectable user reference for this connection to CMX. This user reference can be returned to the current process by CMX as an option in a *t_event()* call. If the current process is maintaining multiple connections this mechanism enables it to associate a TS event with the appropriate connection via a user-defined attribute. The user reference constitutes an alternative to the transport reference *tref*, defined by CMX.

Default value specifying *opt* = NULL: 0

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the pointers *toaddr, fromaddr, opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or the process has not set T_ACTIVE in *t_apmode* for the local TS application, specified in *fromaddr*.

T_WPARAMETER
: The TRANSPORT ADDRESS passed with *toaddr* or the LOCAL NAME passed with *fromaddr* or one of the options specified in *opt* has an invalid format or contains illegal values.

T_WAPPLICATION
: The process is not attached for the TS application that has the LOCAL NAME passed with *fromaddr*.

T_WCONN_LIMIT
: The process has already used the number of connections specified for this TS application in *t_attach()* (*t_conlim* parameter) or the system limit for connections has been exceeded.

T_NOCCP
: The TRANSPORT ADDRESS specified in *toaddr* is not supported by any (currently) operational CCP or the LOCAL NAME specified in *fromaddr* contains no information for this CCP.

T_ETIMEOUT
: The CCP does not respond in the time limit.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_attach(), t_error(), t_event(), t_getaddr()

8.6.6 t_conrs - Respond to connection request (connection response)

t_conrs() is used by the called TS application to accept (confirm) the connection request of a calling TS application, the connection request having been previously indicated to the current process in *t_event()*, with the event T_CONIN. The current process must accept the T_CONIN event with *t_conin()* (passive connection setup) before calling *t_conrs()*. The calling TS application receives this response as connection confirmation with the event T_CONCF.

With *t_conrs()*

– information can be sent to the calling TS application, if the transport system used provides this option;

– the connection is completely set up for the current process.

As soon as a connection has been established, the TS application (not CMX) has the initiative. It may:

– send both normal data and (if agreed) expedited data, or

– indicate, via *t_event()*, that it is prepared to receive normal data or (if agreed) expedited data, or

– close down or redirect the connection.

```
#include <cmx.h>
int   t_conrs (const int *tref,
               const t_opt13 *opt);
```

-> tref

Pointer to a field with the transport reference for the connection used in the corresponding *t_conin()*.

-> opt

For *opt*, specify the value NULL or a pointer to a union with system options.

This is used by the current process to pass the user data that the calling TS application is to receive with the response to the connection request. If *opt* = NULL is specified, CMX uses the given default values.

The following structures are defined in <*cmx.h*>:

```
struct t_optc1 {
->    int  t_optnr;    /* Option no. */
->    char *t_udatap;  /* Data buffer */
->    int  t_udatal;   /* Length of the data buffer */
->    int  t_xdata;    /* Choice for expedited data */
->    int  t_timeout;  /* Inactive time */
   };
   struct t_optc3 {
->    int  t_optnr;    /* Option no. */
->    char *t_udatap;  /* Data buffer */
->    int  t_udatal;   /* Length of the data buffer */
->    int  t_xdata;    /* Choice for expedited data */
->    int  t_timeout;  /* Inactive time */
->    int  t_ucepid;   /* User connection reference */
   };
```

t_optnr

Option number. Specify:

T_OPTC1 in *t_optc1*

T_OPTC3 in *t_optc3*

t_udatap

Pointer to a storage area containing user data that the calling TS application is to receive.

Default value specifying *opt* = NULL: Undefined

t_udatal

Length of the user data, in bytes, to be transferred from the area *t_udatap*. If 0 is specified for *t_udatal*, *t_udatap* is ignored. The maximum value for *t_udatal* depends on the transport system (see the Release Notes).

Default value specifying *opt* = NULL: 0

t_xdata

In *t_xdata* the current process responds to the proposal of the calling TS application regarding the exchange of expedited data. The proposal was passed to the process via the *t_conin()* call.

Permissible values are:

T_YES

The proposal of the calling TS application regarding expedited data is accepted.

T_NO

Expedited data is refused.

The response is binding.

If the calling TS application had ruled out the use of expedited data, the response here must be T_NO.

Default value specifying *opt* = NULL: T_NO

t_timeout

For *t_timeout*, specify the inactive time for the connection. The inactive time specifies how long the connection may be inactive before it will be closed down by CMX.

T_NO

Inactive time will not be monitored.

n > 0

The connection may be inactive for n seconds. Thereafter CMX will close it down.

Default value specifying *opt* = NULL: T_NO.

t_ucepid

This field can be used to pass a freely-selectable user reference for this connection to CMX.

This user reference can be returned to the current process by CMX as an option in a *t_event()* call.

If the current process is maintaining multiple connections this mechanism enables it to associate a TS event with the appropriate connection via a user-defined attribute. The user reference constitutes an alternative to the transport reference *tref*, defined by CMX.

Default value specifying *opt* = NULL: 0

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the pointers *opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or the call was not preceded by a successful *t_conin()* call.

T_WPARAMETER
: The options specified in *opt* have an invalid format or contain illegal values.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.

  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_conin(), t_error(), t_event()

8.6.7 t_datago - Release the flow of data (data go)

t_datago() releases the blocked flow of data on the specified connection. By means of this call the current process informs CMX that it is again ready to receive data. This call also releases the flow of expedited data (if it is being used) if it (also) had been blocked. In particular, the effects of the call are:

– The current process can again receive the events T_DATAIN and T_XDATIN for the specified connection, if they are waiting.

– The sending TS application receives the event T_DATAGO. It may again send data.

Once a connection has been set up, the initiative rests with the TS application (not with CMX). It may send normal data and (if agreed) expedited data, or indicate, with *t_event()*, that it is ready to receive normal data or (if agreed) expedited data.

```
#include <cmx.h>
int   t_datago (const int *tref);
```

-> tref   Pointer to a field with the transport reference of the connection on which the flow of data is to be released.

Return values

T_OK
The call was successful.

T_ERROR
Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_WSEQUENCE
The process is not attached for any TS application, or the process is not in the data phase for the connection specified in *tref*, or the flow of data has not been blocked.

T_CCP_END
    The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datastop(), t_xdatstop(), t_error(), t_event(), t_redin()

8.6.8 t_datain - Receive data (data indication)

t_datain() accepts a T_DATAIN event previously reported via *t_event()*. The *t_datain()* call must be made before the next *t_event()*.

By means of this call the current process receives data of a Transport Interface Data Unit (TIDU) belonging to the current Transport Service Data Unit (TSDU) from the sending TS application on the specified connection.

The maximum length of a TIDU depends on the transport system used. It can be queried for a connection that has already been set up by means of *t_info()*.

A TIDU need not be completely full. The breakdown of a TSDU into TIDUs is purely local and does not indicate anything regarding the breakdown of the TSDU into TIDUs at the sending TS application.

Between two TIDUs of a TSDU any other CMX events can occur for the same or a different connection.

When *t_datain()* is called a contiguous data area *datap* is provided in which CMX enters the data of the TIDU received.

t_datain() indicates:

– (in the *chain* parameter)

  whether a further TIDU belonging to the current TSDU exists (*chain*= T_MORE) or does not exist (*chain* = T_END).

  The individual TIDUs of a TSDU are each indicated via *t_event()* with the event T_DATAIN.

– (with the return value)

  whether the current TIDU has been completely read or not.

  If the value T_OK is returned, the TIDU has fit into the storage area provided. The current process has completely received the current TIDU.

  If a value n > 0 is returned, only a part of the TIDU has been read. n is the number of bytes of the TIDU that have not yet been read (remaining length). In this case *t_datain()* or *t_vdatain()* must be called repeatedly until the entire TIDU has been read. Only then can other CMX calls be issued again, e.g. *t_event()*.

```
#include <cmx.h>
int   t_datain (const int *tref,
                char *datap,
                int *datal,
                int *chain);
```

-> tref

Pointer to a field containing the transport reference of the connection, obtained via *t_event()*.

<- datap

Pointer to a storage area in which CMX enters the data of the TIDU received.

<> datal

Prior to the call, for *datal* a pointer must be specified to a field in which the length of *datap* must be entered (at least 1). Following the call, CMX returns in this field the number of bytes entered in the storage area *datap*. This need not be the maximum length of the TIDU.

<- chain

chain is a pointer to a field in which CMX returns an indicator. This indicator shows whether or not an additional TIDU belonging to the TSDU exists.

Possible values:

T_MORE

Another TIDU belonging to the TSDU follows. It will be indicated with a separate T_DATAIN event.

T_END

The present TIDU is the last of the TSDU.

Return values

T_OK

The call was successful. The TIDU was completely read.

n > 0

n bytes remain from the TIDU.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: The pointer *datap* does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or no T_DATAIN was indicated for the connection specified in *tref*.

T_WPARAMETER
: The length specified in *datal* is invalid.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.

  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_error(), t_event(), t_info(), t_vdatain()

8.6.9 t_datarq - Send data (data request)

t_datarq() sends the next (or only) Transport Interface Data Unit (TIDU) of a Transport Service Data Unit (TSDU) to the receiving TS application on the specified connection.

The TIDU to be sent by *t_datarq()* must be provided by the current process in a contiguous data area.

If the TSDU is longer than one TIDU, it must be transferred using several *t_datarq()* (or *t_vdatarq()*) calls in succession. Therefore in each *t_datarq()* call the sending process must specify in the *chain* parameter whether additional TIDUs belonging to the same TSDU follow.

The maximum length of a TIDU depends on the transport system used. It can be queried for an established connection by means of *t_info()*.

If *t_datarq()* returns the value T_DATASTOP, the TIDU has been accepted by CMX but the flow of TIDUs on this connection has been blocked.

The flow of TIDUs can be blocked by:

– the receiving TS application, which can block the flow of TIDUs by calling *t_datastop()* or *t_xdatstop()*, or

– CMX, if the local buffer is full.

If the flow of TIDUs is blocked, before further TIDUs can be sent you must wait, by means of *t_event()*, for the event T_DATAGO for the connection.

Successful termination of *t_datarq()* (T_OK) does not mean that the receiving TS application has already accepted the data.

Unsuccessful termination of *t_datarq()* (T_ERROR) always means that an error has been detected locally.

```
#include <cmx.h>
int   t_datarq (const int *tref,
                const char *datap,
                const int *datal,
                const int *chain);
```

> tref

Pointer to a field with the transport reference of the connection.

-> datap

Pointer to a data area containing the TIDU to be sent.

-> datal

Pointer to a field containing the number of bytes to be sent from the storage area *datap*. At least 1 and at most the length of a TIDU must be specified.

-> chain

Pointer to an indicator used by the process to indicate whether there is an additional TIDU belonging to the TSDU.

Possible values:

T_MORE

Another TIDU belonging to the TSDU follows.

T_END

The present TIDU is the last of the TSDU.

Return values

T_OK

The call was successful; further TIDUs may be sent immediately.

T_DATASTOP

The call was successful, but further TIDUs may not be sent until the event T_DATAGO has arrived for this connection.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT

The pointer *datap* does not point to the process address space.

T_WSEQUENCE

The process is not attached for any TS application, or the process is not in the data phase for the connection specified in *tref*, or the flow of data is blocked.

T_WPARAMETER

The length specified in *datal* or the value specified in *chain* is invalid.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.

  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datastop(), t_error(), t_event(), t_info(), t_vdatarq(), t_xdatstop()

8.6.10 t_datastop - Stop the flow of data (data stop)

t_datastop() blocks the flow of data on the specified connection.

In particular, the effects of *t_datastop()* are:

– The current process tells CMX that, until further notice, it is not ready to receive data for this connection. However, a T_DATAIN event that has already been indicated must be responded to first.

– The current process no longer receives the event T_DATAIN for the specified connection. However, while the data flow is blocked it may call other CMX functions, e.g. to set up, close down or redirect an additional connection.

– The sending TS application receives the return value T_DATASTOP when it calls *t_datarq()*. It may not send any more data. (See section “Transport system specific features” on page 103.)

The flow of data is released with *t_datago()*.

Expedited data is not affected by *t_datastop()*.

```
#include <cmx.h>
int   t_datastop (const int *tref);
```

-> tref
Pointer to a field with the transport reference of the connection.

Return values

T_OK
The call was successful.

T_ERROR
Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_WSEQUENCE
: The process is not attached for any TS application, or the process is not in the data phase for the connection specified in *tref*, or a TIDU or an ETSDU has not yet been completely read.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datarq(), t_datago(), t_event(), t_xdatstop()

8.6.11 t_detach - Detach a process from a TS application (detach process)

t_detach() detaches the current process for the TS application specified in the parameter *name*. If connections still exist for this process, they are implicitly closed down. Normally though, all connections for this process should be closed down with *t_disrq()* before calling *t_detach()*.

When the last process of a TS application detaches itself, the TS application ceases to exist. Connection requests for that TS application will then no longer be accepted.

```
#include <cmx.h>
int   t_detach (const struct t_myname *name);
```

-> name

Pointer to a structure *t_myname* with the LOCAL NAME of the TS application. The same LOCAL NAME is to be specified as was specified with *t_attach()*.

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT

The pointer *name* does not point to the process address space.

T_WSEQUENCE

The process is not attached for any TS application.

T_WPARAMETER

The LOCAL name passed with *name* has an invalid format or contains illegal values.

T_WAPPLICATION
The process is not attached for the TS application that has the LOCAL NAME passed via *name*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_attach(), t_error()

8.6.12 t_disin - Accept disconnection (disconnection indication)

t_disin() accepts a T_DISIN event previously reported with *t_event()*. T_DISIN indicates that the connection has been closed down.

t_disin() specifies whether the remote TS application or CMX initiated the T_DISIN event.

In addition, *t_disin()* returns:

– the user data sent by the remote TS application, if the T_DISIN event was initiated by the remote TS application and if the transport system used provides this option;

– the reason for closing the transport connection, if the T_DISIN event was initiated by CMX or by the transport system. The readable text form of the code can be obtained with the aid of *t_preason()* or *t_strreason()*.

```
#include <cmx.h>
int   t_disin (const int *tref,
               int *reason,
               t_opt2 *opt);
```

-> tref

Pointer to a field containing the transport reference of the connection.

<- reason

Pointer to a field in which CMX enters the reason for the disconnection.

Possible values:

T_USER

The connection was closed down by the remote TS application.

other

The connection was closed down by CMX or the transport system.

The possible values for this parameter and their meanings can be found in the appendix to this manual. The code returned by CMX for the disconnection can be decoded with the aid of the *cmxdec* command (see the “CMX, Operation and Administration” manual [1]).

<> opt

For *opt*, specify the value NULL or a pointer to a union containing a structure with system options.

This union can be used to check the user data that the remote TS application specified when closing down the connection.

If *opt* = NULL is specified, CMX discards the user data and options.

If the remote TS application specified no user data and no options, CMX returns the default values specified.

The following structure is defined in <*cmx.h*>:

```
   struct t_optc2 {
->    int  t_optnr;    /* Option no. */
<-    char *t_udatap;  /* Data buffer */
<>    int  t_udatal;   /* Length of the data buffer */
   };
```

t_optnr

Option number. Specify T_OPTC2.

t_udatap

Pointer to a data area in which CMX enters the user data received from the remote TS application.

Default value specifying *opt* = NULL: Undefined

t_udatal

Prior to the call 0 or the length of the data area *t_udatap* must appear here.

The area must be large enough that the received data completely fits. The maximum permissible length for the user data depends on the transport system used. T_MSG_SIZE is the maximum size suitable for all transport systems. After the call, CMX returns in this field the number of bytes placed in *t_udatap*.

Default value specifying *opt* = NULL: 0

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: The pointer *opt* (!= NULL) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or no T_DISIN was indicated for the connection specified in *tref*.

T_WPARAMETER
: The options specified in *opt* have an invalid format or contain illegal values, or the buffer for the data to be received is too small.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_detach(), t_disrq(), t_event(), t_preason(), t_strreason()

8.6.13 t_disrq - Close down connection (disconnection request)

t_disrq() closes down the specified connection, or rejects the connection indication of a calling TS application. In both cases the remote TS application receives a disconnect indication with the reason T_USER.

Either partner may close down the connection, regardless of which one actively set it up.

Along with the disconnection the remote TS application may be sent user data, if the transport system provides this option.

The *t_disrq()* call may overtake data that is still in transit. This data is then lost.

```
#include <cmx.h>
int   t_disrq (const int *tref,
               const t_opt2 *opt);
```

-> tref

Pointer to a field containing the transport reference of the connection to be closed down.

-> opt

For *opt*, specify the value NULL or a pointer to a union containing a structure with system options. This union is used to specify the user data that the remote TS application is to receive along with the disconnection indication.

If *opt* = NULL is specified, CMX uses the default values specified.

The following structure is defined in <*cmx.h*>:

```
   struct t_optc2 {
->    int  t_optnr;    /* Option no. */
->    char *t_udatap;  /* Data buffer */
->    int  t_udatal;   /* Length of the data buffer */
   };
```

t_optnr

Option number. Specify T_OPTC2.

t_udatap

Pointer to a storage area containing user data to be received by the remote TS application.

Default value specifying *opt* = NULL: Undefined t_udatal

Length of the user data to be passed in the storage area *t_udatap*. If *t_udatal* = 0 is specified, *t_udatap* is ignored. The maximum value for t_udatal depends on the transport system (see the Release Notice).

Default value specifying *opt* = NULL: 0

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT

At least one of the pointers *opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE

The process is not attached for any TS application, or the connection specified in *tref* is neither set up nor being set up, nor is it being redirected.

T_WPARAMETER

The options specified in *opt* have an invalid format or contain illegal values.

T_CCP_END

The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_detach(), t_disin(), t_event(), t_error()

8.6.14 t_error - Error diagnosis (error)

t_error() returns diagnostic information when another CMX call returns T_ERROR.

The possible error messages for calls to the ICMX(L) program interface are generated either in the CMX library functions in the user process or in the operating system kernel. At this point you should establish if the error messages were generated in the CMX itself or if the messages are the result of operating system calls in CMX. In the first case, *t_error()* outputs an internal CMX error code; in the second case it contains the external variable *errno*.

In both cases the error codes can be converted to readable text form with the aid of the calls *t_strerror()* and *t_perror()*. *t_strerror()* returns a pointer to a static area that contains the readable text form of an error message.

t_perror() writes the readable text form of an error message to *stderr*. The error code can be decoded using the *cmxdec* command (see the “CMX, Operation and Administration” manual [1]). The format of CMX error messages is described in the section “Error handling” on page 40.

```
#include <cmx.h>
int   t_error (void);
```

Return values

The value returned by *t_error()* is the hexadecimal code for the error value generated by CMX. The error values are defined in *<cmx.h>*. A list of all possible error values of error type T_CMXTYPE (0) and error class T_CMXCLASS (0), i.e. all possible return values for *t_error()*, is provided in the appendix.

In the descriptions of the individual ICMX(L) function calls, the error values that *t_error()* returns if a particular function terminates in error are listed under the heading "Errors".

Files

<cmx.h> – Global CMX definition file
<tnsx.h> – TNSX definition file
<errno.h> – Messages for system calls

See also

t_perror(), t_strerror

8.6.15 t_event - Await or query event (event)

t_event() determines whether a CMX event has arrived for the current process.

The parameter *cmode* specifies the processing mode of *t_event()*. *t_event()* can:

– **synchronously** wait for a CMX event for the current process to arrive. While waiting, the process is suspended. Waiting can be interrupted using signals. A time limit for synchronous waiting may be specified in the *opt* options. If no event arrives within this waiting period, waiting is terminated.

– **asynchronously** check whether a CMX event for the current process has arrived. The function always returns immediately to the current process.

Along with the appropriate event, *t_event()* returns:

– the transport reference of the connection involved, to permit the event to be associated with the appropriate connection (*tref* parameter),

– event-specific additional information, if this has been specified in the *opt* options.

In addition, *t_event()* permits CMX to signal the arrival of more data for a connection, if data indications for the connection have not been explicitly blocked via *t_datastop()* or *t_xdatstop()*.

If a T_DATAIN or T_XDATIN event is indicated for a process, the connection involved may not be redirected (see section “States of TS applications and permissible state transitions” on page 96). More importantly, *t_event()* may not be called again until the current process has accepted the indicated data with *t_datain()*, *t_vdatain()* or *t_xdatin()*.

If several events are present for a connection, they are indicated one after another in the order in which they arrived.

Exceptions:

– A T_XDATIN event (expedited data received) may overtake T_DATAIN events (normal data received) without destroying them.

– A T_DISIN event (disconnection indication) may overtake T_DATAIN and T_XDATIN events for the connection involved and thus destroy them.

  The data that T_DATAIN/T_XDATIN was to have indicated is lost.

```
#include <cmx.h>
int   t_event (int *tref,
               int cmode,
               t_opte *opt);
```

<- tref

Pointer to a field in which CMX returns the connection-specific transport reference. The transport reference specifies the connection to which the event belongs. For the events T_NOEVENT and T_ERROR the contents of *tref* are undefined.

-> cmode

cmode is used to specify whether *t_event()* is to synchronously wait for an event or is to asynchronously check whether an event has arrived.

Possible values:

T_WAIT (synchronous processing)

The current process is suspended until a TS event arrives, the specified waiting time elapses (*t_timeout* parameter in *opt*) or a signal occurs (e.g. *alarm(CES)*). In the last two cases the event T_NOEVENT is returned.

T_CHECK (asynchronous processing)

The current process checks whether a TS event is waiting.

If a TS event is waiting for the current process, the event is returned to the process.

If no event is waiting, the event T_NOEVENT is returned to the process.

-> opt

For *opt*, you may specify NULL or a pointer to a union containing structures with system options.

If NULL is specified, CMX uses the defined default values.

The following structure is defined in <*cmx.h*>:

```
   struct t_opte1 {
->    int  t_optnr;     /* Option no. */
<-    int  t_attid;     /* CMX attachment reference */
<-    int  t_uattid;    /* User attachment reference */
<-    int  t_ucepid;    /* User connection reference */
->    int  t_timeout;   /* Time limit for T_WAIT */
<-    int  t_evdat;     /* Event-specific information */
   };
   struct t_opte2 {
->    int  t_optnr;     /* Option no. */
<-    int  t_attid;     /* CMX attachment reference */
<-    int  t_uattid;    /* User attachment reference */
<-    int  t_ucepid;    /* User connection reference */
->    int  t_timeout;   /* Time limit for T_WAIT */
<-    int  t_evdat;     /* Event-specific information */
<-    int t_evinf[10];  /* BS2000 event information */
   };
```

t_optnr

Option number. Specify

T_OPTE1 in *t_opte1*
T_OPTE2 in *t_opte2*

t_attid

In *t_attid t_event()* returns the CMX-internal reference for the attachment involved.

The CMX reference is also returned by CMX as an option in *t_attach()*. It serves only trace and diagnostic purposes and is used exclusively for logging.

t_uattid

In *t_uattid t_event()* returns the user reference for the attachment involved.

The user reference is passed to CMX as an option in *t_attach*. This enables a process that controls multiple TS applications to associate a TS event with the appropriate attachment of a TS application.

t_ucepid

In *t_ucepid t_event()* returns the user reference for the connection involved for the TS events T_CONCF, T_DATAIN, T_XDATIN, T_DATAGO, T_XDATGO and T_DISIN.

The user reference is passed to CMX in *t_conrq()*, *t_conrs()* or *t_redin()*. This enables a process that maintains multiple connections to associate a TS event with the appropriate connection. This feature, selected by the user, constitutes an alternative to the transport reference *tref*, defined by CMX.

t_timeout

With *cmode* = T_WAIT:

For *t_timeout* a waiting period may be specified during which *t_event()* is to synchronously wait for an event.

With *cmode* = T_CHECK:

Any value specified for *t_timeout* is ignored.

Possible specifications for *t_timeout*:

T_NOLIMIT

No waiting period is defined. The process waits (without time limit) until an event arrives or *t_event()* is terminated by a signal.

T_NO

The process does not wait. It resumes immediately with any TS event present or with T_NOEVENT (corresponds to *cmode* = T_CHECK).

n > 0

The process waits n seconds for the arrival of a TS event. If no TS event for the waiting process arrives within this time period, the process resumes with the event T_NOEVENT. Waiting may be terminated by means of signals.

Default value specifying *opt* = NULL: T_NOLIMIT

t_evdat
: Here, CMX returns event-specific additional information.

  Possible information:

  With the events T_DATAIN and T_XDATIN the length of the indicated data is specified here.

  With the other TS events, including T_NOEVENT, the additional information is undefined.

t_evinf[10]
: This field is used by BS2000 applications and is not supported by CMX in Solaris.

Return values

T_CONIN
: This event indicates that a calling TS application wishes to set up a connection to the current process. This connection indication must first be fetched with *t_conin()*, then confirmed with *t_conrs()* or rejected with *t_disrq()*.

T_CONCF
: This event indicates that the called TS application has responded positively to a connection request of the current process.

  This connection setup confirmation must be fetched with *t_concf()*.

T_DATAIN
: This event indicates that data has been received via the connection specified in *tref*. The data must be fetched with *t_datain()* or *t_vdatain()*. CMX does not indicate this event for a connection so long as data flow on it is blocked, i.e. when the receiving process has issued *t_datastop()* for it.

T_DATAGO
: The local TS application may resume sending data on the connection specified in *tref*.

  Possible reaction: *t_datarq()* or *t_vdatarq()*.

  The event T_DATAGO also permits the local TS application to resume sending expedited data on this connection, assuming the sending and receiving of expedited data was agreed at connection setup.

T_DISIN

This event indicates disconnection of the connection specified in *tref*.

This disconnect indication must be fetched with *t_disin()*.

T_ERROR

Error. Query error code using *t_error()*.

T_NOEVENT

This event means:

If cmode = T_CHECK

No event waiting.

If cmode = T_WAIT

Wait status of the process terminated, either by signal or because the specified waiting period elapsed. No TS event arrived.

The contents of *tref* are undefined.

T_REDIN

This event indicates that another process of the same TS application has redirected a connection to the current process.

The connection redirection must be fetched with *t_redin()*.

T_XDATIN

This event indicates that expedited data has been received on the connection specified in *tref*. The data must be fetched with *t_xdatin()*.

This event is indicated only:

– if the exchange of expedited data was agreed at connection setup, and

– while the flow of expedited data on the connection is not blocked. The flow of expedited data is blocked when the receiving process has issued *t_xdatstop()* for the connection.

T_XDATGO

With this event CMX indicates that the process may resume sending expedited data on the connection specified in *tref*.

Possible reaction: *t_xdatrq()*.

CMX indicates this event only if the exchange of expedited data was agreed at connection setup.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: The pointer *opt* (!= NULL) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or a TIDU or ETSDU has not yet been completely read.

T_WPARAMETER
: The value specified in *cmode* is invalid, or the options specified in *opt* have an invalid format or contain illegal values.

T_CBRECURSIVE
: Recursive *t_event* call in a callback routine is not permitted.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_attach(), t_callback(), t_concf(), t_conin(), t_datain(), t_datago(), t_datastop(), t_disin(), t_error(), t_redin(), t_vdatain(), t_xdatin(), t_xdatgo(), t_xdatstop()

8.6.16 t_getaddr - Query TRANSPORT ADDRESS for the GLOBAL NAME (get address)

t_getaddr returns either the TRANSPORT ADDRESS of the object *globname* from TS directory 1 (*globname* != NULL) or a template (*globname* = NULL) whose address format(s) depend(s) on the value being conveyed via *opt*. Parameter *globname* is the GLOBAL NAME of the TS application. *t_getaddr()* returns a pointer to a static area with the TRANSPORT ADDRESS of this TS application.

```
const struct t_partaddr *t_getaddr (const char *glob,
                                    const t_optg *opt);
```

globname

is the GLOBAL NAME of the TS application.

t_getaddr()

returns a pointer to a static area with the TRANSPORT ADDRESS of this TS application.

"->globname != NULL"

specifies the GLOBAL NAME of the object in the TS directory and is expected in the form NP5.NP4.NP3.NP2.NP1 as a string (NULL-terminated). The items NPi represent the name parts of the GLOBAL NAME in ascending hierarchical order from left to right. If an NPi has no value, the separator ’.’ must be specified if this empty name part is followed by at least one more name part that is higher in the hierarchy (a series of ’.’ at the end may be omitted). At least one of the name parts NPi must have a value. ’.’ must be specified as ’\.’ in the NPi.

„->globname == NULL"

makes *t_getaddr* return a pointer to a template. Its structure depends on the value conveyed via the parameter opt.

„->opt, if globname != NULL"

Pointer to a union with system options or NULL. If *globname* != NULL CMX in UNIX ignores this parameter.

The following structure is specified in *<cmx.h>* only for compatibility with CMX in BS2000/OSD:

```
    struct t_optg1 {
->      int t_optnr;      /* Option no.                     */
->      int t_evref;      /* System event reference point */
->      char *t_buf[200]; /* Work area                      */
     }
```

t_optnr
: Option number. Specify: T_OPTG1 in *t_optg1*

t_evref
: This field is used by BS2000/OSD and is not supported by CMX in UNIX.

t_buf[200]
: This field is used by BS2000/OSD and is not supported by CMX in UNIX.

