English

FUJITSU Software BS2000

CRTE V10.1A

Common Runtime Environment

User Guide

Edition December 2019

Comments… Suggestions… Corrections…

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

Feel free to send us your comments by e-mail to:
bs2000services@ts.fujitsu.com senden.

Certified documentation according to DIN EN ISO 9001:2015