"->opt if globname == NULL"
: points to a union contained in the following *<cmx.h>* defined structure:

```
       struct t_optg7 {
->          int t_optnr;       /* Option no. = T_OPTG7          */
->          int t_addrtype;    /* Selected address format       */
       }
```

In this case *t_getaddr* returns a pointer to an address template. The structure of the address template depends on the value of *t_addrtype* in *t_optg7*:

Address format CX_RFC1006 selected via addrtype = T_INETA

```
pa_header | rest ...........
02000027     0040010 001d300d 03490000 00070001 f0f0f0f0 fe810200  668008
             |  |     |                            |               |port|
      cx_type   ka_type/ka_size                     IPv4-Addr.     nr.|
             61 61616161 616161
             | - tselector -  |
```

Address format CX_LANINET selected via addrtype = T_INETA | T_LANINET

```
pa_header | rest ...........
0200001f     01000010 0016300d 03490000 00070001 f0f0f0f0 fe800431 313131
             |    |    |                            |               | tsel
      cx_type   ka_type/ka_size                     IPv4-Address
```

Adress format CX_RFC1006, selected via addrtype = T_INETA6

```
pa_header | rest ...........
02000033     00040010 00293019 03490000 00130001 fe800000 00000000
             |    |    |                            |
      cx_type   ka_type/ka_size                     IPv6-Address
028017ff     fe287b08 fe810200 66800861 61616161 616161
                             |         |
                      Portnumber     T-Selector
```

Adress format CX_LANINET, selected via addrtype = T_INETA6 / T_LANINET

```
pa_header | rest ...........
0200002b    01000010 00213019 03490000 00130001 fe800000 00000000
            |   |    |                          |
      cx_type   ka_type/ka_size                 IPv6-Address
028017ff    fe287b08 fe800431 313131
                           |
                           T-Selector
```

Address format CX_OSITYPE selected via addrtype = T_OSI

```
pa_header | rest ...........
02000020    20000010 0016400a 49006cf0 f0f0f0f0 f0008008 61616161 61616161
            |   |    |             |-MAC address|       | - tselector - |
      cx_type   ka_type/ka_size

          /* AU90 + AU91 (Routing/CC info) not manageable at present */
```

Address format CX_WANNEA selected via addrtype = T_NEA

```
pa_header | rest ...........
02000018    00010010 000e4102 ffaa8008 61616161 61616161
            |   |    |        |reg-    | - tselector - |
      cx_type   ka_type/ka_size pro-nr

          /* AU90 + AU91 (Routing/CC info) not manageable at present */
```

Address format CX_WANSBKA with ISDN no. selected via addrtype = T_E164

```
pa_header | rest ...........
0200001e    00100010 00147008 03063131 31313131 80086161 61616161 6161
            |   |    |            |ISDN number|     |  - tselector - |
      cx_type   ka_type/ka_size

          /* AU90 + AU91 (Routing/CC info) + AUD0 + AUD1 (transport protocol
               identifier/ protocol class defaults) not manageable for now */
```

Address format CX_WANSBKA with tel. no. selected via addrtype = T_E163

```
pa_header | rest ...........
0200001e    00100010 00147008 08063131 31313131 80086161 61616161 6161
            |   |    |            |tel.-number|     |  - tselector - |
      cx_type   ka_type/ka_size
      /* AU90 + AU91 (Routing/CC info) + AUD0 + AUD1 (transport protocol
        identifier/protocol class defaults) not manageable at present */
```

Address format CX_WANSBKA with DTE addr. selected via addrtype = T_X121

```
pa_header | rest ...........
0200001c    00100010 00107008 01041111 111f8008 61616161 61616161
            |   |    |            |DTE-adr|     |  - tselector - |
      cx_type   ka_type/ka_size

      /* AU90 + AU91 (Routing/CC info) + AUD0 + AUD1 (transport protocol
        identifier/protocol class defaults) not manageable at present */
```

Address format CX_WAN3SBKA with DTE addr. sel. via addrtype=T_X121|T_NULLTP

```
pa_header | rest ...........
0200001c    00080010 00123007 06370611 11118008 61616161 61616161
            |   |    |            |DTE-adr|     |  - tselector - |
      cx_type   ka_type/ka_size

          /* AU90 + AU91 (Routing/CC info) not manageable at present */
```

Address format CX_WANSBKA with X.21 dial no. sel. via addrtype = T_X21

```
pa_header | rest ...........
0200001e    00100010 00147008 08063131 31313131 80086161 61616161 6161
            |   |    |            |tel.number|     |  - tselector - |
      cx_type   ka_type/ka_size

     /* AU90 + AU91 (Routing/CC info) + AUD0 + AUD1 (transport protocol
       identifier/protocol class defaults) not manageable at present */
```

Address format CX_WANSBKA with PVC nr. selected via addrtype = T_PVC

```
pa_header | rest ...........
0200001a    00100010 00107004 050200ff 80086161 61616161 6161
            |   |    |            |PVC     |  - tselector - |
      cx_type   ka_type/ka_size    nr.

     /* AU90 + AU91 (Routing/ CC Info) + AUD0 + AUD1 (transport protocol
        identifier/protocol class defaults) not manageable at present */
```

To alter the template’s values use *t_setaddrpart*.

Return value

t_getaddr()
: returns a pointer to a static area containing the TRANSPORT ADDRESS or the NULL pointer in the case of an error.

Errors

If an error occurs, the error code can be ascertained by means of t_error().

The following error codes of error types from the set T_DSTEMP_ERR, T_DSCALL_ERR, T_DSPERM_ERR, T_DSWARNING may occur.

For error class T_DSPAR_ERR, the following error values may occur:

T_DIRERR
: TS directory DIR1 is not found.

T_NAMERR
: The GLOBAL NAME specified in *globname* does not exist.

T_ILLNAM
: The GLOBAL NAME specified in *globname* is syntactically invalid (too many name parts, invalid lengths of name parts, invalid characters within the name).

T_PROPER
: There is no TRANSPORT ADDRESS assigned to the GLOBAL NAME specified in *globname*.

Possible error values with error class T_DSSYS_ERR are the system error messages defined in *<errno.h>*.

For error class T_DSILL_VERS, the following error values may occur:

T_NOTSPEC
: The CMX library version linked into the process is incompatible with the CMX runtime environment.

For error class T_DSINT_ERR, the following error values are possible:

T_TIMOUT
: tnsxd(CMX_1) did not respond within the time limit (20 sec).

T_PROT
: Errors occurred in the protocol with tnsxd(CMX_1).

T_LFILE
: TS directory DIR1 has an incorrect format.

Application usage

The static area mentioned above is overwritten with every call. The caller must copy the area if it is to be saved. The amount of data to be copied can be determined from the size field *t_palng* defined in struct *t_partaddr*.

8.6.17 t_getaddrpart, t_setaddrpart - Read or change address information in TRANSPORT ADDRESS

t_getaddrpart - Read address information from TRANSPORT ADDRESS
t_setaddrpart - Change address information in TRANSPORT ADDRESS

t_getaddrpart() returns the individual items of service-specific address information about the TRANSPORT ADDRESS in *addr*.

t_setaddrpart() returns a changed TRANSPORT ADDRESS in *newaddr* where the calling program gives the original TRANSPORT ADDRESS in *addr* and the modification to the individual item of service-specific address information in *opt*.

```
#include <cmx.h>
int   t_getaddrpart (const union t_address *addr,
                     t_optg *opt);
int   t_setaddrpart (const union t_address *addr,
                     union t_address *newaddr,
                     const t_optg *opt);
```

-> addr

Pointer to a union *t_address* which contains a TRANSPORT ADDRESS. The application program can obtain this TRANSPORT ADDRESS as a return value from *t_getaddr()* or as a *fromaddr* parameter from *t_conin()*.

<- newaddr

Pointer to a union *t_address*. In this union, *t_setaddrpart()* returns the new TRANSPORT ADDRESS with the value given in *opt*. This TRANSPORT ADDRESS can be transfered as a *toaddr* parameter to *t_conrq()* or as a *addr* parameter to *t_getname()*.

-> opt

Pointer to a union that contains the following <cmx.h> structure:

```
struct t_optg5 {
    int t_optnr;                /* Option no.                */
#define T_OPTG5 5
    struct t_addrpart  t_nsap;   /* NSAP address*/
    struct t_addrpart  t_tsel;   /* T selector*/
    struct t_addrpart  t_ssel;   /* S selector*/
    struct t_addrpart  t_psel;   /* P selector*/
    /* Parameters for TCP TRANSPORT ADDRESS */
    int portnumber;             /* TCP port number           */
    /* Parameters for BAM/HDLC TRANSPORT ADDRESS */
    unsigned char escaddr;      /* BAM/HDLC escape address    */
    unsigned char devtype;      /* Device type               */
    unsigned char proname[9];   /* BAM/HDLC processor name    */
}
```

-> t_optnr

Option number. Specify: T_OPTG5 in *t_optg5*.
The following data structure is defined in *<cmx.h>*. It describes the individual components of a TRANSPORT ADDRESS in the coding given in the corresponding protocols and standards. In other words it describes these components outside CMX and therefore without the CMX internal packing.

```
struct t_addrpart {
     int t_maxlen;                /* Maximum length of t_bufp    */
     int t_type;                  /* Type of NSAP/TSEL/SSEL/PSEL */
     int t_len;                   /* Returned length of
                                     Information in t_bufp       */
     char *t_bufp;                /* NSAP/TSAP/SSAP/PSAP         */
}
```

-> t_maxlen

Length of the buffer *t_bufp* provided by the user program. If the information to be stored in *t_bufp* is longer than *t_maxlen*, an error will occur and no information will be written to *t_bufp*.

<- t_type (*t_getaddrpart()*)
-> t_type (*t_setaddrpart()*)

This parameter is described in the table below. The structures *t_tsel*, *t_ssel* and *t_psel* are valid for both the value T_VOID (does not exist) and the value T_EXIST (exists). The combination
t_type = T_EXIST and *t_len* = 0 is therefore possible. For the remaining values the table gives the the coding of the network service specific address information and the corresponding CMX address format. For *t_setaddrpart()*, the value *t_type* = T_VOID means that the corresponding component of the TRANSPORT ADDRESS remains unchanged. For *t_getaddrpart()*, the value *t_type* = T_VOID means that the corresponding component of the TRANSPORT ADDRESS is missing.

t_type	**Currently used in address format**	**Format**	**Meaning**
T_INETA	LANINET RFC1006	Binary coded (MSB) 4 or 16 octets	Pv4 or IPv6 address
T_OSI	OSITYPE	OSI-Address as specified in ISO 8348/Add.2	OSI address format
T_NEA	STANEA	Binary coded Byte[0] Processor# Byte[1] Region#	NEA address format
T_E164	WANSBKA	Max. 15 Bytes ASCII coded	ISDN number
T_E163	WANSBKA WAN3SBKA	Max. 14 Bytes ASCII coded	Tel. number
T_X121	WANSBKA WAN3SBKA	Max. 15 digits padded with 0xf	X.25 address
T_X21	WANSBKA	Max. 20 Bytes ASCII coded	X.21 address
T_PVC	WANSBKA	Binary coded (MSB) 2 octets	
T_VOID			Address part does not exist or remains unchanged
T_EXIST			Address part exists (*t_len* = 0 is possible)

Table 7: t_type

<- t_len (*t_getaddrpart()*)
-> t_len (*t_setaddrpart()*)

Length of the service-specific component of the TRANSPORT ADDRESS stored in *t_bufp*. This is returned by *t_getaddrpart()* and transferred to *t_setaddrpart()*.

<- t_bufp (*t_getaddrpart()*)
-> t_bufp (*t_setaddrpart()*)

Pointer to a storage area provided by the user program. *t_getaddrpart()* places the corresponding service-specific component of the TRANSPORT ADDRESS here. The new value of the for the corresponding service-specific component of the TRANSPORT ADDRESS is transferred from here to *t_setaddrpart()*. For *t_nsap*, this is the network service; for *t_tsel*, this is the transport service, for *t_ssel*, this is the session service and for *t_psel*, this is the presentation service.

t_nsap

This structure describes the network service specific component of the TRANSPORT ADDRESS.

t_tsel

This structure describes the transport service specific component of the TRANSPORT ADDRESS. If the CMX address format LANINET is used, this structure will be ignored (*t_type* = T_VOID).

t_ssel

This structure describes the session specific component of the TRANSPORT ADDRESS.

t_psel

This structure describes the presentation specific component of the TRANSPORT ADDRESS.

<- portnumber (*t_getaddrpart()*)
-> portnumber (*t_setaddrpart()*)

TCP port number. This is only used if the structure element *t_type* in the structure *t_nsap* is set as *t_type*= T_INET.

escaddr

BAM escape address. Not used.

devtype

BAM device type. Not used.

proname

BAM processor name. Not used.

i The value *t_type* = T_INETA in *t_nsap* is valid for both IPv4 and IPv6 addresses. However, for *t_getaddr()* where *t_addrtype* = T_INETA in the structure *t_optg7*, a TRANSPORT ADDRESS template with an IPv4 address is required. Where *t_addrtype* = T_INETA6, a template with an IPv6 address is required.

Return values

T_OK
: Successful completion. Information about the TRANSPORT ADDRESS has been written into the structure.

T_ERROR
: Error. Query error code using t_error().

ERRORS

If an error occurs, the error code can be ascertained by means of t_error().

For error type T_CMXTYPE and error class T_CMXCLASS, the following error values may occur:

T_WPARAMETER
: The value specified in *addr* or *opt* is a NULL pointer or *addr* does not point to a partner address.

Application usage

Applications can modify the result of *t_getaddr()* and *t_conin()* for their own usage.

8.6.18 t_getloc - Query LOCAL NAME

t_getloc returns either the LOCAL NAME of the object *globname* from TS directory 1 (*globname* != NULL) or a template (*globname* = NULL).

```
#include <cmx.h>
const struct t_myname *t_getloc (const char *glob,
                                 const t_optg *opt);
```

-> globname!=NULL

specifies the GLOBAL NAME of the object in the TS directory and is expected in the form NP5.NP4.NP3.NP2.NP1 as a string (NULL-terminated). The items NPi represent the name parts of the GLOBAL NAME in ascending hierarchical order from left to right. If an NPi has no value, the separator ’.’ must be specified if this name part is followed by at least one more name part that is higher in the hierarchy (a series of ’.’ at the end may be omitted). At least one of the name parts NPi must have a value. ’.’ must be represented as ’\.’ in the NPi. ’.’ as a component of a name part must be devaluated with ’\.’.

> globname=NULL

makes *t_getloc* return a pointer to a template with the following structure:

```
mn_header | rest ...........
01000032    000e0000 00000100 00043131 31310000
00000000 /*LANINET Tsel*/
                        0040 000a6161 61616161 61613300 /*EMSNA
Tsel  */
                        241f 00086161 61616161 61610000
                        |
                        CX_WANNEA + CX_LANSBKA + CX_RFC1006
+ OSITYPE Tsel
```

To alter the template’s values use *t_setlocpart*.

-> opt

Pointer to a union with system options or NULL.

```
   struct t_optg1 {
-> int t_optnr;      /* Option no.                   */
-> int t_evref;      /* System event reference point */
-> char *t_buf[200]; /* Work area                    */
   }
```

t_optnr

Option number. Specify: T_OPTG1 in *t_optg1*

t_evref

This field is used by BS2000/OSD and is not supported by CMX in UNIX.

t_buf[200]
    This field is used by BS2000/OSD and is not supported by CMX in UNIX.

Return Values

t_getloc() returns a pointer to a static area containing the LOCAL NAME or the NULL pointer in the case of an error.

Errors

If an error occurs, the error code can be ascertained with t_error().

For error types T_DSTEMP_ERR, T_DSCALL_ERR, T_DSPERM_ERR, and T_DSWARNING, the following error codes may occur.

The following error values may occur with error class T_DSPAR_ERR:

T_DIRERR
    TS directory DIR1 not found.

T_NAMERR
    The GLOBAL NAME specified in *globname* does not exist.

T_ILLNAM
    The GLOBAL NAME specified in *globname* is syntactically invalid (too many name parts, invalid lengths of name parts, invalid characters within the name).

T_PROPER
    There is no LOCAL NAME assigned to the GLOBAL NAME specified in *globname*.

Possible error values with error class T_DSSYS_ERR are the system error messages defined in <errno.h>.

For error class T_DSILL_VERS, the following error value may occur:

T_NOTSPEC
    The CMX library version linked into the process is incompatible with the CMX runtime environment.

For error class T_DSINT_ERR, the following error values are possible:

T_TIMOUT
    tnsxd(CMX_1)did not respond within the time limit (20 sec).

T_PROT
    Errors occurred in the tnsxd(CMX_1) protocol.

T_LFILE
TS directory DIR1 has an incorrect format.

Application usage

The static area mentioned above is overwritten in every call. The caller must copy the area if it is to be saved. The amount of data to be copied can be determined from the size of the field *t_mnlng* defined in struct *t_myname*.

8.6.19 t_getlocpart, t_setlocpart - Read or change address information in LOCAL NAME

t_getlocpart - Read address information from LOCAL NAME
t_setlocpart - Change address information in LOCAL NAME

t_getlocpart() returns the individual items of service specific address information in *opt* for the LOCAL NAME given in *addr*.

t_setlocpart() returns a changed LOKAL NAME in *newaddr*, where the calling program gives the original LOCAL NAME in *addr* and the modification to the individual item of service specific address information in *opt*.
For transport service specific address information you should note the following special features:
The CMX address format LANINET uses *portnumber* and the CMX address format EMSNA uses *lu_size*, *luname* and *lunumber* instead of *t_tsel*. In cases where a LOCAL NAME in any of the remaining CMX address formats has several concurrently valid names, then these must have the same value for the T selectors and must have the same coding.

```
#include <cmx.h>
int   t_getlocpart (const union t_address *addr,
                    t_optg *opt);
int   t_setlocpart (const union t_address *addr,
                    union t_address *newaddr,
                    const t_optg *opt);
```

-> addr

Pointer to a union *t_address* that contains the LOCAL NAME of a TS application. The CMX program can, for example, obtain the LOCAL NAME by using the *t_getloc()* call.

<- newaddr

Pointer to a union *t_address*. In this case, *t_setlocpart()* returns the modified LOCAL NAME. This can be used as the parameter *t_myname* with *t_attach()* or as the parameter *fromaddr* with *t_conrq()*.

-> opt

Pointer to a union that contains one of the following <cmx.h> structures:

```
    struct t_optg5 {
    int t_optnr;                  /* Option no.               */
#define T_OPTG5 5
    struct t_addrpart  t_nsap;    /* Information about the NSAP */
    struct t_addrpart  t_tsel;    /* Information about the TSEL */
    struct t_addrpart  t_ssel;    /* Information about the SSEL */
    struct t_addrpart  t_psel;   /* Information about the PSEL */

    /* Parameters for TCP TRANSPORT ADDRESS */
int portnumber;             /* TCP port number               */

    /* Parameters for BAM/HDLC TRANSPORT ADDRESS */
    unsigned char escaddr;       /* BAM/HDLC escape address    */
    unsigned char devtype;       /* Device type                */
    unsigned char proname[9];    /* BAM/HDLC processor name    */
}


struct t_optg6 {
    int t_optnr;                 /* Option no.                 */
#define T_OPTG6 6
    struct t_addrpart  t_nsap;   /* Information about the NSAP */
    struct t_addrpart  t_tsel;   /* Information about the TSEL */
    struct t_addrpart  t_ssel;   /* Information about the SSEL */
    struct t_addrpart  t_psel;   /* Information about the PSEL */

    /* Parameters for TCP TRANSPORT ADDRESS */
    int portnumber;              /* TCP port number            */

    /* Parameter for SNA LUNAME */
    int lu_size;                 /* Length of LUNAME           */

    /* Parameters for BAM/HDLC TRANSPORT ADDRESS */
    unsigned char escaddr;       /* BAM/HDLC escape address    */
    unsigned char devtype;       /* Device type                */
    unsigned char proname[9];    /* BAM/HDLC processor name    */

    /* Parameter for SNA LUNAME */
    unsigned char luname[8];     /* LU name (max. 8) +         */
    unsigned char lunumber;      /* LU number                  */
}
```

t_optnr

Option number. Specify: T_OPTG5 when using *t_optg5*, or T_OPTG6 with *t_optg6*. *t_optg6*; these should only be used with the address format CX_EMSNA where the LUNAME is part of the LOCAL NAME.

The following data structure is defined in <cmx.h>. It describes the individual components of a LOCAL NAME using the coding given in the corresponding protocols and standards. In other words, it describes these components outside CMX and therefore without the CMX internal packing.

```
struct t_addrpart {
     int t_maxlen;              /* Buffer length          */
     int t_type;                /* T_EXIST or T_VOID      */
     int t_len;                 /* Information length     */
     char *t_bufp;              /* Buffer                 */
  }
```

-> t_maxlen

Length of the buffer *t_bufp* provided by the user program. With *t_getlocpart*, if the information to be stored in *t_bufp* is longer than *t_maxlen,* an error will occur and no information will be written to *t_bufp*.

<- t_type (*t_getlocpart()*)
-> t_type (*t_setlocpart()*)

If *t_getlocpart()* returns the value T_VOID, this means that the corresponding address information is not present. If *t_setlocpart()* returns the value T_VOID, this means that the corresponding address information can remain unchanged.
If *t_getlocpart()* returns the value T_EXIST, this means that the corresponding address information is present. If *t_setlocpart()* returns the value T_EXIST, this means that the corresponding address information will be replaced by the new value entered.

<- t_len (*t_getlocpart()*)
-> t_len (*t_setlocpart()*)

Length of the service specific address information stored in the buffer *t_bufp*.

<- t_bufp (*t_getlocpart()*)
-> t_bufp (*t_setlocpart()*)

Pointer to a storage area provided by the user program.
t_getlocpart() places the corresponding service specific address information of the LOCAL NAME here.
The new value for the corresponding service specific address information of the LOCAL NAME is transferred from here to *t_setlocpart()*.

t_tsel

This structure describes the transport service specific address information of the LOCAL NAME (T selector).

t_ssel

This structure describes the session service specific address information of the LOCAL NAME (S selector).

t_psel

This structure describes the presentation service specific address information of the LOCAL NAME (P selector).

<- portnumber (*t_getlocpart()*)
-> portnumber (*t_setlocpart()*)

TCP portnumber (transport service specific address information for the CMX address format LANINET).

<- lu_size (*t_getlocpart()*)
-> lu_size (*t_setlocpart()*)
    Length of the LU name given in *luname* (transport service specific address information for the CMX address format EMSNA).

escaddr
    BAM escape address. This parameter is not used.

devtype
    BAM device type. This parameter is not used.

proname
    BAM processor name. This parameter is not used.

<- luname (*t_getlocpart()*)
-> luname (*t_setlocpart()*)
    SNA LU name (transport service specific address information for the CMX address format EMSNA).

<- lunumber (*t_getlocpart()*)
-> lunumber (*t_setlocpart()*)
    SNA LU number (transport service specific address information for the CMX address format EMSNA).

Return values

T_OK
    Successful completion.

T_ERROR
    Error. Query error code using t_error()

Errors

If an error occurs, the error code can be ascertained by means of t_error ().

For error type T_CMXTYPE and error class T_CMXCLASS, the following error value may occur:

T_WPARAMETER
    The value specified in *name* or *opt* is a NULL pointer or *name* does not point to a union *t_address*.

8.6.20 t_getname - Query GLOBAL NAME (get name)

Given the TRANSPORT ADDRESS of a remote TS application, *t_getname()* ascertains its GLOBAL NAME from TS directory 1.

The TRANSPORT ADDRESS of the TS application must be specified by the caller in the parameter *addr*.

t_getname() returns a pointer to a static area containing the GLOBAL NAME of the TS application.

This static area is overwritten at each call. If the contents of the area must be saved, the caller must copy the area.

The GLOBAL NAME is returned by CMX as a NULL-terminated string in the form NP5.NP4.NP3.NP2.NP1

The items NPi (i=1,2,3,4,5) represent the name parts of the GLOBAL NAME. NP5 is name part[5], i.e. the name part at the lowest hierarchical level. NP1 is name part[1], i.e. the highest name part in the hierarchy. The remaining name parts are specified in increasing hierarchical order from left to right.

If one of the name parts for a particular GLOBAL NAME has no value (e.g. NP4), and this name part is followed by another name part that is higher in the hierarchy (e.g. NP3), the separator (.) from the name part with no value is nevertheless returned.

A series of separators appearing at the end of the value of *globname* is omitted.

The GLOBAL NAME is then specified by CMX as follows: "NP5..NP3"

If the separator character (.) is a component of a name part, it is represented as \. (backslash period).

```
#include <cmx.h>
const char   *t_getname (const struct t_partaddr *addr,
                         const t_optg *opt);
```

-> addr

Pointer to a storage area with the TRANSPORT ADDRESS

-> opt

Pointer to a unit with system options or NULL.

CMX in Solaris ignores this parameter. The following structure is only defined in <*cmx.h*> to maintain compatibility with CMX in BS2000/OSD.

```
struct t_optg1 {
->    int  t_optnr;   /* Option no. */
->    int  t_evref;   /* System event reference point */
->    char *t_buf[200]; /* Work area */
   };
```

t_optnr

Option number. Specify:

T_OPTG1 in *t_optg1*

t_evref

This field is used by BS2000/OSD and is not supported by CMX in Solaris.

t_buf[200]

This field is used by BS2000/OSD and is not supported by CMX in Solaris.

Return values

If the call was successful, *t_getname()* returns a pointer to a storage area containing the GLOBAL NAME.

In case of error, *t_getname()* returns a NULL pointer. The error code can be queried using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

Error values of error types T_DSTEMP_ERR, T_DSCALL_ERR, T_DSPERM_ERR and T_DSWARNING may occur.

For error class T_DSPAR_ERR, the following error values may occur:

T_DIRERR

TS directory DIR1 is not present.

T_LENERR

The length field *t_palng*, contained in the TRANSPORT ADDRESS specified in *addr*, has an invalid value.

Possible error values with error class T_DSSYS_ERR are the system error messages defined in <*errno.h*>.

For error class T_DSILL_VERS, the following error value may occur:

T_NOTSPEC
: The CMX version linked into the process and the CMX runtime environment are incompatible.

For error class T_DSINT_ERR, the following error values are possible:

T_TIMOUT
: The TNSX daemon *tnsxd* does not respond within 20 seconds.

T_PROT
: Errors occurred in the protocol with *tnsxd*.

T_LFILE
: TS directory 1 (DIR1) has an incorrect format.

For error class T_DSMESSAGE, the following error value is possible:

T_LEAFNO
: In TS directory 1 there are either none or more than one GLOBAL NAME to which the TRANSPORT ADDRESS specified in *addr* is assigned.

See also

t_error(), TNSX in the manual „CMX, Operation and Administration“ [1] or [2]

8.6.21 t_info - Query information on CMX (information)

t_info() provides information about the maximum TIDU length. Generally, this information only becomes available after the transport connection has been set up completely.

```
#include <cmx.h>
int   t_info (const int *tref,
              t_opti *opt);
```

-> tref

Pointer to a field with the transport reference of the connection.

<> opt

For *opt*, you may specify NULL or a pointer to a union containing structures with system options.

The following structure is defined in <*cmx.h*>:

```
struct t_opti1 {
->    int t_optnr;          /* Option no. */
<-    int t_maxl;           /* TIDU length */
   };
   struct t_opti2 {
->    int t_optnr;          /* Option no. */
<-    int t_evref;          /* System reference point */
<-    int t_buffer[180];    /* Buffer for Name Service
                               output /*
   };
```

t_optnr

Option number. Specify:
T_OPTI1 in *t_opti1*
T_OPTI2 in *t_opti2*

t_maxl

CMX enters the maximum length of the TIDU in this field.

This value specifies the maximum number of bytes that can be sent to CMX or received from CMX per call when transferring data over this connection.

t_evref

This field is used by BS2000 applications and is not supported by CMX in Solaris.

t_buffer[180]
: This field is used by BS2000 applications and is not supported by CMX in Solaris.

Return values

T_OK
: The call was successful.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: The pointer *opt* (!= NULL) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or

  the desired information is not available. *t_info()* can only be executed when the connection has been set up, because only then can information be output about the connection.

T_WPARAMETER
: The options passed via *opt* have an invalid format or contain illegal values.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

8.6.22 t_perror - Output CMX error message in decoded form

t_perror() decodes CMX error messages passed to the process in hexadecimal form by CMX when *t_error()* is called.

t_perror() writes the plain English form of the CMX error message specified in *code* to the standard error output *stderr*.

In the *s* parameter an additional explanatory text may be specified, e.g an indication of the CMX call and TS application to which the error refers.

Format of output from *t_perror()*:

t_perror() first writes the text specified with *s* (if *s* != NULL), then : (colon) and \n (newline). This is followed by the plain English form of the CMX error message passed. This text consists of the error symbols, as defined in <*cmx.h*>, and an accompanying text. Each error symbol is preceded by \t. Each accompanying text ends with \n.

The accompanying text is taken from the message file *cmxlib.cat*. It will not be output if *cmxlib.cat* is not available on your system. The format of *cmxlib.cat* is dependent on the operating system and the set language variable. See the appropriate system manual for more details.

```
#include <cmx.h>
void   t_perror (const char *s,
                 int code);
```

-> s

Pointer to a storage area containing text that is to precede the readable text form of the error message, or the value NULL.

-> code

For *code*, specify the representation of the error message that was passed to the process by CMX when *t_error()* was called.

Files

cmxlib.cat - Message file

See also

t_error(), t_strerror()

8.6.23 t_preason - Decode and output reasons for disconnection

t_preason() decodes reasons for disconnection passed to the process in hexadecimal form when *t_disin()* is called.

t_preason() writes the plain English form of the reason for disconnection specified in *reason* to the standard error output *stderr*.

In the *s* parameter an additional explanatory text may be specified, e.g an indication of the connection or TS application to which the output refers.

Format of output from *t_preason()*:

t_preason() first writes the text specified with *s* (if *s* != NULL), then : (colon) and \n (newline). This is followed by the plain English form of the disconnection reason passed. This text consists of the symbol for the disconnection reason, as defined in <*cmx.h*>, and an accompanying text. The symbol for the disconnection reason is preceded by \t. The accompanying text ends with \n.

The accompanying text is taken from the message file *cmxlib.cat*. It will not be output if *cmxlib.cat* is not available on your system. The format of *cmxlib.cat* is dependent on the operating system and the set language variable. See the appropriate system manual for more details.

```
#include <cmx.h>
void   t_preason (const char *s,
                  int reason);
```

-> s

Pointer to a storage area containing text that is to precede the plain English form of the disconnection reason, or the value NULL.

-> reason

For *reason*, specify the representation of the disconnection reason that was passed to the process by CMX when *t_disin()* was called.

Files

cmxlib.cat - Message file

See also

t_disin(), t_strreason()

8.6.24 t_redin - Accept redirected connection (redirection indication)

t_redin() accepts a T_REDIN event previously reported with *t_event()*. T_REDIN indicates that another process of the same TS application has redirected a connection to the current process.

The event T_REDIN **must** be accepted with *t_redin()*. If the connection is unwanted, it can be given back to the original process using *t_redrq()* or closed down using *t_disrq()*.

The *t_redin()* call returns

– the process ID of the calling process, and

– the user data that the calling process included with the redirection.

If the current process is attached for multiple TS applications, it must itself determine via suitable means the TS application to which the redirected connection belongs. Suitable means are, for example, the user data and the optional user reference to attachment of the TS application returned with *t_event()*.

```
#include <cmx.h>
int   t_redin (const int *tref,
               int *pid,
               t_opt23 *opt);
```

t_opt23 *opt;
: Pointer to a field with the transport reference of the connection.

<- pid
: Pointer to a field in which CMX returns the process ID of the redirecting process.

<> opt
: For *opt*, specify a NULL pointer or a pointer to a union with system options.

  This union is used to fetch user data that the calling process included with the redirection request (*t_redrq()*).

  If *opt* = NULL is specified, CMX discards the user data.

  If the calling process specified no user data, CMX returns the default values given.

The following structures are defined in <*cmx.h*>:

```
struct t_optc2 {
->    int  t_optnr;     /* Option no. */
<-    char *t_udatap;   /* Data buffer */
<>    int  t_udatal;    /* Length of the data buffer */
   };
   struct t_optc3 {
->    int  t_optnr;     /* Option no. */
<-    char *t_udatap;   /* Data buffer */
<>    int  t_udatal;    /* Length of the data buffer */
<-    int  t_xdata;     /* Choice for expedited data */
<-    int  t_timeout;   /* Inactive time */
->    int  t_ucepid;    /* User connection reference */
   };
->     int  t_optnr;     /* Option no.. */
<-     char *t_udatap;   /* Data buffer */
< >    int  t_udatal;    /* Length of the data buffer */
<-     int  t_xdata;     /* Choice for expedited data */
<-     int  t_timeout;   /* Inactive time */
->     int  t_ucepid;    /* User connection
                            reference */
<->    int  t_tid_valid; /* T_YES / T_NO */
<-     void *t_tid;      /* Pointer to thread ID */
};
```

t_optnr
: Option number. Specify:
T_OPTC2 in *t_optc2*
T_OPTC3 in *t_optc3*
T_OPTC4 in *t_optc4* for multithreading

t_udatap
: Pointer to a data area in which CMX enters the user data received.

Default value specifying *opt* = NULL: Undefined

t_udatal
: Prior to the call 0 or the length of the data area *t_udatap* must appear here. The area must be large enough that the received data completely fits. T_MSG_SIZE, defined in <*cmx.h*>, is a suitable maximum size. CMX returns in this field the number of bytes received.

Default value specifying *opt* = NULL: 0

t_xdata
: In *t_xdata* the value T_NO is always returned.

t_timeout
: In *t_timeout* the value T_NO is always returned.

t_ucepid
: This field can be used to pass a freely-selectable user reference for this connection to CMX.

  During subsequent processing this user reference can be returned to the current process by CMX as an option in a *t_event()* call.

  If the current process is maintaining multiple connections this mechanism enables it to associate a TS event with the appropriate connection via a user-defined attribute. The user reference constitutes an alternative to the transport reference *tref*, defined by CMX.

  Default value specifying *opt* = NULL: 0

t_tid_valid
: Marker showing that the field *t_tid* is valid; the value T_YES or T_NO must be entered here before the call. In the case of T_YES, the application expects CMX to supply the thread ID of the thread redirecting the connection in **t_tid*. In this case, the application has to provide an area of the size *pthread_t* in **t_tid*.

  Following the call, CMX returns the ID of the thread in **t_tid* that has redirected the connection and indicates this in the field *t_tid_valid* with T_YES; if the ID is not available, T_NO is set.

t_tid
: Pointer to the field of type *pthread_t* where CMX returns the thread ID of the thread redirecting the connection; *t_tid* is only valid if *t_tid_valid* contains the value T_YES. The field is ignored if T_NO is transferred in *t_tid_valid* before the call.

  See also the information about *t_redrq()* and connection redirection with multithreading on page 192.

Return values

T_OK
: The call was successful.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the pointers *opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or

  no T_REDIN was indicated for the connection specified by *tref*.

T_WPARAMETER
: The options specified in *opt* have an invalid format or contain illegal values.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_error(), t_event(), t_disrq(), t_redrq()

8.6.25 t_redrq - Redirect connection (redirection request)

t_redrq() redirects the specified connection to another process. The receiving process is specified with the parameter *pid*. It must be attached for the TS application to which the connection to be redirected belongs.

With *t_redrq()*, the current process may specify, in the options *opt*:

– a waiting period, during which time processing waits for the receiving process to be attached for the TS application, and

– user data to be passed to the receiving process when it accepts the connection. The user data can be used e.g. to inform the receiving process of the TS application to which the connection belongs.

Following the *t_redrq()* call the connection is no longer known to the calling process and the transport reference for it is invalid in the calling process. The called process receives the event T_REDIN.

The connection may not be redirected

– if T_DATASTOP or T_XDATSTOP is waiting for it, or

– while a TIDU on this connection is being fetched in piecemeal fashion with *t_datain()* (return value: n > 0).

```
#include <cmx.h>
int   t_redrq (const int *tref,
               const int *pid,
               const t_opt12 *opt);
```

-> tref

Pointer to a field with the transport reference of the connection to be redirected.

-> pid

Pointer to a field in which the process ID of the called process is to be specified.

-> opt

For the parameter *opt*, specify the value NULL or a pointer to a union with user options.

This union can be used to send information to the called process with the connection redirection. The called process receives this along with the connection redirection. If *opt* = NULL is specified, CMX delivers the given default values to the called process.

The following structures are defined in <*cmx.h*>:

```
   struct t_optc1 {
->    int  t_optnr;     /* Option no. */
->    char *t_udatap;   /* Data buffer */
->    int  t_udatal;    /* Length of the data buffer */
      int  t_xdata;     /* Choice for expedited data */
->    int  t_timeout;   /* Waiting period for
                           attachment */
   };
   struct t_optc2 {
->    int  t_optnr;     /* Option no. */
->    char *t_udatap;   /* Data buffer */
->    int  t_udatal;    /* Length of the data buffer */

    };
    struct  t_optc4 {
->     int  t_optnr;     /* Option no. */
<-     char *t_udatap;   /* Data buffer */
< >    int  t_udatal;    /* Length of the data buffer */
<-     int  t_xdata;     /* Choice for expedited data */
<-     int  t_timeout;   /* Waiting period for
                            attachment*/
->     int  t_ucepid;    /* User connection
                            reference */
->     int  t_tid_valid; /* T_YES / T_NO */
->     pthread_t t_tid;  /* Thread ID */
};
```

t_optnr
: Option number. Specify:
  T_OPTC1 in *t_optc1*
  T_OPTC2 in *t_optc2*
  T_OPTC4 in *t_optc4* for multithreading

t_udatap
: Pointer to a storage area with user data to be delivered to the receiving process.

  Default value specifying *opt* = NULL: Undefined

t_udatal
: Number of bytes to be transferred from the data area *t_udatap*. The maximum possible number is defined in <*cmx.h*> as T_RED_SIZE.

  If *t_udatal* = 0 is specified, *t_udatap* is ignored. The maximum value for t_udatal depends on the transport system (see the Release Notice).

Default value specifying *opt* = NULL: 0

t_xdata

This field has not yet been defined in this version. Specifications made for *t_xdata* will be ignored.

t_timeout

For this parameter a waiting period may be specified, in seconds. During this time the current process waits synchronously for the receiving process to be attached for the same TS application. Waiting is ended by the expected attachment or terminated by a signal.

If the waiting period elapses without the receiving process having attached itself for the proper TS application, or if waiting is terminated by a signal, the call ends with an error message.

Possible values for *t_timeout*:

T_NOLIMIT

No specific waiting period is defined. The process waits indefinitely for the receiving process to be attached.

T_NO

The process does not wait. It resumes immediately. If the receiving process is not attached as expected, the call ends with an error message.

n > 0

The current process waits n seconds for the attachment. During this time it is suspended. If the attachment does not take place within this time period, the call ends with an error message.

Default value specifying *opt* = NULL: T_NO

t_tid_valid

t_tid_valid indicates if a thread ID has been transferred to **t_tid*. With T_YES, **t_tid* contains the thread ID of the thread to which the connection is to be redirected. With T_NO, the field *t_tid* is not evaluated.

t_tid

Pointer to a field of type *pthread_t* containing the thread ID to which the connection is to be redirected. This thread must be in the **pid* process.

t_tid is only valid if *t_tid_valid* contains the value T_YES. The field is ignored if T_NO is transferred in *t_tid_valid* before the call.

Note about connection redirection in multithreading

The two functions *t_redrq()* and *t_redin()* use the process ID to identify the process to which the connection is to be redirected or to identify the process from which the connection will be received.

Connection redirection is possible to a thread of the same process or to the thread of another process.

The table below shows how the redirection is processed:

Caller PID	Receiver PID	Thread ID	Redirection
p1	p2	–	to the first available thread in the process p2
p1	p2	k	to thread k in process p2 (*)
p1	p1	–	to the first available thread in the same process
p1	p1	k	to thread k in the same process provided that k does not have the same thread ID

Table 8: Connection redirection in multithreading

Explanation:

(*) The caller must transmit the thread ID to the receiving process using, for example, interprocess communication.

"Available" here means that the reception criteria must be fulfilled (*t_conlim* is not yet fully used up, T_REDIRECT is set for *t_apmode* and is attached with the same LOCAL NAME).

Return values

T_OK
: The call was successful.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the pointers *opt* (!= NULL) or *t_udatap* (!= NULL and *t_ndatal* != 0) does not point to the process address space.

T_ETIMEOUT
: The waiting period of the current process, during which it waits for the receiving process to be attached for the same TS application, has elapsed.

T_WSEQUENCE
: The process is not attached for any TS application, or

  the connection specified in *tref* does not exist, or

  the flow of data on the connection *tref* has been blocked by the sending side, or

  a TIDU or an ETSDU has not yet been completely read.

T_WPARAMETER
: The process specified with the parameter *pid* is the current process,
  or the process specified in *pid* is not attached for this TS application,
  or the process specified in *pid* did not specify T_REDIRECT in *t_apmode* when attaching itself with *t_attach()*,
  or the options specified in *opt* have an invalid format or contain illegal values.

T_WCONN_LIMIT
: The process specified in *pid* has already used all the connections available to it.

T_WRED_LIMIT
: The limit for simultaneously permissible connection redirections has been exceeded.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datain(), t_error(), t_event(), t_xdatin()

8.6.26 t_setaddrpart - Add information to TRANSPORT ADDRESS

See section “t_getaddrpart, t_setaddrpart - Read or change address information in TRANSPORT ADDRESS” on page 166.

8.6.27 t_setlocpart

See section “t_getlocpart, t_setlocpart - Read or change address information in LOCAL NAME” on page 174.

8.6.28 t_setopt - Set options in CMX (set options)

t_setopt can be used to switch options on and off in CMX.

In this version the option T_DEBUG only is provided for activating/deactivating library traces.

```
#include <cmx.h>
int   t_setopt (int component,
                const t_opts *opt);
```

-> component

Specifies in which CMX component the option should be set.

Possible values:

T_LIB

The set option is a library option.

-> opt

Pointer to a union that contains an options structure.

The following structure is defined in <*cmx.h*>:

```
struct t_opts1 {
->    int t_optnr;        /* Option no.  */
->    int t_optname;      /* Option name */
->    char *t_optvalue;   /* Pointer to options string */
   };
```

t_optnr

Option number. Specify T_OPTS1.

t_optname
: Specifies the option that is to be switched on or off.

  Possible values:

  T_DEBUG
  : Activate/deactivate trace mechanism.

t_optvalue
: Pointer to a (NULL-terminated) string that contains the option value. If the string is empty, the option specified in *t_optname* is switched off. The contents of the string depends on the value of *t_optname*.

  t_optname = T_DEBUG: The format of the trace options is the same as for the environment variable CMXTRACE (see the manual „CMX, Operation and Administration“ [1] or [2]).

Return value

T_OK
: The call was successful.

T_ERROR
: Error. Query error code with *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_WPARAMETER
: The value specified in *component* is invalid or the option specified in *opt* has an incorrect format or contains incorrect values.

8.6.29 t_strerror - Decode CMX error message

t_strerror() decodes CMX error messages passed to the process in hexadecimal form by CMX when *t_error()* is called.

t_strerror() returns a pointer to a static area that contains the plain English form of the CMX error message specified in *code*.

This text consists of error symbols, as defined in <*cmx.h*>, and accompanying text. Each error symbol is preceded by \t. Each accompanying text ends with \n.

The accompanying text is taken from the message file *cmxlib.cat*. It will not be output if *cmxlib.cat* is not available on your system. The format of *cmxlib.cat* is dependent on the operating system and the set language variable. See the appropriate system manual for more details.

```
#include <cmx.h>
const char *t_strerror(int code);
```

-> code

For *code*, specify the representation of the error message that was passed to the process by CMX when *t_error()* was called.

Return values

If the call was successful, *t_strerror()* returns a pointer to a storage area with the plain English form of the CMX error message as a C string.

If an undefined value is specified in *code*, *t_strerror()* returns a pointer to the text:

```
"\t<code> Cannot decode\n"
```

In case of error, *t_strerror()* returns a NULL pointer.

Files

cmxlib.cat - Message file

See also

t_error(), t_perror()

8.6.30 t_strreason - Decode reasons for disconnection

t_strreason() decodes reasons for disconnection passed to the process in hexadecimal form when *t_disin()* is called.

t_strreason() returns a pointer to a static area that contains the plain English form of the reason for disconnection specified in *reason*.

This text consists of the symbol for the disconnection reason, as defined in <*cmx.h*>, and an accompanying text. The symbol for the disconnection reason is preceded by \t. The accompanying text ends with \n.

The accompanying text is taken from the message file *cmxlib.cat*. It will not be output if *cmxlib.cat* is not available on your system. The format of *cmxlib.cat* is dependent on the operating system and the set language variable. See the appropriate system manual for more details.

```
#include <cmx.h>
const char *t_strreason (int reason);
```

-> reason

For *reason*, specify the representation of the disconnection reason that was passed to the process by CMX when *t_disin()* was called.

Return values

If the call was successful, *t_strreason()* returns a pointer to a storage area with the plain English form of the disconnection reason as a C string.

If a value is specified in *reason* that is not defined, *t_strreason()* returns a pointer to the text:

```
"\t<reason> Cannot decode\n"
```

In case of error, *t_strreason()* returns a NULL pointer.

Files

cmxlib.cat - Message file

See also

t_disin(), t_preason()

8.6.31 t_vdatain - Receive data (data indication)

t_vdatain() accepts a T_DATAIN event previously reported via *t_event()*. The *t_vdatain()* call must be made before the next *t_event()*.

By means of this call the current process receives a Transport Interface Data Unit (TIDU) of the current Transport Service Data Unit (TSDU) from the sending TS application on the specified connection.

t_vdatain() places the data of a received TIDU into a series of non-contiguous storage areas. These storage areas are described by means of the array *vdata*. The number of storage areas, i.e. the number of elements in *vdata*, is specified in the parameter *vcnt*.

Thus, *vcnt t_data* structures are entered in *vdata*. Each *t_data* entry describes one of the storage areas vdata[0], vdata[1],..., vdata[vcnt-1].

The data received is stored in these storage areas sequentially; each storage area is completely filled before the next one is used.

Between two TIDUs of a TSDU any other CMX events can occur for the same or a different connection.

The maximum length of a TIDU depends on the transport system used. It can be queried for an established connection by means of *t_info()*.

A TIDU need not be completely full. The breakdown of a TSDU into TIDUs is purely local and does not indicate anything regarding the breakdown of the TSDU into TIDUs at the sending TS application.

t_vdatain() indicates:

– (in the *chain* parameter)

  whether a further TIDU belonging to the current TSDU exists (*chain* = T_MORE) or does not exist (*chain* = T_END).

  The individual TIDUs of a TSDU are each indicated via *t_event()* with the event T_DATAIN.

– (with the return value)

  whether the current TIDU has been completely read or not.

  If the value T_OK is returned, the TIDU has fit into the storage area provided. The current process has completely received the current TIDU.

If a value n > 0 is returned, only a part of the TIDU has been read. n is the number of bytes of the TIDU that have not yet been read (remaining length). In this case *t_vdatain()* or *t_datain()* must be called repeatedly until the entire TIDU has been read. Only then can other CMX calls be issued again, e.g. *t_event()*.

```
#include <cmx.h>
int   t_vdatain (const int *tref,
                 struct t_data *vdata,
                 int *vcnt,
                 int *chain);
```

-> tref

Pointer to a field containing the transport reference of the connection.

<> vdata

Pointer to an array of *t_data* structures for data buffers in which CMX enters the data of the received TIDU. The following structure is defined in <*cmx.h*>:

```
   struct t_data {
<-    char *t_datap;   /* Data area */
<>    int t_datal;     /* Length of the data area */
   };
```

t_datap

Pointer to a data area in which CMX enters data of the TIDU received.

t_datal

Prior to the call the length of the data area *t_datap* must be entered in *t_datal* (at least 1). Following the call, CMX returns in this field the number of bytes entered.

-> vcnt

Number of elements in *vdata*. At least 1 and at most T_VCNT must be specified.

<- chain

Pointer to an indicator used by CMX to show whether there is an additional TIDU belonging to the TSDU. Possible values:

T_MORE

Another TIDU belonging to the TSDU follows. It will be indicated with a separate T_DATAIN event.

T_END

The present TIDU is the last of the TSDU.

Return values

T_OK
: The call was successful. The TIDU was completely read.

n > 0
: n bytes remain from the TIDU.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the addresses specified in *vdata* does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or
no T_DATAIN was indicated for the connection specified in *tref*.

T_WPARAMETER
: The value specified in *vcnt* is invalid, or
at least one of the lengths specified in *vdata* is invalid.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.

  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datain(), t_error(), t_event(), t_info()

8.6.32 t_vdatarq - Send data (data request)

t_vdatarq() sends the next (or only) Transport Interface Data Unit (TIDU) of a Transport Service Data Unit (TSDU) to the receiving TS application on the specified connection.

The TIDU is provided in a series of non-contiguous storage areas.

These storage areas are defined by means of the array *vdata*. The number of storage areas, i.e. the number of elements in *vdata*, is specified in the parameter *vcnt*.

Thus, *vcnt t_data* structures are entered in *vdata*. Each *t_data* entry describes one of the storage areas vdata[0], vdata[1],..., vdata[vcnt-1].

CMX takes the data sequentially from these storage areas. Each storage area is completely read before the next one is used.

If the TSDU is longer than one TIDU, it must be transferred using several *t_vdatarq()* (or *t_datarq()* calls in succession. Therefore in each *t_vdatarq()* call the sending process must specify in the *chain* parameter whether an additional TIDU belonging to the same TSDU follows.

The maximum length of a TIDU depends on the transport system used. It can be queried for an established connection by means of *t_info()*.

If *t_vdatarq()* returns T_DATASTOP, the TIDU has been accepted but the flow of TIDUs on this connection has been blocked.

The flow of TIDUs can be blocked by:

– the receiving TS application,

  which can block the flow of TIDUs by calling *t_datastop()* or *t_xdatstop()*, or

– CMX,

  if the local buffer is full.

If the flow of TIDUs is blocked, before further TIDUs can be sent you must wait, by means of *t_event()*, for the event T_DATAGO for the connection.

Successful execution of *t_vdatarq()* (T_OK) does not mean that the receiving TS application has already accepted the data. If *t_vdatarq()* fails (T_ERROR), this always indicates that a local error has been found.

```
#include <cmx.h>
int   t_vdatarq (const int *tref,
                 const struct t_data *vdata,
                 const int *vcnt,
                 const int *chain);
```

-> tref

Pointer to a field containing the transport reference of the connection.

-> vdata

Pointer to an array of *t_data* structures for data buffers from which CMX takes the data of the TIDU to be sent. The following structure is defined in <*cmx.h*>:

```
   struct t_data {
->    char *t_datap;   /* Data area */
->    int t_datal;     /* Length of the data area */
   };
```

t_datap

Pointer to a data area from which CMX takes data of the TIDU to be sent.

t_datal

For this parameter, specify the length of the data area *t_datap*. At least 1 and at most the length of a TIDU must be specified.

-> vcnt

Number of elements in *vdata*. At least 1 and at most T_VCNT must be specified. The sum of the *t_datal* values of all *vcnt t_data* elements may not exceed the length of a TIDU.

-> chain

Pointer to an indicator used to show whether there is an additional TIDU belonging to the TSDU. Possible values:

T_MORE

Another TIDU belonging to the TSDU follows.

T_END

`The present TIDU is the last of the TSDU.`

Return values

T_OK
: The call was successful; further TIDUs may be sent immediately.

T_DATASTOP
: The call was successful, but further TIDUs may not be sent until the event T_DATAGO has arrived for the specified connection.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: At least one of the addresses specified in *vdata* does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or

  the process is not in the data phase for the connection specified in *tref*, or the flow of data is blocked.

T_WPARAMETER
: The value specified in *vcnt* or *chain* is invalid, or

  at least one of the lengths specified in *vdata* is invalid, or the sum of the lengths specified in *vdata* is invalid.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.
  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datarq(), t_datastop(), t_error(), t_event(), t_info(), t_xdatstop()

8.6.33 t_xdatgo - Release the flow of expedited data (expedited data go)

t_xdatgo() releases the blocked flow of expedited data on the specified connection. By means of this call the current process informs CMX that it is again ready to receive expedited data.

More specifically, the call has the following effects:

– The current process can again receive the event T_XDATIN for the specified connection, if one is waiting.

– The sending TS application receives the event T_XDATGO. It may again send data.

  Normal data is not affected by *t_xdatgo()*.

t_xdatgo() may be called only if the exchange of expedited data was agreed when the connection was set up.

```
#include <cmx.h>
int   t_xdatgo (const int *tref);
```

-> tref
: Pointer to a field with the transport reference of the connection on which the flow of expedited data is to be released again.

Return values

T_OK
: The call was successful.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_WSEQUENCE

The process is not attached for any TS application, or

the process is not in the data phase for the connection specified in *tref*, or the exchange of expedited data was not agreed for this connection.

T_CCP_END

The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_event(), t_error(), t_xdatstop()

8.6.34 t_xdatin - Receive expedited data (expedited data indication)

t_xdatin() accepts a T_XDATIN event previously reported via *t_event()*. The *t_xdatin()* call must be made before the next *t_event()*.

By means of this call the current process receives an Expedited Transport Service Data Unit (ETSDU) from the sending TS application on the specified connection. The maximum length of an ETSDU depends on the transport system used. However, it is never greater than T_EXP_SIZE bytes.

If the expedited data fits into the storage area *datap* provided, the value T_OK is returned. Otherwise, a value n > 0 is returned, where n is the number of bytes of the ETSDU that have not yet been read (remaining length). In this case, *t_xdatin()* must be called repeatedly until the entire ETSDU has been read. Only then can other CMX calls be issued again, e.g. *t_event()*.

```
#include <cmx.h>
int   t_xdatin (const int *tref,
                char *datap,
                int *datal);
```

-> tref
: Pointer to a field containing the transport reference of the connection, obtained via *t_event()*.

<- datap
: Pointer to a storage area in which CMX enters the data of the ETSDU received.

<> datal
: Pointer to a field in which prior to the call the length of the data area *datap* must be entered. A value of at least 1 must be specified.

  Following the call, CMX returns in this field the number of bytes entered.

Return values

T_OK
: The call was successful. The expedited data was completely read.

n > 0
: n bytes remain from the ETSDU.

T_ERROR
: Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT
: The pointer *datap* does not point to the process address space.

T_WSEQUENCE
: The process is not attached for any TS application, or

  the exchange of expedited data was not agreed for the connection specified in *tref*, or no T_XDATIN was indicated for the connection specified in *tref*.

T_WPARAMETER
: The length specified in *datal* is invalid.

T_COLLISION
: The event T_DISIN (disconnect indication) occurred for the connection, but has not yet been queried with *t_event()*.

  Action: call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_error(), t_event()

8.6.35 t_xdatrq - Send expedited data (expedited data request)

t_xdatrq() sends an Expedited Transport Service Data Unit (ETSDU) with expedited data to the receiving TS application via the connection specified. The maximum length of a ETSDU depends on the transport system used. However, it is never greater than T_EXP_SIZE bytes (does not apply to WAN-X25).

The *t_xdatrq()* call is permitted only when the exchange of expedited data was agreed when the relevant connection was set up.

ETSDUs may overtake Transport Interface Data Units (TIDUs) with normal data that had been sent earlier. It is guaranteed that ETSDUs will never arrive at the receiving TS application later than TIDUs sent after them.

If T_XDATSTOP is returned, the ETSDU has been accepted but the send flow of ETSDUs and TIDUs on this connection has been blocked.

The flow of expedited data can be blocked by:

– the receiving TS application,

  which can block the flow of ETSDUs by calling *t_xdatstop()*, or

– CMX,

  if the local buffer is full.

If the flow of ETSDUs is blocked, before further ETSDUs can be sent you must wait, by means of *t_event()*, for the event T_XDATGO or T_DATAGO for the connection.

Successful execution of *t_xdatrq()* (T_OK) does not mean that the receiving TS application has already accepted the data.

If *t_xdatrq()* fails (T_ERROR), this always indicates that a local error has been found.

```
#include <cmx.h>
int   t_xdatrq (const int *tref,
                const char *datap,
                const int *datal);
```

-> tref

  Pointer to a field with the transport reference of the connection on which the expedited data is to be sent.

-> datap

Pointer to a storage area containing the ETSDU to be sent.

-> datal

Pointer to a field containing the number of bytes to be sent from the storage area *datap*.

Minimum value: 1

Maximum value: T_EXP_SIZE

(T_EXP_SIZE is defined in <*cmx.h*>.)

Return values

T_OK

The call was successful; further expedited data may be sent immediately.

T_XDATSTOP

The call was successful, but further ETSDUs may not be sent until the event T_XDATGO or T_DATAGO has arrived for this connection.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_EFAULT

The pointer *datap* does not point to the process address space.

T_WSEQUENCE

The process is not attached for any TS application, or

the process is not in the data phase for the connection specified in *tref*, or the exchange of expedited data was not agreed for the connection specified in *tref*, or the flow of expedited data is blocked for the connection specified in *tref*.

T_WPARAMETER

The length specified in *datal* is not permitted.

T_COLLISION
: The event T_DISIN (disconnect indication) has arrived for the connection, but has not yet been fetched with *t_event()*.

  Response: Call *t_event()*.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_error(), t_event(), t_xdatstop()

8.6.36 t_xdatstop - Block the flow of expedited data (expedited data stop)

t_xdatstop() blocks the flow of both expedited and normal data on the specified connection.

More specifically, the effects of *t_xdatstop()* are:

– The current process tells CMX that, until further notice, it is not ready to receive normal or expedited data for this connection. However, a T_DATAIN event or a T_XDATIN event that has already been indicated must be responded to first.

– The current process no longer receives the events T_DATAIN and T_XDATIN for the specified connection. However, while the data flow is blocked it may call other CMX functions, e.g. to set up, close down or redirect an additional connection.

– The sending TS application receives the return value T_XDATSTOP when it calls *t_xdatrq()* and the return value T_DATASTOP when it calls *t_datarq()*. It may not send any more normal or expedited data.

The flow of expedited data is released with *t_xdatgo()* or with *t_datago()*.

t_xdatstop() may be called only if the exchange of expedited data was agreed when the connection was set up.

```
#include <cmx.h>
int   t_xdatstop (const int *tref);
```

-> tref

Pointer to a field with the transport reference of the connection.

Return values

T_OK

The call was successful.

T_ERROR

Error. Query error code using *t_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *t_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the following may occur:

T_WSEQUENCE
: The process is not attached for any TS application, or the process is not in the data phase for the connection specified in *tref*, or a TIDU or an ETSDU has not yet been completely read on the specified connection, or the exchange of expedited data was not agreed for this connection.

T_CCP_END
: The CCP is no longer operational.

In addition, the errors listed under *ioctl(2)* may occur.

See also

t_datago(), t_error(), t_event(), t_xdatgo(), t_xdatrq()

9 The ICMX(NEA) program interface

This chapter describes the ICMX(NEA) program interface to the NEABX migration service. It contains:

– A summary of the functions of the ICMX(NEA) interface, with details on the communication phases,

– Notes on the correct use of the functions (finite-state automata),

– Precise descriptions of the ICMX(NEA) function calls, with all parameters, in alphabetical order.

Notes on the availability of the system options for the transport systems are contained in the Release Notice.

9.1 Overview of the program interface

The program interface ICMX(NEA) implements the migration service NEABX. There are TS applications (e.g. UTM applications) in host computers and communication computers of the TRANSDATA family that require the NEA-specific services of the transport system. This means that such TS applications require services which go beyond the functionality of an ISO transport system as defined in the ISO standard 8072. The migration service NEABX enables communication between a TS application on your system (i.e. CMX applications) and TS applications in a host or communication computer without the need to modify the communication interface of the TS application at the remote system. In other words, ICMX(NEA) offers the functions that are needed to bridge the gap between the services required and the services provided by an ISO transport system.

The primary purpose served by ICMX(NEA) functions is to set up, exchange, and interpret the NEABV protocol in the connection setup phase and the NEABX protocol in the data phase.

ICMX(NEA) is implemented as a set of C functions. Each ICMX(NEA) function is internally converted by the NEABX into one or more ICMX(L) functions.

ICMX(NEA) - Interface to the connection-oriented transport service

ICMX(NEA) is offered by the migration service NEABX as a program interface to the connection-oriented transport service (TS). This TS permits two applications (TS applications) to exchange messages over a transport connection (TC). When communication takes place via a connection-oriented TS, messages are exchanged without loss or duplication, and the sequence of messages is maintained. Once a TC is established, it is assigned a transport reference (tref) in each of the two end systems. These transport references identify the TC uniquely and thus make it possible to dispense with the transfer and processing of addresses in the data phase. Parameters that influence the transport of messages on the TC can be negotiated between the TS applications at connection setup. There are also some rules that must be observed to ensure that communication proceeds smoothly. These rules are described below.

ICMX(NEA) makes communication between TS applications largely independent of the specific characteristics of the transport systems used (layers 1 - 4 in the OSI Reference Model) with regard to profile, protocol classes, etc.

Internally each TC is assigned an exclusively-opened special file, which, however, is invisible to the TS application. The exclusive opening simplifies the measures taken in NEABX to tidy up following premature termination of the TS application.

Names and addresses

Every TS application has a GLOBAL NAME. This name uniquely identifies the TS application in the network. GLOBAL NAMES are assigned by the administration. It must ensure that the names of all TS applications are different from one another.

A TS application works exclusively with GLOBAL NAMES. When attaching itself it specifies its own GLOBAL NAME, and at connection setup it specifies the GLOBAL NAME of the communication partner.

The TNSX, a component of CMX, converts the GLOBAL NAMES into the LOCAL NAME of the local TS application and the TRANSPORT ADDRESS of the remote TS application, and vice-versa.

For querying the LOCAL NAME of a TS application and the TRANSPORT ADDRESS of a communication partner the ICMX(L) interface provides the functions *t_getloc()* and *t_getaddr()*

<*neabx.h*> defines the structures *x_myname* and *x_partaddr*.

x_myname is used by a TS application to receive (pass) its LOCAL NAME from (to) the TNSX; *x_partaddr* is used similarly for the TRANSPORT ADDRESS.

The contents of these structures are as follows:

```
struct x_myname {
   char  x_mnmode;        /* = X_MNMODE */
   char  x_mnres;         /* = 0 */
   short x_mnlng;         /* Length of the filled-in part
                             of x_myname */
   char  x_mn[X_MNSIZE];  /* Field for the T-selectors of
                             the LOCAL NAME */
}
struct x_partaddr {
   char  x_pamode;        /* = X_PAMODE */
   char  x_pares;         /* = 0 */
   short x_palng;         /* Length of the filled-in part
                             of x_partaddr */
   char  x_pa[X_PASIZE];  /* Field for the partner address */
}
```

The meanings of members in the structure *x_myname* are shown below:

x_mnmode = X_MNMODE
: specifies that the field *x_mn* contains a LOCAL NAME.

x_mnres, *x_mn*[X_MNSIZE]
: are of no relevance to you. The contents of these fields are simply taken from the TNSX and passed on to NEABX.

x_mnlng
: specifies the length of all data passed in the *x_myname* structure.

The meanings of members in the structure *x_partaddr* are as follows:

x_pamode = X_PAMODE
: specifies that the field *x_pa* contains a TRANSPORT ADDRESS.

x_pares, *x_pa*[X_PASIZE]
: are of no relevance to you. The contents of these fields are simply taken from the TNSX and passed on to NEABX.

x_palng
: specifies the length of all data passed in the *x_partaddr* structure. The LOCAL NAME and TRANSPORT ADDRESS are passed to NEABX or received from NEABX in the union *x_address*

```
union x_address {
struct x_myname xmyname;
struct x_partaddr xpartaddr;
}
```

Error handling and diagnosis

All function calls terminate with a return code. One example of such a code is T_OK, which indicates successful completion. When an error occurs, the function returns the value X_ERROR as a general indication of the error. You can then obtain more detailed diagnostic information by using the function *x_error()*. This function must be called immediately after the error occurs.

All errors detected by NEABX as violations of the communications rules by the TS application have specific error codes and are defined in <*neabx.h*>, <*cmx.h*> or <*tnsx.h*>. The structure of these error codes is described in the chapter “Event processing and error handling” on page 37.

Other errors result from failures when calling functions in the operating system environment in CMX; they are described in <*errno.h*>.

The transport systems used generate no error messages. If an error occurs here, the connection is closed down, and a reason for disconnection is passed to CMX. This reason for disconnection is delivered to the TS application when it calls *x_disin()*.

The following functions return the text string in plain English for an error code returned by *x_error()*:

x_strerror()
: Returns a pointer to the text string for an error code received from ICMX(NEA).

x_perror()
: Calls *x_strerror()* to ascertain the text string for an error code received from ICMX(NEA) and writes the string to *stderr*.

  The codes returned by *x_disin()* for a disconnection reason can be converted into plain English by using the ICMX(L) functions *t_strreason()* and *t_preason()*.

The error code and reason for disconnection can be edited at the command level in plain English by using the program *cmxdec*.

For diagnostic purposes ICMX(NEA) provides a trace facility, which can be flexibly controlled via the environment variable NEATRACE. The trace mechanism logs the calls with their arguments in compressed form in temporary files. The editing program *neal* then converts the log to plain English in a separate step (see the “CMX, Operation and Administration” manual [1] or [2]).

TS applications, transport connections, and processes

A TS application is a system of programs that uses the TS, i.e. the services of NEABX. The mapping of a TS application to the process concept of the system is left up to the implementer. A TS application may organize itself into one or more (not necessarily related) processes. The processes may, essentially independently from one another, maintain TCs to remote TS applications. The processes of a TS application may exchange their TCs among one another. However, at any point in time the transport reference of a TC is assigned to exactly one process. It therefore cannot be inherited by child processes. In NEABX there is a separate local service, REDIRECT, for redirecting a TC to another process.

One process may also simultaneously control multiple TS applications. In this case, the implementation must provide for suitable coordination of the execution of the various TS applications. NEABX supports this through its asynchronous processing mode.

Synchronicity and asynchronicity; TS events

Communications operations are by nature asynchronous, i.e. different TS events can occur independently of the activity of a TS application. For example, a TS application may be sending data over a TC when, asynchronously, a disconnection indication arrives, of which the TS application must be informed immediately.

In principle, the functions of NEABX are asynchronous. This means that after issuing a call a TS application need not wait for a possible answer (TS event) from the network. Any answer will be accepted by NEABX when it arrives and sent to the TS application at the next opportunity when requested.

To do this, NEABX provides the TS application with a query mechanism in two forms: synchronous (waiting) and asynchronous (checking). This query mechanism must be appropriately used by the TS application if it wishes to react quickly and properly to TS events.

With synchronous execution, the calling process is suspended until a TS event arrives. This wakes up the process, so that it can immediately process the TS event. Waiting can be limited by specifying a waiting period or it can be cut short by a signal such as SIGALARM. In both cases NEABX continues the process with the TS event X_NOEVENT. The synchronous mechanism is useful for TS applications that cannot do anything between TS events.

With asynchronous execution, at convenient times, such as at the end of a processing step, the process can check whether a TS event has arrived, and handle it before continuing with the next processing step. This is useful for processes that expect longer delays between TS events, during which times they can or must attend to other operations.

The relevant function in NEABX is

x_event
: If the parameter value X_WAIT is passed, the process is suspended until a TS event arrives, the time limit expires or a signal arrives. The suspended process is awakened when a signal arrives, and *x_event()* returns with X_NOEVENT or X_ERROR. If the time limit expires, the process resumes with the TS event X_NOEVENT. If a TS event arrives, or there is an error, the function immediately returns the code of the TS event, or X_ERROR.

  When the parameter value X_CHECK is passed, *x_event()* always returns immediately and returns X_NOEVENT or the code of the TS event encountered or X_ERROR.

The following thirteen asynchronous TS events are defined in NEABX:

X_NOEVENT
: In the asynchronous case: No TS event present; In the synchronous case: Function aborted by a signal, or time limit expired.

X_REPCRQ
: NEABX requests repetition of the connection request.

X_CONIN
: Indication of a connection request arriving from a calling TS application.

X_REPCIN
: NEABX indicates the continuation of a connection request that has not yet been completely passed.

X_CONCF
: Indication of a connection confirmation arriving from a called TS application.

X_REPCCF
: NEABX indicates the continuation of a connection confirmation that has not yet been completely passed.

X_DISIN
: Disconnection indication arriving from a remote TS application or caused by NEABX.

X_REDIN
: Indication of a connection redirection arriving from a process of the same TS application. (This TS event is local; it is an extension of the TSs to make the implementation of TS applications more flexible.)

X_DATAIN
: Normal data sent by a TS application has arrived.

X_XDATIN
: Expedited data sent by a TS application has arrived.

X_DATAGO
: A block on the sending of normal data set through flow control is canceled.

X_XDATGO
: A block on the sending of expedited data set through flow control is canceled.

X_ERROR
: Fatal error; more detailed information will be returned by the query function *x_error()*.

With each TS event (except for X_NOEVENT and X_ERROR) the TS application is also given the transport reference, so that it can react to the TS event in a way specific to that TC.

Some TS events must be accepted by the TS application by calling corresponding functions. Exceptions are: X_DATAGO, X_XDATGO. Such function calls return additional information on the TS events. The following table lists the TS events and the corresponding functions.

TS event	**Function for fetching**
X_CONCF	x_concf()
X_CONIN	x_conin()
X_DATAIN	x_datain()
X_DISIN	x_disin()
X_ERROR	x_error()
X_REDIN	x_redin()
X_REPCRQ	x_conrq()
X_REPCCF	x_concf()
X_REPCIN	x_conin()
X_XDATIN	x_xdatin()

As a rule, TS events are delivered in the order in which they occur. Of course, the TS event X_XDATIN may overtake the TS event X_DATAIN, and X_DISIN may overtake X_DATAIN and X_XDATIN. In the latter case the overtaken TS events on that TC are dropped.

Attaching to/detaching from NEABX

Communication by a TS application via NEABX is activated when the first process attaches itself to NEABX for that application. When this is done, a special file is opened exclusively for that process. This special file is used for exchanging jobs between the NEABX library functions and the operating system. A TS application is generated when the first process attaches itself for that TS application. When this is done, a Transport Service Access Point (TSAP) is created, at which the TS is accessible. When the first process is attached the TS application is linked to this TSAP. The TSAP is assigned the LOCAL NAME of the TS application, under which the TS application can be reached in that end system. It thereby becomes addressable from the network.

When the TS application is detached, any TCs still in existence are closed down, along with the TSAP; the process environment is dissolved and assigned resources are released for future use.

One and the same process may attach itself for several TS applications at once (i.e. manage multiple TSAPs) and in each of these TS applications maintain multiple Transport Connection Endpoints (TCEP). Also, several processes may attach themselves for the same TS application (use the same TSAP) and actively set up TCs or passively wait for connection indications without interfering with one another. However, each TCEP is assigned to exactly one process.

The following functions are used for attaching and detaching. They perform primarily local tasks. If no implicit disconnection must be performed, no information is passed to the network.

x_attach()
: Attaches (the current process of) a TS application to NEABX. When attached, the process may specify its future behavior in the TS application. The first time a process is attached NEABX begins accepting connection indications for the TS application.

x_detach()
: Detaches (the current process of) a TS application from NEABX. Any existing TCs of the process in the TS application are closed down by CMX. If no more processes of the TS application are attached, the TS application is thereafter no longer known to NEABX.

Connection setup, disconnection, and redirection

In this phase, two TS applications set up a TC between them or close one down. One of the two TS applications is viewed as the calling TS application; it initiates connection setup. The other is the called TS application; it waits for requests from calling TS applications.

The calling TS application issues a connection request and receives an answer from the called TS application. The called TS application waits for a connection indication (indication of a connection request) and accepts it or rejects it. During connection setup, the TS applications negotiate certain attributes of the TC for the data transfer and may exchange user data.

The TC may be closed down at any time by either of the TS applications or by NEABX. This is not negotiated between the TS applications, but instead is immediately carried out by NEABX. The other TS application (or both if NEABX closes down the TC) receives a disconnect indication, which may be neither answered nor averted. NEABX may also close down the TC at any time; all errors in the transport systems are indicated in this way. NEABX does not guarantee that data in transit at the time of the disconnection request will still be delivered.

Connection redirection is a local service in NEABX that simplifies organizing a TS application into processes. A process holding a completely established TC may redirect it (depending, of course, on the state; see section “Finite-state automata” on page 226) to another process of the same TS application. The TSAP and the TCEP remain unchanged. The redirecting process loses the transport reference for the TC, whereupon the TC is no longer available to the process.

The relevant functions are:

x_conrq()

Requests connection setup to the called TS application with the specified TRANSPORT ADDRESS. Reference to the TSAP is established via the LOCAL NAME used when the calling TS application was attached. The function returns immediately after issuing the request; the calling TS application receives a transport reference. It must then wait synchronously or asynchronously for the answer of the called TS application (see above).

The NEABV protocol must be passed in the form of user data when *x_conrq()* is called. This user data is passed to NEABX in an option structure.

If *x_conrq()* returns the value X_REPEAT, it must be called again with the same parameters after the event X_REPCRQ has arrived as the user data has not yet been completely transmitted.

x_conin()

Accepts from NEABX the connection request of the calling TS application, indicated with X_CONIN, with its TRANSPORT ADDRESS. Reference to the TSAP is established for the called TS application through provision of the LOCAL NAME specified when it was attached.

The NEABV protocol is passed in the form of user data when the connection is set up. The called TS application must accept the user data from NEABX in an option structure.

If *x_conin()* returns the value X_REPEAT, it must be called again with the same parameters after the event X_REPCIN has arrived as the user data has not yet been completely received.

x_conrs()
: Answers (accepts) the connection request, after it has been indicated with X_CONIN and received from NEABX. The NEABV protocol must be passed in the form of user data when *x_conrs()* is called. This user data is passed to NEABX in an option structure.

x_concf()
: Accepts from NEABX the answer of the called TS application, indicated with X_CONCF; this completes connection setup. The NEABV protocol is passed in the form of user data when the connection is set up. The TS application must accept the user data from NEABX in an option structure.

  If *x_concf()* returns the value X_REPEAT, it must be called again with the same parameters after the event X_REPCCF has arrived as the user data has not yet been completely received.

x_disrq()
: Requests disconnection. This function may be called at any time by either of the TS applications. A connection request that has been indicated by NEABX and received may, if it is not accepted, be rejected with this function.

x_disin()
: Accepts from NEABX the disconnect indication indicated with X_DISIN. With this function call the TS application also obtains the reason for disconnection.

x_redrq()
: Redirects the TC to another process of the same TS application. The TC is then no longer available to the redirecting process.

  An option structure must be specified when *x_redrq()* is called. The migration service of the redirecting process passes information to the migration service of the receiving process in this structure.

  If *x_redrq()* returns the value X_IMPOSSIBLE, the redirection request cannot be executed.

x_redin()
: Receives from NEABX the connection redirection indicated with X_REDIN. The receiving process must accept the redirection, but may immediately pass it on (or return it) or close down the connection. An option structure must be specified when *x_redin()* is called. The migration service of the receiving process receives information from the migration service of the redirecting process in this structure.

Data exchange and flow control

Once a TC has been set up, normal data and (optionally) expedited data may be transferred over it. Data transfer is message-oriented: The TS applications exchange Transport Service Data Units (TSDU) - messages of any length - or Expedited Transport Service Data Units (ETSDU) - expedited data of limited length. Expedited data is limited to a few bytes; when transferred it is given priority over the stream of normal data and placed into separate queues. NEABX guarantees only that expedited data will never arrive at the receiving TS application later than normal data sent subsequently. At most one complete ETSDU may be passed to NEABX per call.

A message is passed to NEABX in portions the length of one Transport Interface Data Unit (TIDU). The length of the TIDU is TC-specific and must therefore be queried (*x_info()*) by NEABX for each TC. If a message is longer than one TIDU, it must be transferred using multiple send calls. A parameter in each send call indicates whether a further TIDU for that message follows (X_MORE) or not (X_END). It cannot be determined from this how a TIDU is packed for transfer or delivery to the receiving TS application. NEABX guarantees only that sequential joining of the TIDUs on the receiving side will reproduce the message from the sending side. The TIDU length may be different for the two TS applications and depends on the TC. NEABX does not guarantee that at the receiving TS application any except the last TIDU of a message will be delivered completely filled.

The arrival of a TIDU or an ETSDU is indicated to the receiving TS application by means of the TS event X_DATAIN or X_XDATIN. The TS application then completely fetches the TIDU or ETSDU with a corresponding function call. The function *x_event()* returns the length of the data as additional information in the option structure.

The transfer of TIDUs and ETSDUs is subject to flow control mechanisms, which can be controlled by NEABX and the TS applications. The return code X_DATASTOP or X_XDATSTOP returned when data is sent indicates to the sending TS application that the TIDU or ETSDU was successfully processed, but the flow of TIDUs (ETSDUs) has been blocked. No further TIDU (ETSDU) may be sent until the flow is released again. Release is indicated by means of the TS event X_DATAGO (X_XDATGO).

The receiving TS application stops and starts the flow of TIDUs and ETSDUs by means of function calls to NEABX, which affect the sending TS application as described above.

The following functions implement data exchange and (active) flow control:

x_datarq()
: Requests transfer of a TIDU (possibly partially filled). The return code X_DATASTOP signifies that the data flow is blocked; further send requests are rejected with an error until the flow is released again.

  If the exchange of the NEABX protocol in the data phase was agreed when setting up the connection, an option structure for exchanging the NEABX protocol must be specified with the **first** TIDU of a TSDU. The specification of an option structure is **not permitted** for any further TIDU of the TSDU. If the exchange of the NEABX protocol was not negotiated, **no** option structure may be specified.

x_datain()
: Accepts the data of a TIDU from NEABX after the TIDU has been indicated with X_DATAIN. If the exchange of the NEABX protocol in the data phase was agreed when setting up the connection, an option structure for exchanging the NEABX protocol must be specified when the **first** TIDU of a TSDU is received. The specification of an option structure is **not permitted** for any further TIDU of the TSDU. If the exchange of the NEABX protocol was not negotiated, **no** option structure may be specified.

x_xdatrq()
: Requests transfer of an ETSDU with expedited data. The return code X_XDATSTOP indicates that the flow of ETSDUs is blocked; further send requests are rejected with an error until the flow is released again.

  If the exchange of the NEABX protocol in the data phase was agreed when setting up the connection, an option structure for exchanging the NEABX protocol must be specified. If the exchange of the NEABX protocol was not negotiated, **no** option structure may be specified.

x_xdatin()
: Accepts the expedited data from NEABX after it has been indicated with X_XDATIN.

  If the exchange of the NEABX protocol in the data phase was agreed when setting up the connection, an option structure for exchanging the NEABX protocol must be specified when receiving the ETSDU. If the exchange of the NEABX protocol was not negotiated, **no** option structure may be specified.

Information service

The information service is a local service with which the TS application can query configuration-dependent parameter values from NEABX. The information service is implemented with the following function:

x_info()
: Returns the length of a TIDU for an established TC. The TIDU can generally only be established when the connection setup has been completed.

9.2 Finite-state automata

The sequences for using the ICMX(NEA) interface can be represented by means of finite-state automata. These are diagrams that contain the defined states of a TS application and the legal state transitions.

The sequences may be divided into four phases. These form a hierarchical structure:

Phase A:	activation/deactivation
Phase B:	attach/detach
Phase C:	connection setup/closing down
Phase D:	data exchange

Phase A corresponds to the highest hierarchical level and phase D corresponds to the lowest.

Each phase is represented by one or more automata. The double rectangles of an automaton represent the state in which the automata of the next phase (or lower hierarchical level) are activated. The transition from a state represented by a double rectangle (see figure “Setup, disconnection, and redirection of TCs” on page 228) to another state causes the associated automata of the lower hierarchical level to be deactivated.

In phase A, a process must be present that can support the ICMX(NEA) interface. In this phase the process is created and destroyed.

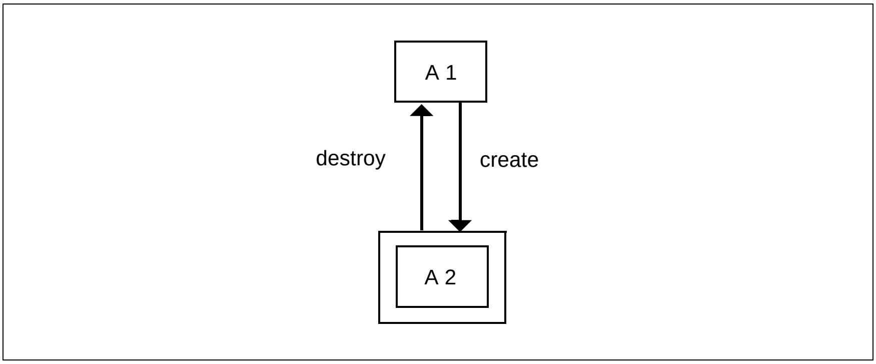


Figure 30: Activation/deactivation

In phase B TS applications are attached and detached. The number of automata in this phase is equal to the number of TS applications executed by the process. In B2 the process has created an (additional) service access point (TSAP).

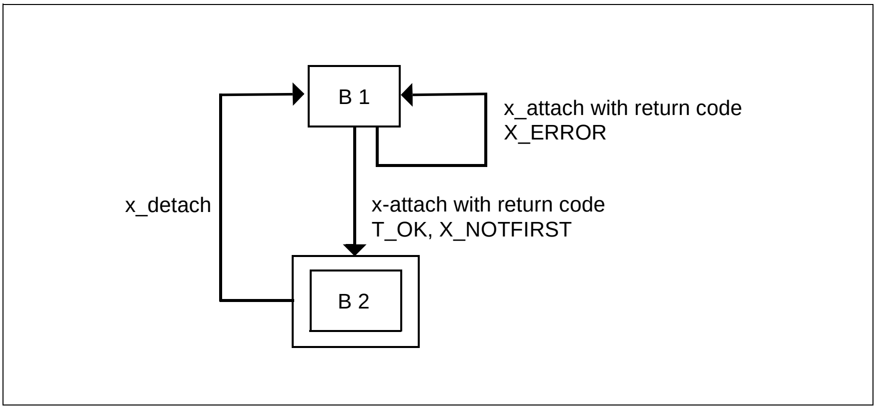


Figure 31: Attaching/detaching TS applications

In phase C TCs are set up, closed down and redirected. The number of automata is equal to the number of TCs supported by the process. In C2 the TS application has actively requested a TC and is waiting for the answer from the called TS application. In C3 the TS application has passively received a TC request. In C4 the connection is established. In the states marked with "+" (after X_REPEAT) the TS application is waiting for the repetition request. In the states marked with "*" the repetition may take place.

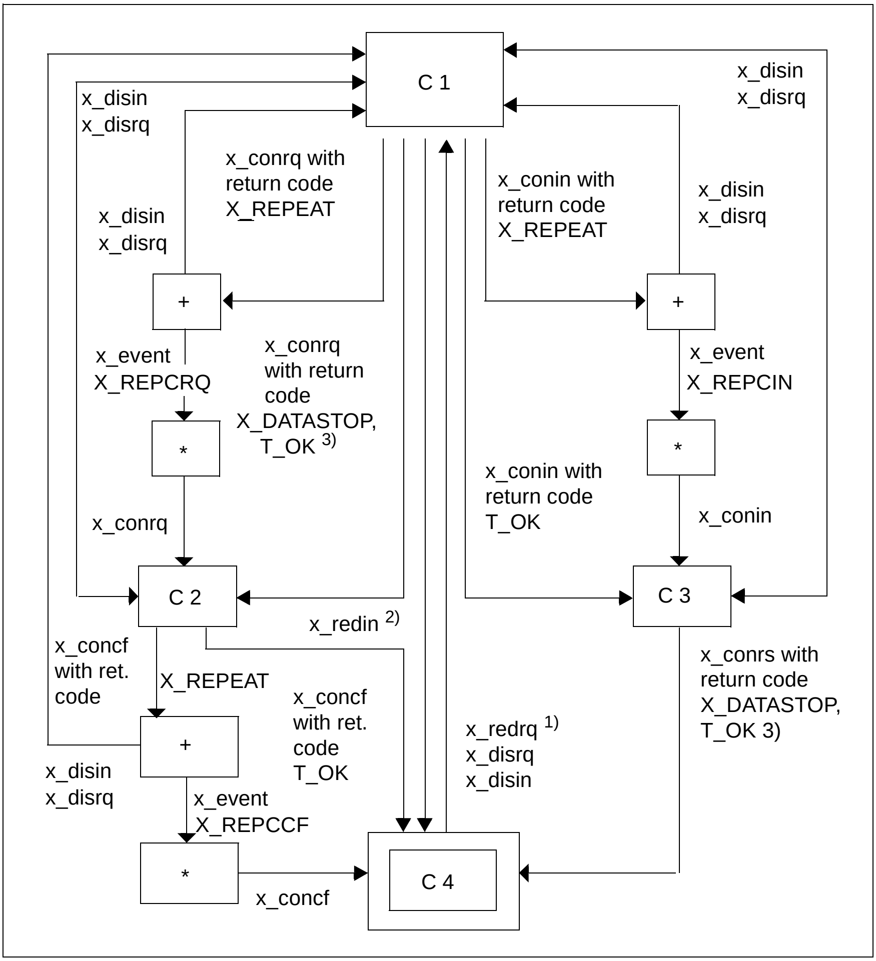


Figure 32: Setup, disconnection, and redirection of TCs

1. x_redrq() is permitted only when the corresponding data sending automaton is in D1 and the data receiving automata are in D1 and D4.
2. x_redin() causes the data automata to switch to the states corresponding to the states of the data automata of the process that initiated the x_redrq().
3. With x_conrq() and x_conrs() the return value X_DATASTOP functions in the connection phase like T_OK; the corresponding data sending automaton switches at the same time to the state D2.

Phase D is the data phase. It is represented by 2 parallel automata (data sending automaton, data receiving automaton). There are thus n automata, where n = 2 * number of TCs. In D2 the data flow for normal data is stopped, in D3 the flow of expedited data is also stopped.

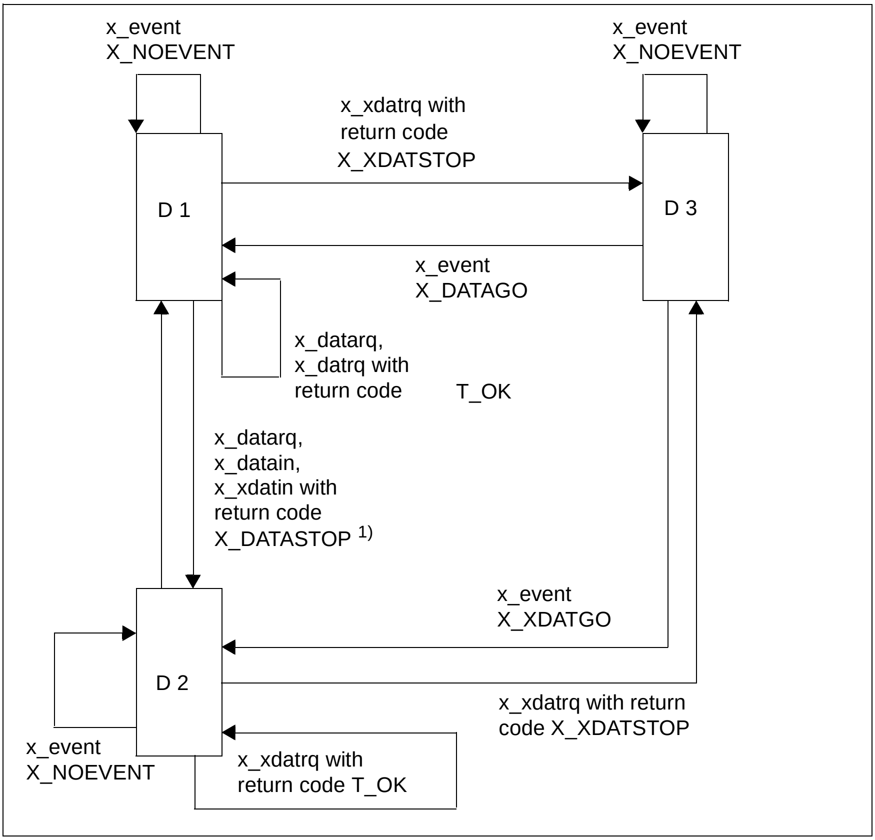


Figure 33: Data sending automaton for normal and expedited data

1. With return code X_DATASTOP following x_conrq() or x_conrs() the corresponding data sending automaton switches to the state D2.

   For the sake of clarity, the data receiving automaton is represented in two separate automata (for normal data and for expedited data). It is to be noted that *x_event()*, *x_xdatstop()* and *x_datago()* work on both automata. In D2 normal data or expedited data ready to be received can be accepted, subsequently in D1 the (expedited) data flow may be actively stopped and in D4 released again. In D3 no flow control is possible.

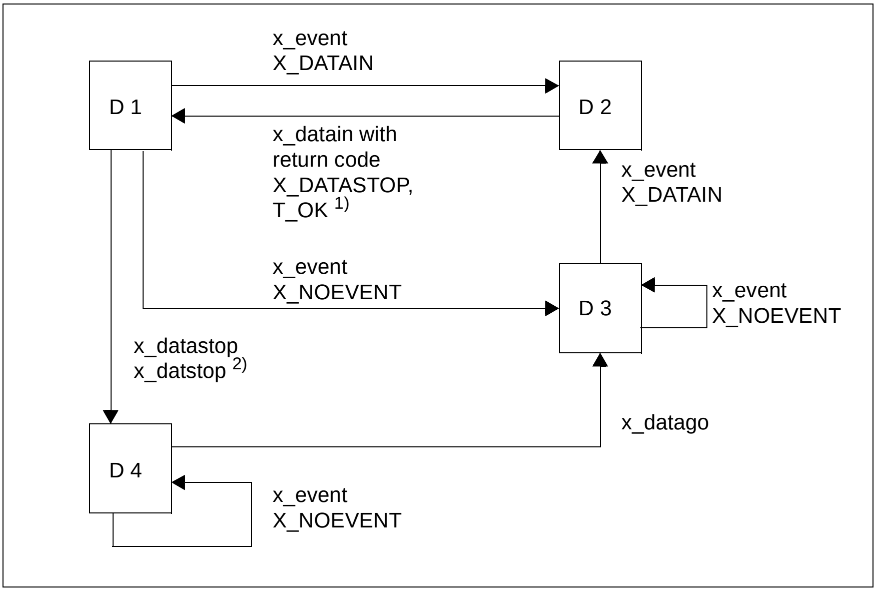


Figure 34: Data receiving automaton for normal data

1. With return code X_DATASTOP the corresponding data sending automaton switches to the state D2.
2. x_xdatstop() is permitted only when the data receiving automaton for expedited data is in D1.

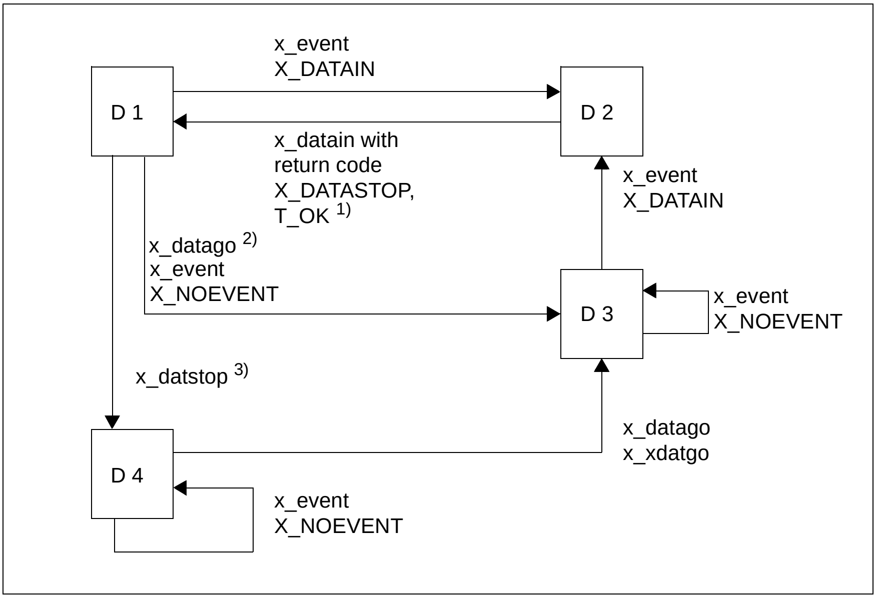


Figure 35: Data receiving automaton for expedited data

1. With return code X_DATASTOP the corresponding data sending automaton switches to the state D2.
2. x_datago() is permitted only when the data receiving automaton for normal data is in D4.
3. x_xdatstop() is permitted only when the data receiving automaton for normal data is in D1.

9.3 NEABV protocol

9.3.1 The NEABV protocol for communication via ICMX(NEA)

If your TS application is to communicate with a TS application that requires TRANSDATA-specific functions of the transport protocol, you must adhere to the user services connection protocol (NEABV, SIEMENS standard SN 77303) when setting up a connection via the migration interface ICMX(NEA).

Communication partners for whom you must adhere to the NEABV protocol may be

– UTM applications (from the point of view of UTM: PTYPE=APPLI)

– DCAM applications (from the point of view of DCAM: EDIT=USER)

– PDN applications (from the point of view of PDN: Partner characteristic with YOPNCON=application)

The following practical notes are intended to make it possible to program connection setup via ICMX(NEA) in a way that is consistent with the protocol even without detailed knowledge of the standard.

At connection setup the NEABV protocol is transferred in the form of structured user data.

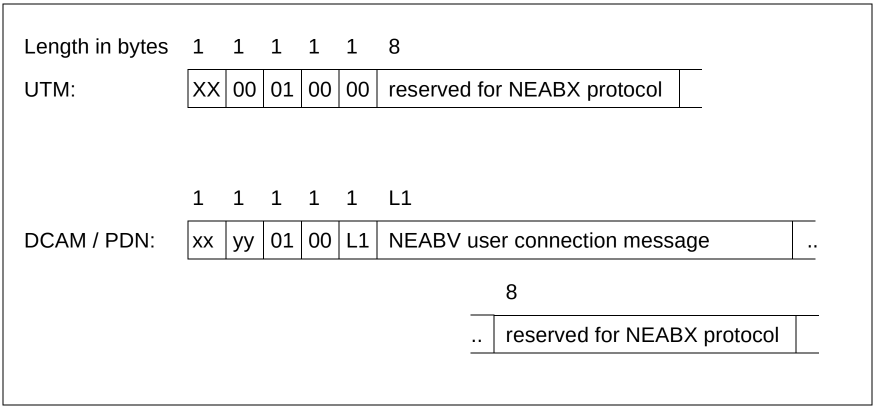
With the calls *x_conrq()* and *x_conrs()* the TS application must enter the NEABV protocol into the data buffer (*x_udatap*) before the actual user connection message. You can also generate the NEABV protocol with the aid of the NEABX service function *x_neavo()*.

With the calls *x_conin()* and *x_concf()* the NEABV protocol appears in the data buffer (*x_udatap*) as user data.

You can analyze the NEABV protocol with the aid of the NEABX service function *x_neavi()*.

The following is a description of the format of the NEABV protocol for computer interconnection via LAN and WAN.

Format of the user data in the data buffer x_udatap in the case of computer interconnection


Length in bytes
1 1 1 1 1 8
UTM:
XX 00 01 00 00 reserved for NEABX protocol
1 1 1 1 1 L1
DCAM / PDN:
xx yy 01 00 L1 NEABV user connection message ..
8
.. reserved for NEABX protocol

The meanings and values of the individual NEABV protocol elements are given below:

xx Stipulation regarding the exchange of the NEABX protocol in the data phase.

x_conrq(), *x_conrs()*, *x_concf()*: xx = X’01’
i.e. no NEABX protocol in the data phase.

x_conin(): xx = X’00’ or X’01’
i.e. the partner application conforms to what specified by the NEABX application (xx = X’00’) or proposes that there be no user services protocol in the data phase (xx = X’01’).

yy Stipulation regarding the initiative in data transfer

yy = X’01’
The sender will start data transfer.

yy = X’00’
No specification, or acceptance of the proposal of the communication partner.

L1 Length of the following user connection message.

In general: X’00’ <= L1 <= X’50’ i.e. the following user connection message may be between 0 and 80 characters in length.

With computer interconnection via WAN the user connection message must not be longer than 79 characters.

9.3.2 The NEABX service functions (NEABV service)

The user is provided with a service that creates the NEABV protocol. The purpose of this service is to prevent any mistakes during the creation of the protocol.

The NEABV protocol must be included in the following:

– the connection request with *x_conrq()*

– the connection response with *x_conrs()*

and is supplied for:

– the connection indication with *x_conin()*

– the connection confirmation with *x_concf()*

The two calls below serve to create and analyze the NEABV protocol.

x_neavo()
Generates the NEABV protocol for output. *x_neavo()* can be called prior to *x_conrq()* and *x_conrs()*.

x_neavi()
Analyzes an incoming NEABV protocol. *x_neavi()* can be called following *x_conin()* and *x_concf()* in order to analyze the NEABV protocol arriving from the partner TS application.

Use of the x_init parameter in x_neavi() and x_neavo() calls

The *x_init* parameter is used by the communication partners to work out who is to begin the data transfer in the data phase. The remarks below indicate what must be observed when specifying values for this parameter.

Possible values for *x_init* are:

X_MYINIT (X’01’):
Proposal to start data transfer.

X_INITRQ (X’00’):
: Waiting for partner’s proposal (occasionally referred to as NOINIT).*)

X_INITOK (X’00’):
: Acceptance of partner’s proposal.*)

(*) The values are coded in the same way.)

- *x_init* specifications for computer interconnection:

  The possible *x_init* specifications of the calling TS application and the expected responses of the called TS application are described here. The calling TS application passes its proposal via *x_conrq()* to the called TS application, which receives this with the *x_conin()* call. The answer *x_init* is issued by the called TS application in *x_conrs()* and received by the calling TS application in *x_concf()*.

  – *x_init* = X_MYINIT

    Proposal from the calling TS application that it should start data transfer.

    Response of the called TS application:

    x_init = X_INITOK (acceptance of partner’s proposal) or *x_disrq()*, in the case of non-acceptance.

  – *x_init* = X_INITRQ

    The calling TS application is awaiting the proposal of the called TS application.

    Expected response of the called TS application:

    x_init = X_MYINIT (the called TS application starts data transfer) or *x_disrq()*.

    If another response is passed to the calling TS application, that application should break the connection with *x_disrq()*, because no agreement has been reached.

    Remark

    If *x_init* = X_INITOK is given as the response, this can cause simultaneous communication or endless waiting. To prevent this risk, use the response *x_disrq()*.

9.4 Transport system specific features

The section “Transport system specific features” on page 103 describes the transport system specific features which also apply to the TS applications in ICMX(NEA). The features described refer to the corresponding function calls with the prefix x_ and the CMX events with the prefix X_.

9.5 Programming notes

The primary purpose of ICMX(NEA) is to make TS applications independent of the transport systems used. This allows TS applications to execute in a variety of network environments. ICMX(NEA) supports this independence for TS applications that adhere to the following rules:

1. The application should make no explicit assumptions regarding the length of a data unit or regarding the way data units are packed for communication.
2. The limits defined in <*neabx.h*> for the options must never be exceeded. Note that some transport systems do not provide certain options.
3. The TS application should handle addressing exclusively with the aid of the TNSX; it should not construct any physical transport addresses in the programs.
4. NEABX functions should not be called in signal handling routines; signal handling is not suitable for performing asynchronous processing outside the current context.
5. The program logic should be arranged in a switch/case construction, which is ideally suited for these purposes.

Example

Calling TS application

```
x_attach();
x_conrq();
for (;;) {
    switch(x_event()) {
    case X_CONCF:
        x_concf();
          :
          :
        x_datarq();
          :
          :
    case X_DATAIN:
        x_datain();
          :
    case X_DISIN:
        x_disin();
        x_detach();
          :
    case X_NOEVENT:
        continue;
    case X_ERROR:
        x_detach();
        exit();
    default:
       :
}
```

Called TS application

```
x_attach();

for (;;) {
    switch (x_event()) {
    case X_CONIN:
        x_conin();
        x_conrs();
          :
    case X_DATAIN:
        x_datain();
          :
        x_datarq();
          :
          :
    case X_DISIN:
        x_disin();
        x_detach();
          :
    case X_NOEVENT:
        continue;
    case X_ERROR:
        x_detach();
        exit();
    default:
       :
    }
```

9.6 Conventions

When using ICMX(NEA) the following conventions must be observed:

1. All identifiers starting with "_" are reserved for the system software.
2. All identifiers starting with "t_", "x_", "ts", "Ts", "cmx" or "neabx" are reserved for NEABX.
3. All preprocessor definitions starting with "T_", "X_" or "TS" are reserved for NEABX.
4. At the request of the user, signals (usually SIGIO and/or SIGTERM) are sent by the CMX components in the operating system kernel and intercepted in the NEABX library. User-defined signal routines should therefore be programmed with caution.

9.7 ICMX(NEA) - function calls

The following pages describe the NEABX calls in detail. Italic type in running text represents ordinary, replaceable formal parameters or the names of functions and files. Names in uppercase letters (e.g. X_MSG_SIZE) represent constants that have been defined in a header file (with #define).

The following conventions are used in the parameter descriptions:

->	Indicates a parameter in which NEABX expects a value provided by the caller.
<-	Indicates a parameter in which NEABX returns a value after the call.
<>	Indicates a parameter in which the caller must provide a value, which is then modified by NEABX.

Of course, if a parameter involves a pointer, this marking does not refer to the pointer itself (which is always provided by the caller), but instead to the contents of the field to which the pointer points.

In all cases, for values to be returned by NEABX appropriate storage space must be provided by the caller and a pointer must be passed to NEABX.

9.7.1 x_attach - Attach a process to NEABX (attach process)

x_attach() attaches the current process to NEABX. The parameters passed in the *x_attach()* call specify:

– the TS application for which the process is being attached,

– the types of connection setup (passive, active, etc.) possible for the process in this TS application,

– the number of connections the process may have simultaneously in this TS application.

The TS application for which the process is being attached has a GLOBAL NAME that is unique in the network and one or more T-selectors that are each unique in the local system. The T-selectors combine to form the LOCAL NAME. The LOCAL NAME must be passed to NEABX as a parameter. With the help of the ICMX(L) call *t_getloc()* and the GLOBAL NAME of the TS application, the LOCAL NAME can be queried from the TNSX and placed in a data area.

Using repeated *x_attach()* calls, the current process may attach itself to NEABX for several different TS applications.

Likewise, several different processes may attach themselves to NEABX for the same TS application, i.e. using the same LOCAL NAME. The first process to attach itself for a TS application generates the TS application. If you wish to attach the same process for the program interfaces ICMX(L) and ICMX(NEA), you must call *t_attach()* and *x_attach()* with different LOCAL NAMES.

NEABX accepts connection requests for a TS application from the network as soon as a process of the TS application has attached itself to NEABX for the acceptance of connection indications, i.e. when X_PASSIVE is specified in *x_apmode*.

If more than one process has attached itself for a TS application with X_PASSIVE, NEABX initially delivers all connection indications for the TS application to the process that first attached itself for the TS application with X_PASSIVE. Only when the maximum number of connections that this process may have for the TS application is attained are arriving connection indications delivered to one of the other processes. The order in which this is done is not defined.

```
#include <cmx.h>
#include <neabx.h>
int   x_attach (struct x_myname *name,
                struct x_opta1 *x_opt);
```

-> name

Pointer to a structure *x_myname* in which the LOCAL NAME of the TS application is to be passed. The LOCAL NAME is returned by the TNSX as a property of the GLOBAL NAME.

-> x_opt

For *x_opt*, you may specify a pointer to the structure *x_opta1* or NULL. If you specify NULL, NEABX uses the defined default values.

The structure *x_opta1* is defined in the file <*neabx.h*>.

```
   struct x_opta1 {
->    int  x_optnr;    /* Option no. */
->    int  x_apmode;   /* Process mode */
->    int  x_conlim;   /* Number of connections */
   };
```

x_optnr

Option number. Specify X_OPTA1.

x_apmode

x_apmode specifies the types of connection setup possible for the process in this TS application.

Permissible values are:

X_ACTIVE

The process is to actively set up connections.

X_PASSIVE

The process is to wait passively for connection requests.

X_REDIRECT

The process is to accept redirected connections.

These values may be combined using OR (**|**), e.g. X_ACTIVE **|** X_PASSIVE.

Default value if NULL specified:

X_ACTIVE **|** X_PASSIVE **|** X_REDIRECT

x_conlim

Maximum number of simultaneous connections that this process may have per application.

If *x_conlim* = T_NOLIMIT is specified, the process may simultaneously maintain the maximum number of connections defined when installing CMX.

Default value if NULL is specified: T_NOLIMIT

Return values

T_OK

The call was successful. The process was the first to attach itself with this LOCAL NAME.

X_NOTFIRST

The call was successful. The process has attached itself as an additional process with this LOCAL NAME.

X_ERROR

Error. Query error code using *x_error()*. The process is not attached.

Errors

If an error occurs, the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_WPARAMETER
: The specifications in *x_opt* have an invalid format or contain illegal values.

The possible error values for error type T_CMXTYPE and error class T_CMXCLASS are the same as those listed in the section “t_attach - Attach a process to CMX (attach process)” on page 110.

See also

x_detach(), t_getloc()

9.7.2 x_concf - Establish connection (connection confirmation)

x_concf() accepts from NEABX an X_CONCF event that was previously reported with *x_event()*. X_CONCF indicates that the called TS application has positively answered a connection request (*x_conrq()* call) of the current process.

x_concf() returns:

– the user data that was sent along by the called TS application. The user data must be accepted by the current process, as it contains the NEABV protocol. The value NULL is therefore illegal for the option structure. The received protocol can be analyzed by calling *x_neavi()*.

– the response of the called TS application if the current process proposed the exchange of expedited data when issuing the connection request *x_conrq()*.

If *x_concf()* returns the value T_OK, the connection is set up for the current process. As soon as a connection is established, the TS application (not CMX) has the initiative. It may:

– send normal data and (if agreed) expedited data, or

– indicate, through *t_event()*, that it is ready to receive normal data or (if agreed) expedited data, or

– redirect or close down the connection.

If NEABX returns the value X_REPEAT, the call was successful, but all user data has not yet been passed. You must call *x_concf()* once more with the same parameters as soon as *x_event()* indicates the event X_REPCCF. Only then will the connection be completely established.

```
#include <cmx.h>
#include <neabx.h>
int   x_concf (int *tref,
               struct x_optc1 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you receive when *x_event()* reports the event X_CONCF.

<> x_opt

Pointer to the *x_optc1* structure in which NEABX stores the user data. *x_opt* must always be specified, as the NEABV protocol is a part of the user data and must always be received.

The structure *x_optc1* is defined in the file <*neabx.h*>.

```
   struct x_optc1 {
->    int   x_optnr;            /* Option no. */
<-    char  *x_udatap;          /* Data buffer */
<>    int   x_udatal;           /* Length of data
                                   buffer */
<-    int   x_xdata;            /* Choice for expedited
                                   data */
<-    int   x_timeout;          /* Inactive time */
<-    char  x_passwd[X_NXPWL];  /* Connection
                                   password */
<-    int   x_prot;             /* Protocol data
                                   phase */
   };
```

x_optnr

Option number. Specify:

X_OPTC1

if space reserved for the NEABX protocol is allowed for in *x_udatal*.

X_OPTC3

if space reserved for the NEABX protocol is not allowed for in *x_udatal*.

x_udatap

Pointer to a data area. In this area NEABX enters the user connection message of the called TS application. The user connection message consists of the NEABV protocol (see section “NEABV protocol” on page 232).

The user message contained in the NEABV protocol is supplied in the code of the partner. The current transport system on the BS2000 side does not supply a connection message delivered using *x_concf()*.

x_udatal

Prior to the call the length of the data area provided must be specified here.

You must select an area large enough for the user connection message. The connection message is at most X_MSG_SIZE bytes long. If the data area is smaller than the length of the connection message received, the return code will be X_ERROR, and the connection will not be established.

In the call, NEABX enters the length of the user connection message received.

x_xdata

Returns the response of the called TS application as to whether expedited data may be used. The response is binding.

Possible values for *x_xdata*:

X_YES

The called TS application accepts the proposal for the exchange of expedited data.

X_NO

The use of expedited data is rejected by the partner.

x_timeout

This field is set to NULL.

x_passwd

Connection password. In conformity with the ISO standard, no connection password is normally delivered for incoming actions. *x_passwd* is set to NULL.

If the value X_SPECIAL was specified for the *x_prot* parameter in the *x_conrq()* call for this connection, the connection password, if one exists, is passed on by NEABX.

x_prot

Determines whether NEABX protocols are to be used in the data phase. For NEA transport systems, this is a local agreement; *x_prot* contains the value that was set in *x_conrq()*. In the case of ISO transport systems, *x_prot* contains the response (confirmation or rejection) of the partner TS application to this proposal.

Possible values for *x_prot* are:

X_NEABX

In the data phase an NEABX protocol will always be sent and expected.

X_NOBX

A NEABX protocol will not be used in the data phase.

Return values

T_OK

The call was successful. The connection is fully established. The data phase has been reached.

X_REPEAT
: The call was successful. If *x_event()* indicates the event X_REPCCF, *x_concf()* must be called again.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling x_error().

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_BADPRPI
: Unknown protocol ID byte received. The protocol element is neither a CONNECT ATTENTION nor a CONNECT protocol element. The connection can be operated as a pure ISO transport connection; however, it is no longer recognized by ICMX(NEA).

X_BADPVBYTE
: Invalid protocol version byte contained in the received NEABX protocol.

X_NOTCNPE
: A CONNECT protocol element was expected, but some other element was received.

X_MAXDAT
: More than X_MSG_SIZE bytes of user data were received at connection setup.

X_NOINFO
: TIDU length cannot be determined. The connection will be closed down again.

X_NOOPT
: No *x_opt* pointer was specified.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_concf - Establish connection (connect confirmation)” on page 121 and the following error may occur:

T_WSEQUENCE
: No *x_concf()* may be called for the connection specified in *tref*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_conrq(), x_error(), x_event(), x_neavi().

9.7.3 x_conin - Receive connection request (connection indication)

x_conin() accepts an X_CONIN event previously reported with *x_event()*. X_CONIN indicates that a calling TS application wishes to set up a connection to the current process.

The call returns:

– the TRANSPORT ADDRESS of the calling TS application,

– the LOCAL NAME of the local TS application, and

– the user data that the calling TS application included in the *x_conrq()* call. The user data contains the NEABV protocol. The NEABV protocol can be analyzed by calling *x_neavi()*.

Subsequently the connection request may be answered (confirmed) with *x_conrs()* or rejected with *x_disrq()*.

If NEABX returns the value X_REPEAT following the *x_conin()* call, the call was successful, but NEABX has not yet received all the user data. *x_conin* must then be called once more with the same parameters as soon as *x_event()* indicates the event X_REPCIN.

```
#include <cmx.h>
#include <neabx.h>
int   x_conin (int *tref,
               union x_address *toaddr,
               union x_address *fromaddr,
               struct x_optc1 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you receive when *x_event()* reports the event X_CONIN.

<- toaddr

Pointer to a union *x_address* in which NEABX enters the LOCAL NAME of the local TS application. This information is important when a process controls multiple TS applications. It indicates to which TS application the connection request is to be attributed.

<- fromaddr

Pointer to a union *x_address* in which NEABX enters the TRANSPORT ADDRESS of the calling TS application.

With the help of the call *t_getname()* (see section “t_getname - Query GLOBAL NAME (get name)” on page 178), the GLOBAL NAME of the calling TS application can be determined from the TRANSPORT ADDRESS.

<> x_opt

Pointer to the structure *x_optc1*.

With this structure you can query the information that the calling TS application included with the connection request.

x_opt must always be specified, as the NEABV protocol is a part of the user data and must always be received. The structure *x_optc1* is defined in the file <*neabx.h*>.

```
   struct x_optc1 {
->    int   x_optnr;       /* Option no. */
<-    char  *x_udatap;     /* Data buffer */
<>    int   x_udatal;      /* Length of data buffer */
<-    int   x_xdata;       /* Choice for
                              expedited data */
->    int   x_timeout;     /* Inactive time */
<-    char  x_passwd[4];   /* Connection password */
<>    int   x_prot;        /* Protocol data phase */
   };
```

x_optnr

Option number. Specify:

X_OPTC1

if space reserved for the NEABX protocol is allowed for in *x_udatal*.

X_OPTC3

if space reserved for the NEABX protocol is not allowed for in *x_udatal*.

x_udatap

Pointer to a data area. In this area NEABX enters the user connection message of the calling TS application.

The user connection message consists of the NEABV protocol (see section “NEABV protocol” on page 232).

The NEABV protocol can be analyzed using the NEABX call *x_neavi()*.

x_udatal

Prior to the call specify *x_udatap* as the length of the data area provided. You must select an area large enough for the user connection message. The connection message is at most X_MSG_SIZE bytes long.

If the data area is smaller than the length of the connection message received, the return code will be X_ERROR.

In the call, NEABX enters the length of the user connection message received.

x_xdata

In this field NEABX indicates whether the calling TS application is proposing the use of expedited data on this connection.

Possible values for *x_xdata*:

X_YES

It is proposed that expedited data be exchanged.

X_NO

It is proposed that expedited data not be exchanged.

x_timeout

This field always contains X_NO.

x_passwd

Connection password. In conformity with the ISO standard, no connection password is normally delivered for incoming actions. *x_passwd* is set to NULL.

If the value X_SPECIAL was specified for the *x_prot* parameter before the *x_conin()* call, the connection password, if one exists, is passed on by NEABX.

x_prot

Before calling *x_conin()*, you can specify the value X_SPECIAL for this parameter. X_SPECIAL causes NEABX to pass on any connection password that has been sent by the calling TS application.

Following the call, *x_prot* contains the proposal of the partner as to whether the NEABX protocol is to be exchanged in the data phase. For an NEA transport system, this represents a local agreement between the TS application and the CCP. For ISO transport systems, the agreement is between the two TS applications.

Possible values:

X_NEABX
: The NEABX protocol is to be processed in the data phase. (Always set locally for NEA transport systems.)

X_NOBX
: No NEABX protocol is to be used in the data phase.

Return values

T_OK
: The call was successful, i.e. the connection request was transferred completely.

X_REPEAT
: The call was successful. If *x_event()* indicates the event X_REPCIN, *x_conin()* must be called again.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADPRPI
: Unknown protocol ID byte received. The protocol element is neither a CONNECT ATTENTION nor a CONNECT protocol element. The connection can be operated as a pure ISO transport connection; however, it is no longer recognized by ICMX(NEA).

X_BADPVBYTE
: Invalid protocol version byte contained in the received NEABX protocol.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_BADTRANS
: The transport system on which the connection is to be established is unknown.

X_MAXDAT
: More than X_MSG_SIZE bytes of user data were received at connection setup.

X_NOINFO
: TIDU length cannot be determined. The connection will be closed down again.

X_NOOPT
: No *x_opt* pointer was specified.

X_NOTCNPE
: A CONNECT protocol element was expected, but some other element was received.

X_WPARAMETER
: The options specified in *x_opt* have an invalid format or contain illegal values.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_concf - Establish connection (connect confirmation)” on page 121 and the following error may occur.

T_WSEQUENCE
: No *x_conin()* may be called for the connection specified in *tref*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_conrq(), x_error(), x_event()

9.7.4 x_conrq - Request connection (connection request)

x_conrq() requests the establishment of a transport connection from the local TS application to a called TS application (active connection setup). When *x_conrq()* is called, the current process must pass the TRANSPORT ADDRESS of the called TS application and the LOCAL NAME of the calling TS application. The TRANSPORT ADDRESS and the LOCAL NAME are returned by the TNSX as properties of each GLOBAL NAME. They can be ascertained before the *x_conrq()* call by using the ICMX(L) calls *t_getaddr()* or *t_getloc()*.

More specifically, *x_conrq()* has the following effects:

– NEABX creates a Transport Connection Endpoint (TCEP) for the requested connection.

– The called TS application receives the event X_CONIN as a connection indication, to which it must respond.

  The answer of the called TS application is later indicated to the current process by NEABX in an *x_event()* call as event X_CONCF or X_DISIN.

– User data is sent to the called TS application along with the connection indication in the form of the NEABV protocol. The NEABV protocol can be generated with the aid of the ICMX(NEA) call *x_neavo()*.

If NEABX returns the value X_REPEAT following the *x_conrq()* call, NEABX has not passed all user data to the transport system. *x_conrq()* must then be called once more with the same parameters as soon as *x_event()* indicates the event X_REPCRQ.

```
#include <cmx.h>
#include <neabx.h>
int   x_conrq (int *tref,
               union x_address *toaddr,
               union x_address *fromaddr,
               struct x_optc1 *x_opt);
```

<- tref

Pointer to the transport reference. The transport reference is entered by NEABX at the first *x_conrq()* call and uniquely identifies the connection for NEABX. It must be specified with all calls referring to this connection. In particular, the content of *tref* must be specified in a repeated *x_conrq()* call.

-> toaddr

Pointer to a union in which the TRANSPORT ADDRESS of the called TS application is to be specified. *x_address* is defined in *<neabx.h>*.

-> fromaddr

Pointer to a union in which the LOCAL NAME of the calling TS application is to be specified. Apart from cases involving repetition, the same LOCAL NAME must be specified here as was specified in the *x_attach()* call for this TS application. *x_address* is defined in *<neabx.h>*.

-> x_opt

Pointer to the structure *x_optc1*. This structure can be used to send information to the called TS application, which receives the data when it receives the connection request.

x_opt must be specified, as the NEABV protocol must always be sent.

The structure *x_optc1* is defined in the file <*neabx.h*>.

```
   struct x_optc1 {
->    int   x_optnr;            /* Option no. */
->    char  *x_udatap;          /* Data buffer */
->    int   x_udatal;           /* Length of data
                                   buffer */
->    int   x_xdata;            /* Choice for
                                   expedited data */
->    int   x_timeout;          /* Inactive time */
->    char  x_passwd[X_NXPWL];  /* Connection
                                   password */
->    int   x_prot;             /* Protocol data
                                   phase */
   };
```

x_optnr

Option number. Specify:

X_OPTC1

if space reserved for the NEABX protocol is allowed for in the length specification in *x_udatal*. X_OPTC1 must be specified if the option number X_OPTRK was specified when the *x_neavo()* routine was called to generate the NEABV protocol.

X_OPTC3

if the space reserved for the NEABX protocol is not allowed for in the length specification in *x_udata1*. X_OPTC3 must be specified if the option number X_OPTRK1 was specified when the *x_neavo()* routine was called to generate the NEABV protocol.

x_udatap

Pointer to a storage area containing data that NEABX passes to the called TS application.

The data area contains only the user connection message (not the NEABX protocol). The user connection message consists of the NEABV protocol (see section “NEABV protocol” on page 232).

The NEABV protocol can also be created and directly passed on here by using the NEABX call *x_neavo()*.

x_udatal

Length of the data area *x_udatap* to be passed by NEABX. When option number X_OPTC1 is specified this includes the user message plus the space reserved for the NEABX protocol (8 bytes). When option number X_OPTC3 is specified it includes only the user connection message (NEABV protocol). The NEABV protocol can be generated using *x_neavo()* and the length returned by *x_neavo()* can be specified here directly.

Maximum length:

for X_OPTC1: X_MSG_SIZE
for X_OPTC3: X_MSG_SIZENEU

Minimum length:

for X_OPTC1: X_RKMSGMIN + 8 bytes

for X_OPTC3: X_RKMSGMIN

x_xdata

In *x_xdata* the current process proposes to the called TS application that the use of expedited data be permitted or ruled out.

Possible values:

X_YES

It is proposed that sending and receiving of expedited data be permitted. In the case of computer interconnection, X_YES must always be specified for communication with TIAM, DCAM, UTM and other BS2000 applications.

X_NO
: The use of expedited data is not permitted.

x_timeout

X_NO
: No time monitoring.

n
: The connection may be inactive for n seconds. Thereafter NEABX will close down the connection. n must be specified as a decimal number.

x_passwd
: Connection password. For partner applications that require a password specify four bytes of binary information. If you do not wish to include a connection password, specify NULL for *x_passwd*. To comply with the ISO standard, NULL should be specified for this field.

x_prot
: In *x_prot* the current process proposes whether or not the NEABX protocol is to be used in the data phase. For NEA transport systems, this is a local agreement, i.e. the proposal is always confirmed in the *x_concf()* call. In the case of ISO transport systems, the proposal is delivered to the partner TS application, which may either confirm or reject it. The response is delivered to the current process with *x_concf()*.

Possible values:

X_NEABX or NULL
: The NEABX protocol is to be processed in the data phase.

X_NOBX
: The NEABX protocol is not to be used in the data phase.

The following value may be specified in addition to X_NEABX or X_NOBX by combining it with a OR (|) operator.

X_SPECIAL
: ICMX(NEA) handles the following points in a special way:

– The connection password of the partner TS application, if one arrives, is passed on to the current process in the *x_concf()* call of this connection.

– Transport acknowledgment requests are not handled by NEABX in the data phase of this connection, but are passed on to the current process, which must then send, and can also request, transport acknowledgments itself.

Return values

T_OK
: The call was successful. The connection request was completely transferred to the transport system.

X_REPEAT
: The call was successful. If *x_event()* indicates the event X_REPCRQ, *x_conrq()* must be called again with the same parameters.

X_DATASTOP
: The call was successful. All user data has been sent. In a subsequent data phase, processing must first wait for the event X_DATAGO.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*

X_BADPROT
: *x_prot* does not contain one of the values X_NEABX, X_NOBX or NULL.

X_BADTABLE
: The *tref* specified in the repeated *x_conrq()* is unknown to NEABX. It was not found in the corresponding table.

X_BADTRANS
: The transport system on which the connection is to be established is unknown.

X_NOOPT
: No *x_opt* pointer was specified.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_concf - Establish connection (connect confirmation)” on page 121 may occur.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_attach(), x_error(), x_concf(), x_event(), t_getaddr(), t_getloc

9.7.5 x_conrs - Respond to connection request (connection response)

x_conrs() is used by the called TS application to accept (confirm) the connection request of a calling TS application, the connection request having been previously indicated to the current process in *x_event()*, with the event X_CONIN. The current process must accept the X_CONIN event with *x_conin()* (passive connection setup) before calling *x_conrs()*. This connection response (i.e. confirmation) is delivered to the calling TS application with the event X_CONCF.

With the response *x_conrs()*

– user data must be passed to NEABX. The user data is passed in the form of the NEABV protocol. The NEABV protocol can be generated with the help of the ICMX(NEA) call *x_neavo()*.

– the connection is completely set up for the current process.

The successful completion of a *x_conrs()* call indicates that the connection has been set up. The initiative is now with the TS application. It can:

– send normal data as well as expedited data (if agreed), or

– indicate with *x_event()* that it is ready to receive normal data or expedited data (if agreed).

– close down or redirect the connection.

```
#include <cmx.h>
#include <neabx.h>
int   x_conrs (int *tref,
               struct x_optc1 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you receive when the *x_event()* call reports the event X_CONIN.

-> x_opt

Pointer to the structure *x_optc1*. *x_opt* must always be specified, as the NEABV protocol must be passed.

The structure *x_optc1* is defined in the file <*neabx.h*>.

```
   struct x_optc1 {
->    int   x_optnr;            /* Option no. */
->    char  *x_udatap;          /* Data buffer */
->    int   x_udatal;           /* Length of data
                                   buffer */
->    int   x_xdata;            /* Choice for expedited
                                   data */
->    int   x_timeout;          /* Inactive time */
->    char  x_passwd[X_NXPWL];  /* Connection password */
->    int   x_prot;             /* Protocol data phase */
   };
```

x_optnr

Option number. Specify:

X_OPTC1

if space reserved for the NEABX protocol is allowed for in *x_udatal*. X_OPTC1 must be specified if the option number X_OPTRK was specified when the *x_neavo()* routine was called to generate the NEABV protocol.

X_OPTC3

the space reserved for the NEABX protocol is not allowed for in the length specification in *x_udata1*.

X_OPTC3 must be specified if the option number X_OPTRK1 was specified when the *x_neavo()* routine was called to generate the NEABV protocol.

x_udatap

Pointer to a storage area containing data that NEABX passes to the calling TS application. The data area contains only the user connection message (not the NEABX protocol). The user connection message consists of the NEABV protocol (see section “NEABV protocol” on page 232). The NEABV protocol can also be created and passed on directly by using the NEABX call *x_neavo()*.

x_udatal

Length of the data area *x_udatap* to be passed by NEABX. When option number X_OPTC1 is specified this includes the user connection message plus the space reserved for the NEABX protocol (8 bytes). When option number X_OPTC3 is specified it includes only the user connection message (NEABV protocol). If the NEABV protocol is generated using *x_neavo()*, the length returned by *x_neavo()* can be specified here directly.

Maximum length:
for X_OPTC1: X_MSG_SIZE
for X_OPTC3: X_MSG_SIZENEU

Minimum length:
for X_OPTC1: X_RKMSGMIN + 8 bytes

for X_OPTC3: X_RKMSGMIN

x_xdata
In *x_xdata* the current process responds to the proposal of the calling TS application regarding the use of expedited data. The response is binding. If the proposal of calling TS application was X_NO, the response must be X_NO.

Possible values:

X_YES
Proposal to send and receive expedited data is accepted.

X_YES must generally be specified for communication with TIAM, DCAM, and UTM applications in BS2000/OSD.

X_NO
Use of expedited data is rejected.

x_timeout
The contents of this field are irrelevant.

x_passwd
Connection password. You may specify four bytes of binary information. If you do not wish to include a connection password, specify NULL for *x_passwd*.

x_prot
Response to the proposal as to whether or not the NEABX protocol is to be used in the data phase. In the case of NEA transport systems, this is always a local agreement; for ISO transport systems, the exchange of the NEABX protocol is negotiated with the partner TS application.

Possible values:

X_NEABX or NULL
: The proposal to process the NEABX protocol in the data phase is accepted. X_NEABX must always be specified for connections to DCAM, TIAM, and UTM applications.

X_NOBX
: NEABX protocols will not be used in the data phase.

  The following value may be specified in addition to X_NEABX or X_NOBX by combining it with an OR (|) operator.

X_SPECIAL
: Transport acknowledgment requests are not handled by NEABX in the data phase of this connection, but are passed on to the current process, which must then send, and can also request, transport acknowledgments itself.

Return values

T_OK
: The call was successful. The connection is fully established.

X_DATASTOP
: The call was successful. All user data was sent. In a subsequent data phase, processing must first wait for the event X_DATAGO.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADPROT
: *x_prot* does not contain one of the values X_NEABX, X_NOBX or NULL.

X_BADTABLE

The specified *tref* is unknown to the migration service. It was not found in the relevant table.

X_NOINFO

TIDU length cannot be determined. The connection was closed down again.

X_NOOPT

No *x_opt* pointer was specified.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_concf - Establish connection (connect confirmation)” on page 121 and the following error may occur:

T_WSEQUENCE

No *x_conrs()* may be called for the connection specified in *tref*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_conin(), x_error(), x_event()

9.7.6 x_datago - Release the flow of data (datago)

x_datago() releases the blocked flow of data on the specified connection. The current process informs NEABX that it is again ready to receive data. This call also releases the flow of expedited data (if it is being used) if it (also) had been blocked.

More specifically, the call has the following effects:

– The current process can again receive the events X_DATAIN and X_XDATIN for the specified connection, if they are waiting.

– The sending TS application receives the event X_DATAGO. It may again send data.

```
#include <cmx.h>
#include <neabx.h>
int   x_datago (int *tref);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference of the connection for which you wish to release the flow of data.

Return values

T_OK

The call was successful. The blocked data flow has been released.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADTABLE

The specified *tref* is not contained in the table of connections known to NEABX. It is either not assigned to a connection, or the associated connection was not set up via ICMX(NEA).

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_concf - Establish connection (connect confirmation)” on page 121 and the following error may occur:

T_WSEQUENCE
: The connection specified in *tref* is not yet fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_datastop(), x_xdatstop(), x_error(), x_event()

9.7.7 x_datain - Receive data (data indication)

x_datain() is called by the current process to fetch a X_DATAIN event previously reported with *x_event()*. By means of this call the current process receives on the specified connection a data unit (TIDU) belonging to the message that is currently being transmitted (TSDU) by the sending TS application.

x_chain indicates whether or not an additional TIDU belonging to the TSDU exists. Each additional TIDU is indicated by NEABX with a new X_DATAIN event.

The length of a TIDU depends on the transport system used. The TIDU length for a connection that has already been set up can be queried by calling *x_info()*. A TIDU need not be completely full. The breakdown of a TSDU into TIDUs is purely local and does not indicate anything regarding the breakdown of the TSDU into TIDUs at the sending TS application.

A TIDU received using *x_datain()* may be longer or shorter than the TIDU sent with *x_datarq()*. If it is shorter, X_MORE appears in the *x_chain* indicator, and *x_event()* indicates with the event X_DATAIN that further data is ready to be received.

If you are not ready to receive data, you can stop the flow of data with *x_datastop()*. You thereby prevent NEABX from delivering the event X_DATAIN to the local TS application. A data unit that has already been indicated with X_DATAIN, however, must always be completely fetched.

If the handling of transport acknowledgments in the TS application was negotiated at connection setup, the return value X_ERROR with the error code X_QUITPE (queried with the *x_error()* call) indicates that a transport acknowledgment has arrived for the current process. The option members *x_quit* and *x_seqno* are then supplied accordingly.

```
#include <cmx.h>
#include <neabx.h>
int   x_datain (int *tref,
                char *x_datap,
                int *x_datal,
                int *x_chain,
                x_optd *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you receive when *x_event()* reports the event X_DATAIN.

<- x_datap

Pointer to a storage area in which NEABX enters the data received.

If *x_opt* is equal to NULL, NEABX passes all received data to the local TS application.

If *x_opt* is not equal to NULL, prior to passing the data to the local TS application NEABX deals with the NEABX protocol in accordance with the specified option number:

X_OPTD1:

The storage area must contain space reserved for the NEABX protocol (mode compatible with CMX V2.1).

X_OPTD2:

The NEABX protocol is invisible to the TS application.

X_OPTD3:

The NEABX protocol is placed at the beginning of the data area and the length of the protocol is recorded in the *x_offset* member of the *x_optd3* structure, thus informing the TS application as to where the net message begins.

<> x_datal

Prior to the call specify the length of the data area *x_datap*. This must be at least the length of one data unit, whose size you must ascertain for each transport connection by means of *x_info()*. In the call NEABX enters the number of bytes entered that are passed to the local TS application. The length returned always refers to just the net data length, even when using option X_OPTD3.

<- x_chain

Pointer to an indicator used by NEABX to show whether there are additional TIDUs belonging to the TSDU.

The following values are possible:

X_MORE

At least one TIDU belonging to the TSDU follows. For each additional TIDU, NEABX reports a separate X_DATAIN event.

X_END

No further TIDU exists. The TSDU has been completely transferred.

<> x_opt

Pointer to a union *x_optd* containing one of the structures *x_optd1*, *x_optd3* or the specification NULL. The *x_opt* specification is mandatory if the use of the NEABX protocol in the data phase was agreed for the connection and the first TIDU of a TSDU is being received. NULL must be specified if an additional TIDU of a TSDU is to be received, i.e. the preceding TIDU on this connection was received with **x_chain* = X_MORE, at connection setup it was agreed that NEABX protocols would not be used in the data phase.

The structures *x_optd1* and *x_optd3* and the union *x_optd* are defined in the file <*neabx.h*>.

```
   struct x_optd1 {
->    int   x_optnr;    /* Option number,
                           X_OPTD1, X_OPTD2 */
<-    int   x_code;     /* Message code */
<-    int   x_strukt;   /* Message structure */
<-    int   x_quit;     /* Transport acknowledgments */
<-    short x_seqno;    /* Message sequence numbers */
   };
   struct x_optd3 {
->    int   x_optnr;    /* Option number, X_OPTD3 */
<-    int   x_code;     /* Message code */
<-    int   x_strukt;   /* Message structure */
<-    int   x_quit;     /* Transport acknowledgments */
<-    short x_seqno;    /* Message sequence numbers */
<-    int   x_offset;   /* Offset to the start of data */
   };
```

x_optnr

Option number. Possible values:

X_OPTD1 or X_OPTD2 for *x_optd1*
X_OPTD3 for *x_optd3*

The meanings of the values are described under *x_datap*.

x_code

Designates the message code. The meanings are given below:

X_ASCII

The arriving data is coded in ASCII.

X_EBCDIC

The arriving data is coded in EBCDIC.

X_TRANS (= X_EBCDIC)
: The arriving data is transparent.

X_UNDEF
: NEABX has no information about the code. The data is encoded as sent by the partner.

  With ISO-CCP and NEA-CCP connections a user services protocol is present in the code in which it was sent by the partner. User services protocols are thus transparent.

x_strukt
: Message structure. The following values are possible:

  X_ETB
  : A further group element of the subgroup follows.

  X_ETX
  : Last or only group element of a subgroup; further subgroup follows.

  X_ETBEOT
  : Last group element of a group.

  X_ETXEOT
  : Last or only subgroup of a group.

x_quit
: Is only relevant if the handling of transport acknowledgments in the TS application was specified at connection setup.

  If a DATA protocol element was received, the following values are possible for *x_quit*:

0	No acknowledgment requested.
1	A transport acknowledgment is requested.

  If an acknowledgment protocol element was received (return value X_ERROR with error code X_QUITPE), the possible values for *x_quit* are:

1	Positive acknowledgment received.
2	Negative acknowledgment received.

x_seqno
: Contains the message sequence number, provided x_quit is not NULL.

x_offset

In this field NEABX returns the length of the NEABX protocol. The TS application is thus informed as to where the net message begins. *x_offset* specifies the offset from *x_datap* at which the net data begins.

Return values

T_OK

The data unit has been completely read.

X_DATASTOP

The data has been fully accepted by the transport system, but a block on the sending of data was indicated when attempting to send a transport acknowledgment needed in the NEABX protocol. It is necessary to wait for the event X_DATAGO.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN

Invalid data buffer length in *x_udatal*

X_BADTABLE

The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_NOTDTPE

DATA protocol element expected but not received.

X_QUITPE

x_datain() received an acknowledgment protocol element. The option members *x_quit* and *x_seqno* are supplied accordingly.

X_BADDTPELI

The length of the DATA protocol element specified in the received NEABX protocol is invalid.

X_WPARAMETER
: Invalid parameter; an invalid value was specified in *x_optnr*.

X_WXOPT
: Invalid *x_opt* specification:

  x_opt != NULL, although no NEABX protocol;

  x_opt != NULL, although second and subsequent TIDU being received

  x_opt = NULL, although NEABX protocol agreed, and first

  TIDU of the TSDU being received.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_datain - Receive data (data indication)” on page 138 and in the section “t_vdatain - Receive data (data indication)” on page 198 and the following error value may occur:

T_WSEQUENCE
: The connection specified in *tref* has not yet been fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_error(), x_event(), x_info()

9.7.8 x_datarq - Send data (data request)

x_datarq() is used to send the next Transport Interface Data Unit (TIDU) of a Transport Service Data Unit (TSDU) to the receiving TS application. With *tref* you specify the connection on which you wish to send the data. *x_info()* returns the maximum length of a data unit that may be sent on this connection. The maximum length depends on the transport system used.

If the message that you wish to send is longer than one data unit, you will have to call *x_datarq()* several times in succession. By means of the indicator *x_chain* you inform NEABX whether or not additional data units belonging to the message follow.

If *x_datarq()* returns X_DATASTOP, the data unit has been accepted but the flow of data for the connection is blocked. This may occur on the initiative of the receiving TS application, by means of *x_datastop()*, or it may be brought about by NEABX, if the local buffer is in danger of overflowing. In such cases you must wait, with *x_event()*, for the event X_DATAGO before sending more data on the connection.

```
#include <cmx.h>
#include <neabx.h>
int   x_datarq (int *tref,
                char *x_datap,
                int *x_datal,
                int *x_chain,
                x_optd *x_opt);
```

-> tref

Pointer to the transport reference. Here you specify the transport reference of the connection on which you wish to send data.

-> x_datap

Pointer to a storage area containing the data that you wish to send. If *x_opt* is not equal to NULL, the quantity of data sent corresponds to that specified below for the option number used. In any case, the values specified by the TS application need only refer to the net data. The TS application knows nothing about any prefixed protocols.

-> x_datal

Pointer to the length specification **x_data1*. The following should be noted for the length specification **x_datal*:

- If at connection setup it was agreed that no NEABX protocols would be exchanged during the data phase (*x_opt* = NULL), **x_datal* corresponds exactly to the length of the data to be sent in *x_datap*. The following applies:

  Maximum value in **x_datal* = *x_maxl* - X_DRQPHL

  Minimum value in **x_datal* = 1 byte (send 1 byte of data)

  x_maxl is the TIDU length (maximum length of a data unit). It can be obtained via *x_info()*.

- If at connection setup it was agreed that the exchange of NEABX protocols during the data phase was to take place, the data length to be specified depends on the option number *x_optnr* specified in *x_opt*. The following applies if

  x_optnr = X_OPTD1

  **x_datal* must be specified to be X_DRQPHL bytes larger than the size needed for the net data to be sent. However, the storage space of the application is not utilized in forming the protocol (mode compatible with CMX V2.1).

  Maximum value for **x_datal* = x_maxl.

  Minimum value for **x_datal* = X_DRQPHL.

  x_optnr = X_OPTD2

  **x_datal* contains only the net data length. The storage space of the TS application need not include any space reserved for the NEABX protocol.

  Maximum value for **x_datal* = *x_maxl* - X_DRQPHL.

  Minimum value for **x_datal* = 0 bytes.

  x_optnr = X_OPTD3

  **x_datal* contains only the net data length. In the *x_offset* member of the option structure the TS application records the offset from *x_datap* at which the net data begins. This offset must be equal to X_DRQPHL.

  Maximum value for **x_datal* = *x_maxl* - X_DRQPHL.

  Minimum value for **x_datal* = 0 bytes.

x_opt = NULL

**x_datal* contains only the net data length. The storage space of the TS application need not include any space reserved for the NEABX protocol.

Maximum value for **x_datal* = *x_maxl* - X_DRQPHL.

Minimum value for **x_datal* = 1 byte

x_maxl is the TIDU length (maximum length of a data unit). It can be obtained via *x_info()*.

-> x_chain

Pointer to an indicator used to indicate to NEABX whether or not there are additional data units belonging to the message.

The following values are possible:

X_MORE

Additional data units of the message follow. *x_datarq()* must be called again for each data unit.

X_END

There are no further data units present. The message has been completely transferred.

-> x_opt

Pointer to a union x_optd containing one of the structures x_optd1 or x_optd3, or the specification NULL.

The x_opt specification is mandatory if the use of the NEABX protocol in the data phase was agreed for the connection and the first TIDU of a TSDU is being sent. NULL must be specified and is only permitted if

a) an additional data unit of a message is to be sent, i.e. the previous data unit was sent with **x_chain* = X_MORE;

b) at connection setup it was agreed that NEABX protocols would not be used in the data phase.

The structures *x_optd1* and *x_optd3* and the union *x_optd* are defined in the file <*neabx.h*>.

```
   struct x_optd1 {
->    int   x_optnr;    /* Option number,
                           X_OPTD1, X_OPTD2 */
->    int   x_code;     /* Message code */
->    int   x_strukt;   /* Message structure */
->    int   x_quit;     /* Transport acknowledgments */
->    short x_seqno;    /* Message sequence numbers */
   };
   struct x_optd3 {
->    int   x_optnr;    /* Option number, X_OPTD3 */
->    int   x_code;     /* Message code */
->    int   x_strukt;   /* Message structure */
->    int   x_quit;     /* Transport acknowledgments */
->    short x_seqno;    /* Message sequence numbers */
->    int   x_offset;   /* Offset to the start of
                           data */
   };
```

x_optnr

Option number. Possible values:

X_OPTD1 or X_OPTD2 for *x_optd1*
X_OPTD3 for *x_optd3*

The meanings of the values are described under *x_datal*.

x_code

Designates the message code for the data in *x_datap*:

X_ASCII

The data to be sent is coded in ASCII.

X_EBCDIC

The data to be sent is coded in EBCDIC.

X_TRANS

The data to be sent is transparent.

The data must be in the code expected by the partner.

With ISO and NEA connections any user services protocol included must be in the code that the partner expects. User services protocols are thus transparent.

x_strukt

Message structure.

Specify X_ETXEOT.

x_quit

Is only relevant if the handling of transport acknowledgments in the TS application was specified at connection setup.

If data is to be sent (*x_datal* != 0), the acknowledgment request bit QVBIT is set in the NEABX protocol.

If no user data is to be sent (*x_datal* = 0 or less), the following values are possible for *x_quit*:

0	Error
1	Positive transport acknowledgment sent.
2	Negative transport acknowledgment sent.

x_seqno

Contains the message sequence number, provided x_quit is not NULL.

x_offset

In the *x_offset* member the TS application records the offset from *x_datap* at which the net data begins. This offset must be equal to X_DRQPHL.

Return values

T_OK

The call was successful.

X_DATASTOP

The call was successful, but you may not send further data until the event X_DATAGO arrives.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_BADXCODE
: The value in *x_code* is invalid.

X_WPARAMETER
: Invalid parameter; an incorrect value was specified for *x_optnr*.

X_WXOPT
: Invalid *x_opt* specification:

  x_opt != NULL, although no NEABX protocol;

  x_opt != NULL, although second and subsequent TIDU being sent

  x_opt = NULL, although NEABX protocol agreed, and first

  TIDU of the TSDU being sent.

X_BADSTRUKT
: No legal value was specified in *x_strukt*.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_datarq - Send data (data request)” on page 141 and in the section “t_vdatarq - Send data (data request)” on page 201 and the following error may occur:

T_WSEQUENCE
: The connection specified in *tref* has not yet been fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_datastop(), x_error(), x_event(), x_info(), x_xdatstop()

9.7.9 x_datastop - Stop the flow of data (data stop)

x_datastop() blocks the flow of data on the specified connection.

In particular, the effects of *x_datastop()* are:

– The current process tells CMX that, until further notice, it is not ready to receive data for this connection. However, a X_DATAIN event that has already been indicated must first be accepted with *x_datain()*.

– The current process no longer receives the event X_DATAIN for the specified connection. However, while the data flow is blocked it may call other CMX functions, e.g. to set up, close down or redirect another connection. It may also send data on the specified connection itself, provided no block on sending data was set for it (X_DATASTOP).

– The sending TS application receives (during this period) the return value T_DATASTOP when it calls *x_datarq()*. It may not send any more data (see also section “Transport system specific features” on page 103.)

The flow of data is released with *x_datago()*.

Expedited data is not affected by *x_datastop()*.

```
#include <cmx.h>
#include <neabx.h>
int   x_datastop (int *tref);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference of the connection for which you wish to stop the flow of data.

Return values

T_OK

The call was successful.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_datastop - Stop the flow of data (data stop)” on page 144 and the following error may occur:

T_WSEQUENCE
: The connection specified in *tref* has not yet been fully established.

In addition, the errors listed under *ioctl(2)* may occur.

9.7.10 x_detach - Detach from NEABX (detach process)

x_detach() detaches the current process for the specified TS application from NEABX. If connections still exist for this process, they are closed down implicitly by NEABX. Normally though, all connections should be closed down with *x_disrq()* before calling *x_detach()*. When the last process of a TS application has been detached, the TS application is unknown to NEABX. Connection requests for that TS application will then no longer be accepted.

```
#include <cmx.h>
#include <neabx.h>
int   x_detach (struct x_myname *name,
                struct x_opta1 *x_opt);
```

-> name

Pointer to the structure *t_myname* in which the LOCAL NAME of the TS application is to be specified. The specified LOCAL NAME must be the same as the one given when *x_attach()* was called.

-> opt

Must be set to NULL.

Return values

T_OK

The call was successful.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_detach - Detach a process from a TS application (detach process)” on page 146 may occur.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_attach(), x_error()

9.7.11 x_disin - Accept disconnection (disconnection indication)

You call *x_disin()* when you have received the event X_DISIN. If you do not call *x_disin()*, NEABX still closes down the connection. With *x_disin()* you learn whether the connection was closed down by NEABX or by the remote TS application.

In addition, *x_disin()* returns:

– the user data sent by the remote TS application, if the disconnection was initiated by the remote TS application and if the transport system used provides this option;

– the reason for closing the transport connection, if the X_DISIN event was initiated by NEABX or by the transport system.

  The reason for the disconnection is returned by *x_disin()* in hexadecimal form. The plain English form of the code can be obtained with the aid of the ICMX(L) function *t_preason()* or *t_strreason()*.

```
#include <cmx.h>
#include <neabx.h>
int   x_disin (int *tref,
               int *reason,
               struct x_optc2 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you obtain if *x_event()* reports the event X_DISIN.

<- reason

Pointer to a field containing the reason for the disconnection. The value returned is either T_USER (the communication partner closed down the connection) or the disconnection reason of CMX or the CCPs, if CMX closed down the connection. The values returned by CMX or the CCPs are described in the appendix.

<> x_opt

Pointer to the structure *x_optc2*. With this structure you can check the information that the remote TS application sent when it closed down the connection. If instead of a pointer you specify NULL, NEABX discards the information sent.

At present, no information can be sent by any of the possible partner TS applications, since the partner transport systems are not yet provided with an interface to transmit user data.

The structure *x_optc2* is defined in the file <*neabx.h*>.

```
   struct x_optc2 {
->    int   x_optnr;      /* Option no. */
<-    char  *x_udatap;    /* Data buffer */
<>    int   x_udatal;     /* Length of the data buffer */
   };
```

x_optnr
: Option number. Specify X_OPTC2.

x_udatap
: Pointer to a data area. In this area NEABX enters the user data that the remote TS application sent when it closed down the connection.

x_udatal
: Prior to the call, specify the length of the allocated data area *x_udatap*. The area must be large enough to accommodate the received user data. The maximum permissible user data length depends on the transport system used. X_MSG_SIZE is a suitable maximum size for all transport systems. After the call, the length of the received user data will be contained in *x_udatal*.

Return values

T_OK
: The call was successful.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in section “t_disin - Accept disconnection (disconnection indication)” on page 148 may occur. They can be queried by calling *x_error()*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_detach(), x_disrq(), x_event()

9.7.12 x_disrq - Close down connection (disconnection request)

With *x_disrq()* you can:

– close down an existing connection, or

– reject the connection request of a remote TS application.

In both cases a disconnect indication (X_DISIN) and the reason for disconnection T_USER are delivered to the remote TS application. Either TS application may close down the connection, regardless of which one actively set it up. If the *x_disrq()* call is successful, the connection is closed down. NEABX can also close down connections, if NEABX-internal reasons demand this.

The *x_disrq()* call may overtake data units that were sent earlier but are still in transit. These data units are then lost. To prevent this, you may e.g. stipulate logical acknowledgments and call *x_disrq()* only when you have received a positive acknowledgment for the last sent TIDU.

No user data can be passed to the remote TS application when closing down the connection, since DCAM does not offer an interface at which user data can be passed to the application.

```
#include <cmx.h>
#include <neabx.h>
int   x_disrq (int *tref,
               struct x_optc2 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference of the connection that you wish to close down. In case you wish to reject a connection request indicated by *x_event()* with the event X_CONIN, *x_event()* also returns the transport reference of the connection concerned.

-> x_opt

For *x_opt*, specify the NULL pointer.

User data cannot be passed for TS applications in BS2000/OSD via BCAM.

Return values

T_OK
: The call was successful.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_WXOPT
: Invalid specification for *x_opt*: NULL **must** be specified for *x_opt*.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_disrq - Close down connection (disconnection request)” on page 151 may occur.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_conin(), x_disin(), x_event(), x_error()

9.7.13 x_error - Query error codes (error)

x_error() returns diagnostic information when a NEABX call returns X_ERROR.

The possible messages for the calls to ICMX(NEA) are generated either in the NEABX library in the user process or in the operating system kernel. The messages from the operating system kernel can be further differentiated according to whether they are generated in NEABX or CMX itself or result from operating system calls.

The error messages generated by NEABX are returned by *x_error()* in hexadecimal form. Error codes of error type X_BX3 and error class X_NEAERR can be converted to plain English with the aid of the calls *x_strerror()* and *x_perror()*. *x_strerror()* returns a pointer to a static area that contains the plain English form of the error message.

x_perror() writes the plain English form of an error message to *stderr*.

The hexadecimal error code can also be decoded by using the *cmxdec* command (see the “CMX, Operation and Administration” manual [1] or [2]).

The format of error messages is described in the section “Error handling” on page 40.

```
#include <cmx.h>
#include <neabx.h>
int   x_error (void);
```

Return values

The value returned by *x_error()* is the hexadecimal code for the error message generated by NEABX (error type, error class, error value). The error messages are defined in *<neabx.h>*. A list of all possible error values of error type X_BX3 (9) and error class X_NEAERR (B), i.e. all possible return values for *x_error()*, can be found in the appendix.

In the descriptions of the individual ICMX(NEA) function calls, the error values that *x_error()* returns if a particular function terminates in error are listed under the heading "Errors".

See also

x_perror(), x_strerror()

9.7.14 x_event - Await or query event (event)

x_event() determines whether an NEABX event has arrived for the current process.

The parameter *x_cmode* specifies the processing mode of *x_event()*.

x_event() can:

– **synchronously** wait for an NEABX event for the current process to arrive. While waiting, the process is suspended.

  Waiting can be interrupted using signals.

  A time limit for synchronous waiting may be specified in the *x_opt* options. If no event arrives within this waiting period, waiting is terminated.

– **asynchronously** check whether an NEABX event for the current process has arrived. The function always returns immediately to the current process.

Along with the appropriate event, *x_event()* returns:

– the transport reference of the connection involved, to permit the event to be associated with the appropriate connection (*tref* parameter),

– event-specific additional information, if this has been specified in the *x_opt* options.

If several events are present for a connection, they are indicated one after another in the order in which they arrived.

Exceptions:

– An X_XDATIN event (expedited data received) may overtake X_DATAIN events (normal data received) without destroying them.

– An X_DISIN event (disconnection indication) may overtake X_DATAIN and X_XDATIN events for the connection involved and thus destroy them. The data that X_DATAIN/X_XDATIN was to have indicated is lost.

i *x_event()* permits a TS application to maintain NEA-compliant as well as ISO-compliant connections within a process, i.e. the process uses both ICMX(NEA) and ICMX(L). A process of this type must independently decide whether a transport reference belongs to an NEA connection or to an ISO connection. *x_event()* also reports events for transport references that are not known to NEABX.

```
#include <cmx.h>
#include <neabx.h>
int   x_event (int *tref,
               int x_cmode,
               struct x_opte1 *x_opt);
```

<- tref

Pointer to the transport reference. Here NEABX returns the transport reference of the connection to which the reported event belongs. For the events X_NOEVENT and X_ERROR the contents of *tref* are undefined.

-> x_cmode

Specifies whether *x_event()* is to wait for an event synchronously or whether it is to asynchronously check if an event has occurred.

Possible values:

X_WAIT (synchronous processing)

The current process is suspended until a TS event arrives, the defined time limit expires (*x_timeout* parameter in *x_opt*) or until it is awakened by a signal.

The event X_NOEVENT is indicated in the latter two cases.

All signals except SIGTERM may be used to wake (*alarm()*) the process.

X_CHECK (asynchronous processing)

x_event() checks whether an event is waiting. If there is no such event, *x_event()* returns with X_NOEVENT.

<> x_opt

For *opt*, specify the value NULL or a pointer to the structure *x_opte1* with user options. The structure *x_opte1* is defined in the file <*neabx.h*>.

```
   struct x_opte1 {
->    int x_optnr;     /* Option number */
->    int x_timeout;   /* Time limit for X_WAIT */
<-    int x_evdat;     /* Event-specific additional
                          information */
   };
```

x_optnr

Specify X_OPTE1.

x_timeout

For *x_timeout* a waiting period may be specified (in seconds). With *x_cmode* = X_WAIT, *x_event()* halts the synchronous waiting when the waiting period elapses.

The specification of a value less than zero (-1) means that no timer is activated. With *x_cmode* = X_CHECK, the value specified for *x_timeout* is ignored.

x_evdat

With the events X_DATAIN and X_XDATIN the length of the indicated data is returned in *x_evdat*. This length can then be specified for the functions *x_datain()* and *x_xdatin()*.

Return values

X_NOEVENT

If x_cmode = X_CHECK: No event waiting.

If x_cmode = X_WAIT: Abort, e.g. by a signal or T_DATAGO was internally indicated by CMX, but T_DATASTOP was initiated again when sending a still pending transport acknowledgment. Consequently, T_DATAGO is undone, and no actual event is to be reported.

The contents of *tref* are undefined.

X_DATAIN

Data has been received on the connection specified in *tref*.

Response expected by NEABX: *x_datain()* call.

NEABX does not indicate this event so long as the data flow is blocked, i.e. when the receiving process has issued *x_datastop()* for the connection.

X_DATAGO

The local TS application can again send data via the connection specified in *tref*.

Possible reaction: *x_datarq()*.

The event X_DATGO also permits the local TS application to again send expedited data via this connection, provided the use of expedited data was agreed at connection setup.

X_XDATIN

Expedited data has been received on the connection specified in *tref*.

Response expected by NEABX: *x_xdatin()*.

NEABX indicates this event only if the use of expedited data was agreed at connection setup.

As long as the flow of expedited data is stopped, i.e. the receiving process has issued *x_datastop()* for the connection, this event is not indicated.

X_XDATGO

The local TS application may again send expedited data via the connection specified in *tref*.

Possible reaction: *x_datarq()*.

NEABX indicates this event only if the use of expedited data was agreed at connection setup. Normal data may still not be sent.

X_CONIN

A partner application wishes to set up a connection to the local TS application (incoming call). This connection request must be received with *x_conin()* and then confirmed with *x_conrs()* or rejected with *x_disrq()*.

Reaction expected by NEABX: *x_conin()*, then *x_conrs()* or *x_disrq()*.

X_CONCF

The remote TS application has accepted the connection request with *x_conrs()*. You must accept this answer with *x_concf()*. The connection is then established.

Reaction expected by NEABX: *x_concf()*.

X_DISIN

Either the called TS application has rejected a connection request or the remote TS application or NEABX has closed down an existing connection. You must accept this indication with *x_disin()*.

Reaction expected by NEABX: *x_disin()*.

X_REDIN

Another process of the TS application would like to redirect an already established connection to this process. You must accept the connection with *x_redin()*.

Reaction expected by NEABX: *x_redin()*.

X_REPCCF
: NEABX has not yet been able to completely set up the connection. The user data passed by the called TS application with *x_conrs* must still be accepted. Reaction expected by NEABX: repetition of the *x_concf()* call.

X_REPCIN
: The user data passed by the calling TS application with the connection request must be accepted. Reaction expected by NEABX: repetition of the *x_conin()* call.

X_REPCRQ
: NEABX has requested the connection, but the request has not been completely received by the remote TS application. Therefore the *x_conrq()* call must be repeated.

  Reaction expected by NEABX: repetition of the *x_conrq()* call.

X_ERROR
: Error. Query error code using *x_error()*.

  The contents of *tref* may have changed as compared to its contents at the time of the call; however, it will always be undefined.

Errors

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_event - Await or query event (event)” on page 154 may occur. They can be queried by calling *x_error()*.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_attach(), x_concf(), x_conin(), x_datain(), x_datago(), x_datastop(), x_disin(), x_error(), x_redin(), x_xdatin(), x_xdatgo(), x_xdatstop()

9.7.15 x_info - Information on NEABX constant (information)

x_info() returns the maximum possible length of a TIDU for the specified connection. The TIDU length depends on the transport system used. You need it for calls for data transfer.

```
#include <cmx.h>
#include <neabx.h>
int   x_info (int *tref,
              struct x_opti1 *x_opt);
```

-> tref

Pointer to the transport reference. Here you specify the transport reference of the connection for which you wish to know the maximum possible length of a TIDU.

<> x_opt

Pointer to a union in which NEABX enters an *x_opti1* structure. The structure *x_opti1* is defined in the file <*neabx.h*>.

```
   struct x_opti1 {
->     int  x_optnr;   /* Option number */
<-     int  x_maxl;    /* Length of a TIDU */
   };
```

x_optnr

Option number. X_OPTI1 is to be specified.

x_maxl

In this field NEABX enters the length of the TIDU. This value specifies how many bytes can be passed to or received by NEABX per call when data is transmitted via the specified connection.

x_maxl is exactly the same value as is returned by the function *t_info()*. NEABX protocol lengths, if any, are NOT taken into account. The rules defined in connection with using the options X_OPTD1, X_OPTD2 and X_OPTD3 with *x_datarq()*, *x_datain()*, *x_xdatrq()* and *x_xdatin()* apply.

Return values

T_OK

The call was successful.

X_ERROR

Error. Query error code using *x_error()*.

Errors

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_info - Query information on CMX (information)” on page 181 may occur. They can be queried by calling *x_error()*.

In addition, the errors listed under *ioctl(2)* may occur.

9.7.16 x_neavi - Analysis of the NEABV protocol

x_neavi() analyzes the NEABV protocol in a data area that has been supplied with values from the network via an *x_conin()* or *x_concf()* call.

The pointer to the user connection message, *x_udatap*, which is returned by *x_conin()* or *x_concf()*, can be passed to *x_neavi()* directly.

x_neavi() interprets the data of the user connection message and writes the information contained in it into the members of the supplied option structure.

The addresses returned in the option structure are subaddresses of the data area *x_udatap* passed by NEABX.

```
int   x_neavi (char *x_udatap,
               int *x_udatal,
               x_optneav *x_opt);
```

-> x_udatap

Pointer to an area containing the NEABV protocol data received by *x_conin()* or *x_concf()* that is to be analyzed by *x_neavi()* and transferred to the structure specified in *x_opt*.

It is best to pass on the data area *x_udatap* returned by *x_concf()* or *x_conin()*.

*x_udatal

Pointer to an area specifying the length of the data area *x_udatap*.

It is best to pass on the value *x_udatal* returned by *x_concf()* or *x_conin()*.

<> x_opt

Pointer to a union *x_optneav* in which NEABX enters the structure *x_optrk*. The structure contains the results of the analysis of the NEABV protocol data passed in *x_udatap*.

Members that are not included in the NEABV protocol are supplied with NULL.

x_opt must be specified, as NEABX will otherwise be unable to pass the results of the analysis to the TS application.

The structures *x_optrk* and the union *x_optneav* are defined in the file *<neabx.h>*.

```
       struct x_optrk {     /* STRUCTURE x_opt FOR
                               COMPUTER INTERCONNECT. */
  ->    int   x_optnr;      /* Option no. = X_OPTRK,
                               X_OPTRK1 */
  <-    int   x_init;       /* Initiative in data transfer */
  <-    int   x_opchl;      /* Length of the OPCH in x_opchp */
  <-    char  *x_opchp;     /* Pointer to OPCH */
  <-    int   x_bvmsgl;     /* Length of the user
                               connection message */
  <-    char  *x_bvmsgp;    /* Pointer to the user
        };                     connection message */
       struct x_optsk {     /* STRUCTURE x_opt FOR STATION
                               INTERCONNECT. */
  ->    int   x_optnr;      /* Option no. = X_OPTSK,
                               X_OPTSK1 */
  <-    int   x_opchl;      /* Length of the OPCH in
                               x_opchp */
  <-    char  *x_opchp;     /* Pointer to OPCH */
  <-    int   x_bvmsgl;     /* Length of the user
                               connection message */
  <-    char  *x_bvmsgp;    /* Pointer to the user
                               connection message */
  <-    int   x_npwl;       /* Length of the network
                               password */
  ->    char  *x_npwp;      /* Pointer to the network
                               password */
        }
```

x_optnr

Option number: Specify:

X_OPTRK or X_OPTRK1

x_init

Initiative for data transfer in the case of computer interconnection.

Possible values: X_MYINIT or X_INITRQ.

The meanings of these values are described in the section “The NEABX service functions (NEABV service)” on page 234.

x_opchl

Length of the operation character OPCH pointed to by *x_opchp*.

NEABX then enters the length contained in the NEABV protocol.

x_opchp

Pointer to the area containing the operation characters (OPCH), or a NULL pointer.

The current length is specified in *x_opchl*.

x_bvmsgl

Length of the area *x_bvmsgp*.

NEABX enters the length contained in the NEABV protocol.

x_bvmsgp

Pointer to an area containing the NEABV user connection message, or a NULL pointer.

The current length is specified in *x_bvmsgl*.

The message is not recoded or handled by *x_neavi()*.

Return values

T_OK

The call was successful. The option structure contains the results of the analysis of the NEABV protocol.

X_ERROR

Error. Query the error code with *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_NVERR2

The value specified in *x_optnr* is invalid.

X_NVERR3

The length specification for one of the protocol elements in the NEABV protocol is too large.

X_NOOPT

No *x_opt* pointer was specified.

See also

x_conin(), x_concf(), x_neavo()

9.7.17 x_neavo - Generate the NEABV protocol

x_neavo() generates the NEABV protocol and places it in a storage area, which can subsequently be made available to an *x_conrq()* or *x_conrs()* call.

The parameters required to create the NEABV protocol must be supplied with values in the structure *x_opt*.

```
int   x_neavo (char *x_udatap,
               int *x_udatal,
               x_optneav *x_opt);
```

<- x_udatap

Pointer to an area in which *x_neavo()* enters the generated NEABV protocol data.

The size of the area is specified in *x_udatal*.

<> x_udatal

Length of the area *x_udatap*.

Before the call, the length of the *x_udatap* area provided is to be entered here.

The maximum required length is 109 bytes.

After the call, the length of the data entered in *x_udatap* is returned here by NEABX. This length consists of the length of the NEABV protocol plus any reserve that may have been allocated for the NEABX protocol. The option number specified in *x_optnr* determines whether or not the reserve for the NEABX protocol is taken into account. The result in *x_udatap* and *x_udatal* can be directly used in the *x_conrq()* or *x_conrs()* call, provided the assignments defined in the table under *x_optnr* are observed.

-> x_opt

Pointer to the union *x_optneav*, containing the structure *x_optrk* defined in the file <*neabx.h*>.

The specification of *x_opt* is mandatory.

```
         struct     x_optrk    {
                                /* STRUCTURE x_opt FOR
                                  COMPUTER INTERCONNECT. */
     ->     int   x_optnr;     /* Option no. = X_OPTRK,
                                  X_OPTRK1 */
     ->     int   x_init;      /* Initiative in data transfer */
            int   x_opchl;     /* Not relevant for output */
            char  *x_opchp;    /* Not relevant for output */
     ->     int   x_bvmsgl;    /* Length of the user
                                  connection message */
     ->     char  *x_bvmsgp;   /* Pointer to the user
                                  connection message */
         }
```

x_optnr

Option number: Specify:

X_OPTRK or X_OPTRK1

If option X_OPTRK1 is used, the reserve need no longer be allowed for in the length specification in *x_udatal*.

Conventions for using options:

Option number with x_neavo()	**Option number with x_con[rq I rs]**	**Reserve for NEABX protocol allowed for x_udatal**
X_OPTRK	X_OPTC1	YES
X_OPTRK1	X_OPTC3	NO

x_init

Initiative for data transfer in the case of computer interconnection.

Possible values: X_MYINIT or X_INITRQ.

If any other value is specified, an error is reported (see also section “The NEABX service functions (NEABV service)” on page 234).

x_bvmsgp

Pointer to an area containing the NEABV user connection message.

The filled length is specified in *x_bvmsgl*.

x_bvmsgl
: Length of the area *x_bvmsgp*.

  The maximum permissible length is X_BVMMXL.

Return values

T_OK
: The call was successful. The NEABV protocol was stored in *x_udatap*. Its length is contained in *x_udatal*.

X_ERROR
: Error. Query the error code with *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_NVERR1
: The allocated storage area *x_udatap* cannot accommodate the NEABV protocol corresponding to *x_opt*. The length *x_udatal* is too small.

X_NVERR2
: An invalid value was specified for *x_optnr*.

X_NVERR4
: For computer interconnection: An invalid value was specified for *x_init*.

X_NVERR5
: Invalid value specified for *x_opchp, x_opchl, x_bvmsgp, x_bvmsgl, x_npwp* or *x_npwl*.

See also

x_conrq(), x_conrs(), x_neavi()

9.7.18 x_perror - Output NEABX error message in decoded form

x_perror() decodes NEABX error messages passed to the process in hexadecimal form by NEABX when *x_error()* is called.

x_perror() writes the plain English form of the NEABX error message specified in *errcod* to the standard error output *stderr*.

The function can only output the plain English form of an error of error type X_BX3 and error class X_NEAERR.

The returned text consists of the error symbol, as defined in *<neabx.h>*, and an explanatory text. The explanatory texts are obtained from a message catalog of the Native Language Support (NLS) facility, if present.

In the *s* parameter an additional explanatory text may be specified, e.g. an indication of the NEABX call and TS application to which the error refers. The text must be passed as a string ("s").

Format of output from *x_perror()*:

x_perror() first writes the text specified with *s* (if s != NULL), then : (colon) and \n (newline).

This is followed by the plain English form of the error message, in three lines:

```
\t<ERROR TYPE symbol> <text from msgcat>\n
\t<ERROR CLASS symbol> <text from msgcat>\n
\t<ERROR VALUE symbol> <text from msgcat>\n
```

(msgcat = message catalog)

For error values that cannot be decoded, a "?" is output.

```
#include <cmx.h>
#include <neabx.h>
void   x_perror (char *s,
                 int errcod);
```

-> s

Pointer to a storage area containing text that is to precede the plain English form of the error message.

-> errcod

For *errcod*, specify the representation of the error message that was passed to the process by NEABX when *x_error()* was called.

Example

1. *x_conin()* returns X_ERROR.

   x_error() returns the hexadecimal value 0x9b0e.

   The call

   ```
   x_perror ( "x_conin" , 0x9b0e)
   ```

   decodes this error as follows:

   ```
   x_conin:
   X_BX3 ICMX(NEA) error
   X_NEAERR Error message of the migration service NEABX
   X_BADPRPI x_conin,x_concf: Protocol ID (=PI) byte incorrect
   ```

2. *x_datarq()* returns X_ERROR.

   x_error() returns the hexadecimal value 0x9b3b.

   The call

   ```
   x_perror ( "x_datarq" , 0x9b3b)
   ```

   decodes this error as follows:

   ```
   x_datarq:
   X_BX3 ICMX(NEA) error
   X_NEAERR Error message of the migration service NEABX
   X_WXOPT x_opt specification incorrect with respect to
   X_MORE/X_END,
             with/without NEABX
   ```

9.7.19 x_redin - Accept redirected connection (redirection indication)

With *x_redin()* a process accepts a connection that another process of the same TS application has redirected to it. This call is required if *x_event()* indicates the event X_REDIN.

When the event X_REDIN is indicated you must accept the redirected connection. If you wish to reject the connection, you may only call *x_redrq()* to pass it on or return it to the original process, or call *x_disrq()* to close it down.

The *x_redin()* call returns:

– the process ID of the calling process,

– the user data that was included with the redirection by the calling process.

If the current process is attached to multiple TS applications, it must use the appropriate means to determine for itself to which TS application the redirected connection belongs. A suitable resource for this purpose is the user data.

```
#include <cmx.h>
#include <neabx.h>
int   x_redin (int *tref,
               int *pid,
               struct x_optc2 *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you received with the *x_event()* call that returned X_REDIN.

<- pid

Pointer to the process ID. NEABX enters here the ID of the process that is redirecting the connection.

<- x_opt

Pointer to the structure *x_optc2*. With this structure you can check the information that the redirecting process sent with its *x_redrq()* call.

The specification for *x_opt* is mandatory, since the migration service must always be in a position to receive a message (the internal x_red protocol).

The structure *x_optc2* is defined in the file <*neabx.h*>.

```
   struct x_optc2 {
->    int x_optnr;      /* Option no. */
<-    char *x_udatap;   /* Data buffer */
<>    int x_udatal;     /* Length of the data buffer */
   };
```

x_optnr
: Option number. Specify X_OPTC2.

x_udatap
: Pointer to a storage area in which NEABX enters the user data.

x_udatal
: Prior to the call you specify the length of the allocated data area *x_udatap*. This area must be large enough to accommodate the user data and must include an additional reserve of X_RED_PL bytes for the x_red protocol.

  The length to be specified is X_RED_SIZE.

  Following the call, *x_udatal* will contain the net length of the user data.

Return values

T_OK
: The call was successful.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_BADREDIR
: Insufficient length specified in *x_udatal*.

X_NOINFO
: TIDU length cannot be determined. The connection was closed down again.

X_NOOPT
: No *x_opt* pointer was specified.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_redin - Accept redirected connection (redirection indication)” on page 185 may occur.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_error(), x_event(), x_disrq(), x_redrq()

9.7.20 x_redrq - Redirect connection (redirection request)

x_redrq() redirects an existing connection to another process of the same TS application. The connection is thereafter no longer known to the redirecting process. NEABX issues the event X_REDIN to the called process.

You may not redirect a connection

- if X_DATASTOP or X_XDATSTOP is present, or
- if the previous *x_event()* call returned X_NOEVENT.

User data may be sent to the receiving process along with the connection redirection. In this user data the current process can indicate to the receiving process to which TS application the connection belongs.

```
#include <cmx.h>
#include <neabx.h>
int   x_redrq (int *tref,
               int *pid,
               struct t_optc2 *x_opt);
```

-> tref
: Pointer to the transport reference. Here you enter the transport reference of the connection that you wish to redirect.

-> pid
: Pointer to the process ID of the called process. Here you specify the ID of the process to which you wish to redirect the connection.

-> x_opt
: Pointer to the structure *x_optc2*. With this structure you can, when redirecting the connection, send user data to the called process. The called process receives the data when it calls *x_redin()*.

  The specification for *x_opt* is mandatory, since the migration service must always be in a position to receive a message (the internal x_red protocol).

  The structure *x_optc2* is defined in the file <*neabx.h*>.

```
   struct x_optc2 {
->    int x_optnr;      /* Option no. */
->    char *x_udatap;   /* Data buffer */
->    int x_udatal;     /* Length of the data buffer */
   };
```

x_optnr
: Option number. Specify X_OPTC2.

x_udatap
: Pointer to a storage area containing user data that NEABX is to pass to the receiving process, plus space reserved for the internal x_red protocol.

  The protocol has a length of X_RED_PL (currently 5) bytes and must appear left-justified in this area.

x_udatal
: Length of the message in *x_udatap*, plus X_RED_PL.

  Maximum value: X_RED_SIZE.

  Maximum transported user data per *x_redrq()* call:

  X_RED_SIZE minus X_RED_PL.

  Minimum specification: X_RED_PL.

Return values

T_OK
: The call was successful.

X_IMPOSSIBLE
: The connection cannot be redirected at present, either because it is not yet completely set up or because transport acknowledgments must still be sent to the partner TS application.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_NOOPT
: No *x_opt* pointer was specified. For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_redrq - Redirect connection (redirection request)” on page 189 may occur.

  In addition, the errors listed under *ioctl(2)* may occur.

See also

x_datain(), x_error(), x_event(), x_xdatin()

9.7.21 x_setopt - Set options in CMX_NEA (set options)

x_setopt can be used to switch options on and off.

The function can be used to activate and deactivate the ICMX(NEA) library trace.

```
#include <cmx.h>
#include <neabx.h>
int x_setopt (int level,
              x_opts *opt);
```

-> level

Specifies in which ICMX(NEA) component the option should be set.

Possible values:

X_LIB

The set option is an ICMX(NEA) library option.

-> opt

Pointer to a union that contains an option structure.

The following structure is defined in <*neabx.h*>:

```
   typedef union x_optset {
   struct x_opts1 opts1 ; /* Control structure */
   } x_opts ;
   struct x_opts1 {
->    int x_optnr;        /* Option no.  */
->    int x_optname;      /* Option name */
->    char *x_optvalue;   /* Pointer to options string */
   };
```

t_optnr

Option number. Specify X_OPTS1.

t_optname

Specifies the option that is to be switched on or off.

Possible values:

X_DEBUG

The library trace mechanism should be switched according to the value in the element *x_optvalue*.

x_optvalue
: Pointer to a (null-terminated) string that contains information on the type and scope of the interface trace to be activated. The format is identical to that of the environment variable NEATRACE in the command neal (see the “Operation and Administration” manual [1] or [2]).

Return value

X_OK
: The call was successful.

X_ERROR
: The option was not set.

Errors

The function *x_error* can be used to ascertain an error value. If the value ranges of the individual parameters are not observed, this error value is of type X_B3, class X_NEAERR and has the value X_WPARAMETER. Other error values are system call errors and correspond to the values of *errno* that are defined in *<errno.h>*.

9.7.22 x_strerror - Decode NEABX error message

x_strerror() can be used to decode the NEABX error messages that are passed to the process in hexadecimal form by NEABX when *x_error()* is called. *x_strerror()* decodes error messages of error type X_XB3 and error class X_NEAERR.

x_strerror() returns a pointer to a static area which contains the plain English form of the NEABX error message specified in *errcod*. Note that the plain English text is passed by *x_strerror()* via the same storage area in each call.

This text consists of error symbols, as defined in <*neabx.h*>, and accompanying text. Each error symbol is preceded by \t. Each accompanying text ends with \n.

The explanatory texts are obtained from the Native Language Support (NLS) facility, if present.

```
#include <cmx.h>
#include <neabx.h>
char   *x_strerror (int errcod);
```

-> errcod
  For *errcod*, specify the representation of the error message that was passed to the process by NEABX when *x_error()* was called.

Return values

If the call was successful, *x_strerror()* returns a pointer to a storage area with the plain English form of the NEABX error message as a string in C notation.

If an undefined value is specified in *errcod*, *x_strerror()* returns a pointer to the text:

```
"\t<errcod> Cannot decode\n"
```

In case of error, *x_strerror()* returns a NULL pointer.

See also

x_error(), x_perror()

9.7.23 x_xdatgo - Release the flow of expedited data (expedited data go)

x_xdatgo() releases the blocked flow of expedited data on the specified connection. The current process informs NEABX that it is again ready to receive expedited data.

The call has the following effects:

– The local TS application can again receive the event X_XDATIN, if it is waiting.

– The remote TS application receives the event X_XDATGO. It may again send expedited data.

x_xdatgo() may only be called if the exchange of expedited data was agreed at connection setup.

```
#include <cmx.h>
#include <neabx.h>
int   x_xdatgo (int *tref);
```

-> tref
    Pointer to the transport reference. Here you enter the transport reference of the connection for which you wish to release the flow of expedited data.

Return values

T_OK
    The call was successful.

X_ERROR
    Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADTABLE
    The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_xdatgo - Release the flow of expedited data (expedited data go)” on page 204 and the following error may occur:

T_WSEQUENCE
    The connection specified in *tref* is not yet fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_event(), x_error(), x_xdatstop()

9.7.24 x_xdatin - Receive expedited data (expedited data indication)

x_xdatin() is called by the current process to receive expedited data sent by the remote TS application. NEABX must have previously reported the event X_XDATIN with *x_event()*, passing in *tref* the transport reference of the connection on which the expedited data arrived. The maximum length for the expedited data depends on the transport system used, but may never exceed X_EXP_SIZE bytes.

If you are not ready to receive expedited data, you can stop the flow of expedited data with *x_xdatstop()*. You thereby prevent NEABX from delivering the event X_XDATIN to the current process. Expedited data that has already been indicated with X_XDATIN, however, must always be completely fetched.

```
#include <cmx.h>
#include <neabx.h>
int   x_xdatin (int *tref,
                char *x_datap,
                int *x_datal,
                x_optd *x_opt);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference that you receive when *x_event()* reports the event >X_XDATIN.

<- x_datap

Pointer to a storage area in which NEABX enters the expedited data received.

If *x_opt* is equal to NULL, NEABX passes the received expedited data to the local TS application. If *x_opt* is not equal to NULL, prior to passing the data to the local TS application NEABX deals with the NEABX protocol in accordance with the specified option number:

X_OPTD1:

The storage area must contain space reserved for the NEABX protocol (mode compatible with CMX V2.1).

X_OPTD2:

The NEABX protocol is invisible to the TS application.

X_OPTD3:
: The NEABX protocol is placed at the beginning of the data area and the length of the protocol is recorded in the *x_offset* member of the *x_optd3* structure, thus informing the TS application as to where the net message begins.

<- x_datal
: Prior to the call specify the length of the expedited data area *x_datap*. This must be at least X_EXP_SIZE. In the call NEABX enters the number of bytes entered that are passed to the local TS application.

<> x_opt
: Pointer to a union *x_optd* containing one of the structures *x_optd1* or *x_optd3*, or the specification NULL. *x_opt* must be specified if it was agreed at connection setup that the NEABX protocol <u>will be used</u> in the data phase. NULL must be specified if the agreement at connection setup was <u>not to use</u> the NEABX protocol in the data phase.

The structures *x_optd1* and *x_optd3* and the union *x_optd* are defined in the file <*neabx.h*>.

```
   struct x_optd1 {
->    int   x_optnr;    /* Option number,
                           X_OPTD1, X_OPTD2 */
<-    int   x_code;     /* Message code */
<-    int   x_strukt;   /* Message structure */
<-    int   x_quit;     /* Transport acknowledgments */
<-    short x_seqno;    /* Message sequence number */
   };
   struct x_optd3 {
->    int   x_optnr;    /* Option number, X_OPTD3 */
<-    int   x_code;     /* Message code */
<-    int   x_strukt;   /* Message structure */
<-    int   x_quit;     /* Transport acknowledgments */
<-    short x_seqno;    /* Message sequence number */
<-    int   x_offset;   /* Offset to the start of data */
   };
```

x_optnr
: Option number. Possible values:

  X_OPTD1 or X_OPTD2
  : for *x_optd1*

  X_OPTD3
  : for *x_optd3*

  The meanings of the values are described under *x_datap*.

x_code
: Designates the message code. Specify X_TRANS; the received expedited data is transparent.

x_strukt
: Message structure. Specify X_ETX. Expedited data can only be received unblocked.

x_quit
: Is only relevant if handling of transport acknowledgments in the TS application was specified at connection setup.

  If a DATA protocol element was received, the following values are possible for *x_quit*:

0	No acknowledgment required.
1	A transport acknowledgment is required.

x_seqno
: Contains the message sequence number, provided x_quit is not NULL.

x_offset
: In this field NEABX returns the length of the NEABX protocol. The TS application is thus informed as to where the net message begins.

Return values

T_OK
: The expedited data has been completely read.

X_DATASTOP
: The expedited data has been completely read. If you wish to send data, you must wait for the event X_DATAGO.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_NOTDTPE
: DATA protocol element expected but not received.

X_QUITPE
: *x_xdatin()* received an acknowledgment protocol element, but transport acknowledgments are not expected as expedited data.

X_BADDTPELI
: The length of the DATA protocol element specified in the received NEABX protocol is invalid.

X_BADXREAD
: The value returned by *x_xdatin()* is greater than 0.

  The specified data area was not large enough to accommodate the complete message; more data is available for this message.

X_NOTDTPE
: DATA protocol element expected, but not received.

X_WPARAMETER
: Invalid parameter; an incorrect value was specified for *x_optnr*.

X_WXOPT
: Invalid *x_opt* specification:

  x_opt != NULL, although no NEABX protocol;

  x_opt = NULL, although NEABX protocol agreed.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_xdatin - Receive expedited data (expedited data indication)” on page 206 may occur.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_error(), x_event()

9.7.25 x_xdatrq - Send expedited data (expedited data request)

x_xdatrq() is used to send expedited data to the receiving TS application, assuming the use of expedited data was agreed at connection setup. With *tref* you specify the connection on which you wish to send the expedited data. The maximum length of the expedited data depends on the transport system used, but may never exceed X_EXP_SIZE bytes.

Expedited data is subject to its own flow control; it may overtake normal data units. Conversely, the transport system guarantees that expedited data will never be overtaken by normal data units.

If *x_xdatrq()* returns X_XDATSTOP, the expedited data has been accepted but the flow of expedited data for the connection is blocked. This may occur on the initiative of the receiving partner application, by means of *x_xdatstop()*, or it may be brought about by NEABX, if the local buffer threatens to overflow. In such cases you must wait, with *x_event()*, for the event X_XDATGO or X_DATAGO before sending more expedited data on the connection.

```
#include <cmx.h>
#include <neabx.h>
int   x_xdatrq (int *tref,
                char *x_datap,
                int *x_datal,
                x_optd *x_opt);
```

-> tref

Pointer to the transport reference. Here you specify the transport reference of the connection on which you wish to send expedited data.

-> x_datap

Pointer to a storage area containing the data that you wish to send. If *x_opt* is not equal to NULL, this area must be X_DRQPHL bytes (datarq protocol header length) larger than the size needed for the net data to be sent, in order to accommodate protocol headers added later. However, it may never be larger than X_EXP_SIZE.

The user connection message must appear left-justified in this area.

-> x_datal

Pointer to a length specification **x_datal*. **x_datal* may never exceed X_EXP_SIZE and has the following meaning:

Case *x_opt* = NULL (see *x_opt*):

**x_datal* corresponds exactly to the length of the data to be sent in *x_datap*.

Maximum value in **x_datal* = X_EXP_SIZE.

Maximum user data per *x_xdatrq()*: X_EXP_SIZE.

Minimum for **x_datal* (send 1 byte of data): **x_datal* = 1.

Case *x_opt* != NULL (see *x_opt*) with option number:

X_OPTD1:

**x_datal* must be specified to be X_DRQPHL bytes larger than the size needed for the net data to be sent. However, the storage space of the TS application is not utilized in forming the protocol.

Maximum value in **x_datal* = X_EXP_SIZE.

Maximum user data per *x_xdatrq()*: X_EXP_SIZE - X_DRQPHL

Minimum value in **x_datal* (1 byte of send data):

**x_datal* = 1 + X_DRQPHL.

X_OPTD2:

**x_datal* contains only the net data length. The storage space of the TS application need not include any space reserved for the NEABX protocol.

Maximum value in **x_datal* = X_EXP_SIZE - X_DRQPHL.

X_OPTD3:

**x_datal* contains only the net data length. In the *x_offset* member of the option structure the TS application records the offset from *x_datap* at which the net data begins. This offset must be equal to X_DRQPHL.

Maximum value in **x_datal* = X_EXP_SIZE - X_DRQPHL.

Minimum value in **x_datal* (send 1 byte of data):

**x_datal* = 1 + X_DRQPHL.

-> x_opt

Pointer to a union that contains the structure *x_optd1*, *x_optd3* or the specification NULL. NULL is mandatory if at connection setup it was agreed that NEABX protocols would not be used in the data phase. NULL may not otherwise be specified.

The structures *x_optd1*, *x_optd3* and the union *x_optd* are defined in the file <*neabx.h*>.

```
   struct x_optd1 {
->    int   x_optnr;    /* Option number,
                           X_OPTD1, X_OPTD2 */
->    int   x_code;     /* Message code */
->    int   x_strukt;   /* Message structure */
->    int   x_quit;     /* Transport acknowledgments */
->    short x_seqno;    /* Message sequence numbers */
   };
   struct x_optd3 {
->    int   x_optnr;    /* Option number, X_OPTD3 */
->    int   x_code;     /* Message code */
->    int   x_strukt;   /* Message structure */
->    int   x_quit;     /* Transport acknowledgments */
->    short x_seqno;    /* Message sequence numbers */
->    int   x_offset;   /* Offset to the start of data */
   };
```

x_optnr

Option number. Possible values:

X_OPTD1 or X_OPTD2
for *x_optd1*

X_OPTD3
for *x_optd3*

The meanings of the values are described under *x_datal*

x_code

Designates the message code. Specify X_TRANS; the expedited data to be sent is transparent.

x_strukt

Message structure. Specify X_ETX. Expedited data can only be sent unblocked.

x_quit

Is only relevant if handling of transport acknowledgments in the TS application was specified at connection setup.

The acknowledgment request bit QVBIT is set in the NEABX protocol.

x_seqno
: Contains the message sequence number, provided *x_quit* is not null.

x_offset
: In the *x_offset* member the TS application records the offset from *x_datap* at which the net data begins. This offset must be equal to X_DRQPHL.

Return values

T_OK
: Call successful; you may continue sending.

X_DATASTOP
: Call successful, but you may not send further expedited data until the event X_XDATGO or X_DATAGO arrives.

X_ERROR
: Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADLEN
: Invalid data buffer length in *x_udatal*.

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

X_BADXCODE
: The value in *x_code* is invalid.

X_WPARAMETER
: Invalid parameter; an invalid value was specified for *x_optnr*.

X_WXOPT
: Invalid *x_opt* specification:

  x_opt != NULL, although no NEABX protocol;

  x_opt = NULL, although NEABX protocol agreed.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_xdatrq - Send expedited data (expedited data request)” on page 208 and the following error may occur:

T_WSEQUENCE
The connection specified in *tref* is not yet fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_error(), x_event(), x_xdatstop()

9.7.26 x_xdatstop - Stop the flow of expedited data (expedited data stop)

x_xdatstop() blocks the flow of expedited data on the specified connection. It tells NEABX that you are not ready to receive expedited data for this connection. An X_XDATIN event that has already been indicated must still be accepted first.

More specifically, the effects of the call are:

– The local TS application no longer receives the events X_XDATIN and X_DATAIN. However, you may call other NEABX functions in the meantime, e.g. to set up an additional connection over which you want to forward the incoming expedited data. You may also send data over the specified connection. *x_xdatastop()* blocks the data flow only for data to be received.

– The sending TS application receives the return value X_XDATSTOP when it calls *x_xdatrq()* for this connection, and X_DATASTOP for *x_datarq()*. It then may send neither expedited data nor normal data units.

If you have already received the event X_XDATIN or X_DATAIN in *x_event()*, you must first read in the waiting data completely. *x_xdatstop()* merely prevents further X_XDATIN and X_DATAIN events from being delivered.

You can release the flow of expedited data again with *x_xdatgo()*.

```
#include <cmx.h>
#include <neabx.h>
int   x_xdatstop (int *tref);
```

-> tref

Pointer to the transport reference. Here you enter the transport reference of the connection for which you wish to stop the flow of expedited data.

Return values

T_OK

The call was successful.

X_ERROR

Error. Query error code using *x_error()*.

Errors

If an error occurs the following error values are possible. They can be queried by calling *x_error()*.

The following error values may occur for error type X_BX3 and error class X_NEAERR:

X_BADTABLE
: The specified transport reference *tref* is unknown to the migration service. It is not present in the relevant table.

For error type T_CMXTYPE and error class T_CMXCLASS, the error values listed in the section “t_xdatstop - Block the flow of expedited data (expedited data stop)” on page 211 and the following error may occur.

T_WSEQUENCE
: The connection specified in *tref* is not yet fully established.

In addition, the errors listed under *ioctl(2)* may occur.

See also

x_datago(), x_error(), x_event(), x_xdatgo(), x_xdatrq()

10 Appendix

10.1 Complete list of CMX error messages

The following tables contain all possible CMX error messages, i.e. all error messages generated at the ICMX(L) and ICMX(NEA) program interfaces. The error messages are sorted by error type and error class.

Error messages with error class = T_DSSYSERR (5) or X_BX2 (8) are not defined in the header files of the CMX program interfaces. They can only be decoded using <*errno.h*>.

Error messages generated at ICMX(L)

Errors with error type (0) = T_CMXTYPE and error class = T_CMXCLASS (0):

Num. value	**Symbolic value**	**Meaning**
0	T_NOERROR	No error
4	T_ENOENT	All internally allocated resources are occupied.
5	T_EIO	The CCP is no longer operational.
12	T_ENOMEM	Not enough working memory.
14	T_EFAULT	Illegal address. One of the specified pointers does not point to the process address space.
22	T_EINVAL	Invalid argument.
100	T_UNSPECIFIED	Error not more precisely specified.
101	T_WSEQUENCE	Function call not permitted in this state.
102	T_WREQUEST	Illegal function call. For SNA interconnection: only the dummy module is linked in the process.
103	T_WPARAMETER	Invalid parameter.
104	T_WAPPLICATION	Unknown application or application already known under this name.

Table 9: At ICMX generated error messages

Num. value	**Symbolic value**	**Meaning**
105	T_WAPP_LIMIT	No more processes in applications may be attached.
106	T_WCONN_LIMIT	Limit for connections reached.
107	T_WTREF	Invalid transport reference.
109	T_COLLISION	Collision in connection setup, disconnection, redirection or in sending data.
110	T_WPROC_LIMIT	Too many processes have attached applica-tions.
111	T_NOCCP	No CCP present for desired application or connection.
112	T_ETIMEOUT	Waiting period elapsed.
113	T_WROUTINFO	Illegal CC list.
115	T_WRED_LIMIT	Too many simultaneously redirected connec-tions.
116	T_WLIBVERSION	Version of CMX library used is incompatible.
117	T_CBRECURSIVE	Recursive call from *t_event* not permitted.
118	T_W_NDS_ACCESS	An NDS error has occured.
119	T_EMUTEX	An error occured during Mutex handling
120	T_NOTSD	Error during addressing of a thread-specific memory

Table 9: At ICMX generated error messages

In addition, in ICMX(L) the following errors, not belonging to error type = T_CMXTYPE and error class = T_CMXCLASS, may also occur:

Num. value	**Symbolic value**	**Meaning**
0	T_NOTSPEC	Error not more precisely specified.
1	T_DIRERR	The specified TS directory is unknown.
2	T_NAMERR	The specified name is not present or the specified name is already present.
3	T_ILLNAM	The name is syntactically invalid.
5	T_PROPER	The requested property is not present or the specified property is already present or syntactically invalid.
7	T_TIMOUT	The TNSX daemon *tnsxd* does not respond within the time limit.
10	T_LEAFNO	More or fewer names were found than expected.
16	T_LENERR	One of the properties has an incorrect length.
20	T_PROT	Error in the protocol for *tnsxd*.
100	T_LFILE	The TS directory has an invalid format.

Table 10: Additional error values at ICMX(L)

Error messages generated at ICMX(NEA)

Errors with error type = X_BX3 (9) and error class = X_NEAERR (0xb):

num. value	**Symbolic value**	**Meaning**
0	X_NOERROR	No error
1	X_DIDSSSTK	Partner has specified neither ETB nor ETX with DDTconnection.
3	X_MAXDAT	More than X_MSG_SIZE bytes of user data received atconnection setup.
4	X_BADLEN	Invalid data buffer length.

Table 11: Error messages generated at ICMX(NEA)

num. value	Symbolic value	Meaning
5	X_BADTRANS	Invalid transport system.
7	X_BADTABLE	No table entry present.
8	X_BADPROT	Invalid protocol ID byte received.
9	X_DIDSSCDE	Data not in EBCDIC with DDT connection.
10	X_SENDQUIT	Error in sending an ACKNOWLEDGEMENT protocol element.
11	X_XSNDQUIT	Error in sending an ACKNOWLEDGEMENT protocol element for expedited data.
12	X_NOOPTDSS	No x_opt specified with station intercon-nection.
13	X_BADXCODE	Invalid transmission code specified or received.
14	X_BADPRPI	Unknown protocol identification byte received.
15	X_NOTCNPE	CONNECT protocol element expected but not received.
16	X_NOTCNATT	CONNECT-ATTENTION protocol element expected but notreceived.
17	X_NOTDTPE	DATA protocol element expected but not received.
18	X_QUITPE	*x_datain* has received an ACKNOWL-EDGEMENT protocol element.
20	X_BADPVBYTE	Invalid protocol version byte received.
21	X_NONEBYTE	With DDT connection no NEABX in the data phase agreed.
22	X_BADDTPELI	Incorrect length received in the length indicator byte.
23	X_BADSTRUKT	Incorrect structure specification in *x_optd1*.
25	X_BADREDIR	Invalid connection redirection.
42	X_BADMSGLEN	Invalid user message length.
43	X_NOXDRDSS	DDT may not call *x_xdatrq()*.

Table 11: Error messages generated at ICMX(NEA)

num. value	**Symbolic value**	**Meaning**
44	X_NOXDIDSS	DDT may not call *x_xdatin()*.
46	X_NOINFO	Cannot determine TIDU length.
47	X_BADXREAD	*t_xdatin* returned value greater than zero.
48	X_QOVERFLOW	Transport acknowledgment can no longer be stored.
49	X_NOOPT	No *x_opt* pointer specified.
50	X_WPARAMETER	Invalid parameter, e.g. incorrect *x_optnr* specified.
51	X_NVERR1	Insufficient length specified for *x_udatal* in *x_neavo()*. The NEABV protocol cannot be stored.
52	X_NVERR2	Illegal option number in *x_neavi()*, *x_neavo()*.
53	X_NVERR2	Illegal length specified in the NEABV protocol in *x_neavi()*.
54	X_NVERR4	Illegal specification for *x_init* in *x_neavo()*.
55	X_NVERR5	Illegal specification for *x_opch*, *x_bvmsg*, *x_npw* in x_neavo()
59	X_WXOPT	Incorrect x_opt specification, e.g. *x_opt* != NULL; *x_opt* != NULL, although no NEABX protocol;*x_opt* != NULL, although second and following data units with *x_datarq()* and *x_datain()*.

Table 11: Error messages generated at ICMX(NEA)

10.2 List of reasons for disconnection

The reasons for disconnection passed by CMX in *reason* following the calls *t_disin()* and *x_disin()* are described below. The symbolic values specified here are numerically defined in <*cmx.h*>. The abbreviation CCP stands for "Communication Control Program", meaning the transport system. "Local CCP" stands for the CCP in the system of the current process, while "partner CCP" stands for the CCP in the system of the connection partner of the current process.

Reasons given by CMX

num. value	Symbolic value	Meaning
0	T_USER	Disconnection by the communication partner; possibly also due to a user error on the part of the communication partner
1	T_RTIMEOUT	Local disconnection by CMX due to inactivity on the connection as specified by the parameter t_timeout.
2	T_RADMIN	Local disconnection by CMX due to deactivation of the CCP by the administration.
3	T_RCCPEND	Local disconnection by CMX due to CCP breakdown

Table 12: Reasons for disconnection - given by CMX

Reasons given by the partner CCP

num. value	Symbolic value	Meaning
256	T_RUNKNOWN	Disconnection by the partner or the CCP; no reason specified.
257	T_RSAPCONGEST	Disconnection by the partner CCP due to a TSAP-specific bottleneck.
258	T_RSAPNOTATT	Disconnection by the partner CCP because the TSAP addressed is not attached there.

Table 13: Reasons for disconnection - given by the partner CCP

num. value	**Symbolic value**	**Meaning**
259	T_RUNSAP	Disconnection by the partner CCP because the TSAP addressed is not known there.
261	T_RPERMLOST	Disconnection by network administration or by adminof partner CCP.
262	T_RSYSERR	Error in network.
385	T_RCONGEST	Disconnection by the partner CCP due to resource bottleneck.
386	T_RCONNFAIL	Disconnection by the partner CCP due to failure in connection setup. Connection setup may fail e.g. because user data is too long or expedited data is not permitted.
387	T_RDUPREF	Disconnection by the partner CCP because a second connection reference was assigned for an NSAP pair (system error).
388	T_RMISREF	Disconnection by the partner CCP due to a connection reference that could not be assigned (system error).
389	T_RPROTERR	Disconnection by the partner CCP due to a protocol error (system error).
391	T_RREFOFLOW	Disconnection by the partner CCP due to connection reference overflow.
392	T_RNOCONN	Establishment of the network connection rejected by the partner CCP.
394	T_RINLNG	Disconnection by the partner CCP due to incorrect header or parameter length (system error).

Table 13: Reasons for disconnection - given by the partner CCP

Reasons given by the local CCP

num. value	Symbolic value	Meaning
448	T_RLCONGEST	Disconnection by the local CCP due to resource bottleneck.
449	T_RLNOQOS	Disconnection by the local CCP because quality of service can no longer be provided.
451	T_RILLPWD	Invalid (connection) password.
452	T_RNETACC	Network access refused
464	T_RLPROTERR	Disconnection by the local CCP due to a transport protocol error (system error).
465	T_RLINTIDU	Disconnection by the local CCP because it received an overly long interface data unit (TIDU) (system error).
466	T_RLNORMFLOW	Disconnection by the local CCP due to violation of the flow control rules for normal data (system error).
467	T_RLEXFLOW	Disconnection by the local CCP due to violation of the flow control rules for expedited data (system error).
468	T_RLINSAPID	Disconnection by the local CCP because it received an invalid TSAP ID (system error).
469	T_RLINCEPID	Disconnection by the local CCP because it received an invalid TCEP ID (system error).
470	T_RLINPAR	Disconnection by the local CCP due to an illegal parameter value, e.g. user data too long or expedited data not permitted.
480	T_RLNOPERM	Connection setup blocked by the adminis-tration of the local CCP.
481	T_RLPERMLOST	Disconnection by the administration of the local CCP.
482	T_RLNOCONN	Connection could not be set up by the local CCP because no network connection available.

Table 14: Reasons for disconnection - given by the local CCP

num. value	**Symbolic value**	**Meaning**
483	T_RLCONNLOST	Disconnection by the local CCP due to loss of the network connection. Most common cause: generation error on CCP and PDN side, e.g. inconsistent link addresses. Error can also occur if partner is not available, modem is faulty or incorrectly set, communication link not plugged in, or data communications board faulty.
484	T_RLNORESP	Connection could not be set up by the local CCP because the partner does not respond to CONRQ. Most common cause: The SINIX computer was not entered in the partner for the processor link via a WAN Solution: enter the processor and region number of the system that has been added in the KOGS of its partner systems in the network.
485	T_RLIDLETRAF	Disconnection by the local CCP due to loss of the connection (Idle Traffic Timeout).
486	T_RLRESYNC	Disconnection by the local CCP because resynchronization was unsuccessful (more than 10 repetitions).
487	T_RLEXLOST	Disconnection by the local CCP because the expedited data channel is defective (more than 3 repetitions).

Table 14: Reasons for disconnection - given by the local CCP

Glossary

active partner

The *communication partner* that sets up a *connection* to another *TS application*.

address

See *TRANSDATA address* and *TRANSPORT ADDRESS*. .

agent

Recipient of network management requests.

API (Application Programming Interface)

APIs are program interfaces that provide the functions of a program system. As the programmer, you use the APIs when programming applications. APIs offer functions for connection management, data exchange, and mapping names to addresses. APIs in the CMX environment are sockets, ICMX, XTI, and TLI.

ASCII

International character set for DP systems (ISO 7-bit code).

CC (Communications Controller)

A CC is a component for connecting a Solaris system to a network. You need a CC to physically attach your system to a subnetwork, unless the interface is integrated on a different module, e.g. the motherboard (onboard interface).

To obtain a logical connection to the network, CCs loaded with the corresponding subnetwork profile. The subnetwork profile is a component of the *CCPs*. Examples of loadable CCs for connecting to X.25, telephone networks and ISDN are PWXV, PWS0 and PWS2.

CCP (Communication Control Program)

A CCP is a program system which, together with one or more *CCs*, provides the logical access of a Solaris system to a *network*. A CCP implements the four lower layers (transport system) of the *OSI reference model* for data communication. A CCP comprises a *subnetwork profile* and *transport service providers*.

CMX (Communications Manager UNIX)

CMX provides communication services for using *CMX applications* and *communication services* in the network, and enables the programming of CMX applications. CMX standardizes the services of different networks and thereby permits utilization of the same CMX application regardless of the underlying network. As the runtime system, CMX switches between the current network environment and CMX applications, and offers the network administrator uniform functions for *OA&M* (Operation, Administration, Maintenance) of *CCPs* and *CCs*. As a development system, CMX provides interfaces (APIs) and procedures for programming network-independent CMX applications.

CMX applications

CMX applications are applications that use the services of CMX. CMX applications have a network address known as the *TRANSPORT ADDRESS*. They are identified uniquely by means of a symbolic name, the *GLOBAL NAME* of an application.

CMX constant

Item predefined for specific computers for CMX, e.g. the length of a *data unit*. Can be queried with the t_info() call.

communication method

An access method for the transport services defined in the *OSI Reference Model*.

communication partner

A *TS application* that maintains a logical connection to another *TS application* and exchanges data with it.

connection, virtual

An association between two *communication partners* which allows them to exchange data with each other.

data unit

The set of characters that can be sent in one go with the t_datarq() call or received in one go with t_datain().

DCAM application

A *TS application* in BS2000 which uses the DCAM access method.

EBCDIC

EBCDIC is an extended 8-bit version of BCD code which is used on BS2000 mainframes, TRANSDATA communication computers and IBM-compatible systems.

ETSDU

Expedited data unit.

GLOBAL NAME of an application

Each *CMX application* identifies itself and its communication partners in the network by symbolic, hierarchical GLOBAL NAMES. A GLOBAL NAME consists of up to five name parts (NP[1- 5]), which you can use to define the application (NP5), the processor (NP4), and (up to three) administrative domains (NP[3-1]).

Example: The GLOBAL NAME “YourApplication.D018S065.mch-p.sni.de” means: “YourApplication” resides on the host “D018S065” in the domain “mch-p.sni.de”.

When you, as administrator, are choosing a GLOBAL NAME, you must adhere to the regulations and recommendations of the specific application.

As an administrator you can assign properties to the GLOBAL NAME of an application. You can, for example, assign a *TRANSPORT ADDRESS* or a *LOCAL NAME*. As a programmer, you can obtain the TRANSPORT ADDRESS or the LOCAL NAME from the GLOBAL NAME with the aid of the function calls t_getaddr() and t_getloc().

ICMX

Standard transport system interface for applications.

KOGS (configuration-oriented generator language)

KOGS is the configuration-oriented generator language with which the physical and logical properties of the subnetwork interfaces of a processor are described in a text file. Language elements of KOGS are macros, operands, and operand values. Normally, the system administrator or network administrator defines the specific properties of a subnetwork interface using the *CMXGUI*. KOGS is only used in exceptional cases.

LOCAL NAME of an application

A CMX application uses the LOCAL NAME to attach to CMX in its local system for communication. The LOCAL NAME comprises one or more *T-selectors*, which identify the transport system via which the CMX application is to communicate. As the administrator, you can enable or disable the communication of a CMX application via particular transport systems and fulfill any requirements of the CMX application for specific T-selector values, e.g. in file transfer.

As an administrator, you can assign the LOCAL NAME of an application to the *GLOBAL NAME* of the application. As a programmer, you can obtain the LOCAL NAME from the GLOBAL NAME using the function call t_getloc().

message

A logically related set of data which is to be sent to a *communication partner*.

migration service

Service in CMX for adapting a *CMX application* to meet the requirements of *TS applications* in PDN and BS2000 that use NEA transport protocol functions not available in ISO transport protocols.

NEABX

Migration protocol for converting from an NEA transport system to an ISO transport system.

OSI Reference Model

Open Systems Interconnection is the communication architecture defined by the International Organization for Standardization (ISO) in ISO standard 7498. This architecture defines reliable data interchange between applications running on different hardware platforms. To perform this complex task, the OSI Reference Model distinguishes between seven interoperating subtasks, each of which is implemented on a particular layer. The lower four layers represent the *transport system*, while the top three layers represent the view of the *application*, e.g. the data formats.

partner

see *communication partner*.

passive partner

The *communication partner* that does not set up a *connection* itself but is addressed by another communication partner.

PDN application

A TS application that runs in a communication computer.

process

A process is a program during execution. It consists of the executable program, the program data, and process-specific administration data required to control the program.

processor

Network-wide addressable entity in a host or communication computer which provides the functionality of the transport service.

processor name

Part of the *TRANSDATA address*. The processor name is specified in the form: processor number/region number.

property

Attribute of a *TS application* in the *TS directory*, where the application is registered together with the *GLOBAL NAME*.

station

Terminal in the data communication system; addressable network-wide for transport service purposes.

station interconnection

Way of connecting a SINIX system to a network. The applications in the SINIX system are generated as stations in the adjacent computer.

station name

Part of the *TRANSDATA address*. The station name corresponds to the *LOCAL NAME* of the *TS application*.

TEP

XTI transport endpoints and in CMX attached processes or threads from *TS applications*.

TNS (Transport Name Service)

The TNS is a component of *CMX* which supports the correct mapping of the *GLOBAL NAMES* of *CMX applications* in the network to *TRANSPORT ADDRESSES* and *LOCAL NAMES*. As the administrator, you configure your chosen assignment of GLOBAL NAME to TRANSPORT ADDRESS for remote applications, as well as the assignment of GLOBAL NAME to LOCAL NAME for local applications. As the applications programmer, you can use these maps via an *API* and thereby work solely with the GLOBAL NAMES of applications without assessing the maps.

The TNS provides network-wide identification of applications by means of logical GLOBAL NAMES and their mapping to corresponding *network addresses*. This means that you can identify applications without having to know their network addresses. Together with the *FSS*, the TNS provides a complete mapping of the logical name to a concrete *subnetwork address* and a *route* through the various subnetworks of the network.

TRANSPORT ADDRESS of an application

A calling *CMX application* transfers the TRANSPORT ADDRESS of a called communication partner to *CMX* when communication is being established. CMX uses the TRANSPORT ADDRESS to locate the communication partner in the network and determine a *route* through the network. The TRANSPORT ADDRESS generally depends on the logical and physical structure of the network (and its subnetworks). The TRANSPORT ADDRESS contains the specifications of your network operator(s) which are specific to your network. As the administrator, you can influence the TRANSPORT ADDRESS and hence the communication paths independently of the application.

The components of a TRANSPORT ADDRESS are: a network address for uniquely identifying the remote system on which the application resides, the type of *transport system* via which the remote application can be reached, and the *T-selector* that identifies the remote application in the remote system.

Examples of network addresses are: the Internet address in dot notation “192.11.44.1”, the NEA network address in the notation processor/region number “47/11”, and the X.25 address (DTE address) as a string of digits “45890010123”.

As an administrator, you can assign a TRANSPORT ADDRESS of the application to the *GLOBAL NAME* of the application. As a programmer, you can obtain the TRANSPORT ADDRESS from the GLOBAL NAME using the function call t_getaddr().

TRANSDATA address

A character string which uniquely identifies an addressable entity in the TRANSDATA network. It is made up of the *processor name* and the *station name*.

Transport Layer

Fourth layer in the *OSI Reference Model*; described in ISO standard 8072.

transport reference

A number which uniquely identifies a *connection* within a *TS application*.

transport system

The transport system is represented by the four lower layers of the *OSI Reference Model*. A *CCP* implements the four layers of the transport system. The transport system guarantees the secure exchange of data between systems whose *applications* communicate with each other, regardless of the underlying network structures. The transport system uses protocols for this purpose.

TSAP

Used by a *TS application* to access the transport system.

TS application

Transport service application:

A TS application is an application that uses the services of the transport system. It consists of programs that can set up a virtual *connection* to another TS application in order to exchange data with it.

TS directory

Database containing information about *TS applications*. The TS directory is managed using the *Transport Name Service in SINIX*.

T-selector

The T-selector identifies a communication application within the system on which the application is running. Together with the *network address* of the system, the T-selector forms the *TRANSPORT ADDRESS* of an application which uniquely identifies this application within the network.

TSP (Transport Service Provider)

A TSP is a component of a *CCP* or of *CMX* which, with the exception of the NTP (null transport), provides the OSI transport service in the network using a transport protocol. As the administrator, you can determine the usage of a particular TSP for the communication of *applications*. RFC1006 is the TSP in CMX which, together with TCP/IP, provides the OSI transport service in the Internet. NTP (null transport) offers *CMX applications* direct access to the network services of the X.25 subnetwork. TP0/2, TP4, and NEA are the TSPs for an OSI environment and the TRANSDATA network.

Together with a *subnetwork profile*, a TSP forms a *transport system*. It offers a set of configurable runtime and tuning parameters, assesses the *TRANSPORT ADDRESS*, and finds a suitable route through the network. To do this, the TSP uses your specifications in the *FSS*, if necessary.

TSDU

Message.

UTM application

A transaction processing application in BS2000 which can also communicate with other applications.

XTI

The standard program interface to transport services defined by X/Open.

Abbreviations

ASCII	American Standard Code for Information Interchange
CC	Communications Controller
CF	Configuration file
CCITT	Comite Consultatif International Telegraphique et Telephonique
CCP	Communication Control Program
CMX	Communications Manager in Solaris
DCAM	Data Communication Access Method
DMA	Direct Memory Access
EBCDIC	Extended Binary Coded Decimal Interchange Code
EBNF	Extended Backus Naur Form
EMDS	Emulation Datensichtstation
EOF	End of File
EOS	End of String

ETHN	ETHERNET
ETSDU	Expedited Transport Service Data Unit
FT	File Transfer
FSB	Forwarding Support Base
ICMX	Program interface of CMX
IS	Intermediate System
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ITU	International Telecommunication Union
ITU-T	Telecommunication Standardization Sector
KOGS	Configuration-oriented generator language
LAN	Local Area Network
MES	German abbreviation for “menu development system”
MSV1	Communication protocol: Medium Speed Variant 1

NEA	Network architecture for TRANSDATA systems
NSAP	Network Service Access Point
OSI	Open Systems Interconnection
PDN	Program system for telecommunication and network control
PID	Process Identifier
PVC	Permanent Virtual Circuit
REMOS	Remote Operation System for linking LANs
SNA	Systems Network Architecture
SNID	Subnet identification
SNPA	Subnet Point of Access
TCEP	Transport Connection Endpoint
TCP/IP	Transmission Control Protocol/Internet Protocol
TEP	Transport Endpoint
TIDU	Transport Interface Data Unit

TLI
Transport Level Interface

TNS
Transport Name Service in Solaris

TSAP
Transport Service Access Point

TSDU
Transport Service Data Unit

TSTAT
TEP Status

VAR
German abbreviation for “host computer”

WAN
Wide Area Network

XTI
X/Open Transport Interface

Related publications

The manuals are available as online manuals, see *http://manuals.fujitsu-siemens.com*, or in printed form which must be payed and ordered separately at *http://FSC-manualshop.com*.

[1] **CMX V6.0** (Solaris)
Operation and Administration
User Guide

Target group
System administrators

Contents
The manual describes the function of CMX as mediator between applications and the transport system. It contains basic information on configuration and administration of systems in network environments.

[2] **CMX V5.1** (Reliant UNIX)
Communications Manager in UNIX
Operation and Administration
User Guide

Target group
System administrators

Contents
The manual describes the function of CMX as mediator between applications and the transport system. It contains basic information on configuration and administration of Reliant UNIX systems in network environments.

[3] **XTI V6.0**
X/Open Transport Interface
User Guide

Target group
Programmers of TS applications

Contents
The manual contains implementation-specific supplements to the function calls of XTI.

[4] **CMX/CCP V6.0** (Solaris)
WAN Communication
User Guide

Target group
Network administrators and system administrators

Contents
The manual describes the computer-to-computer connection via WAN (Wide Area Network) allowing communication in the remote area (Wide Area Network, WAN).

[5] **CCP-WAN V5.1** (Reliant UNIX)
Communication Control Program
User Guide

Target group
System administrators

Contents

– Computer-to-computer connection via WAN (Wide Area Network)
– WAN connections of CCP-WAN
– Protocol profiles of CCP-WAN
– Operation and administration of CCP-WAN
– Configuration of the protocol profiles

[6] **CMX/CCP V6.0** (Solaris)
ISDN Communication
User Guide

Target group
Network administrators

Contents
The manual describes the computer-to-computer connection via ISDN (Integrated Services Digital Network).

[7] **CCP-ISDN V5.1** (Reliant UNIX)
Communication Control Programm
User Guide

Target group
Network administrators

Contents
The manual describes the computer-to-computer connection via ISDN (Integrated Services Digital Network) allowing communication in the remote area (Wide Area Network, WAN).

[8] **CMX V6.0** (Solaris)
TCP/IP via WAN/ISDN
User Guide

Target group
Network and system administrators

Contents
The manual describes how CMX enables the connectionless IP traffic via the connection-oriented WAN.

[9] **CS-ROUTE V2.0** (Reliant UNIX)
User Guide

Target group
Network and system administrators

Contents
Description of CS-ROUTE which enables the TCP/IP- OSI TP4/CLNP network access to X.25 and ISDN networks and the parallel LAN-WAN routing of IP- and CLNP packages via X.25 and ISDN.

[10] **CMX V6.0**
CS-GATE
User Guide

Target group
This manual is intended for network and system administrators.

Contents
This manual describes how you use CS-GATE to establish LAN-WAN-, LAN-LAN- and LAN-WAN-LAN gateways. It describes addressing, address conversion functions, configuration, control and diagnosis for CS-GATE.

[11] **Reliant UNIX 5.45**
Network Administration
System Administrator’s Guide

Target group
System administrators

Contents
This manual describes the network administration activities, that have to be performed when using the TCP/IP software on Reliant UNIX 5.45 as well es the basic network function (BNU).

Index

Fujitsu Siemens Computers GmbH
User Documentation
81730 Munich
Germany

Fax: (++49) 700 / 372 00000

email: manuals@fujitsu-siemens.com
http://manuals.fujitsu-siemens.com

Comments
Suggestions
Corrections

Submitted by

Comments on CMX V6.0
Programming Applications

Fujitsu Siemens Computers GmbH
User Documentation
81730 Munich
Germany

Fax: (++49) 700 / 372 00000

email: manuals@fujitsu-siemens.com
http://manuals.fujitsu-siemens.com

Comments
Suggestions
Corrections

Submitted by

Comments on CMX V6.0
Programming Applications

U41136-J-Z145-3-76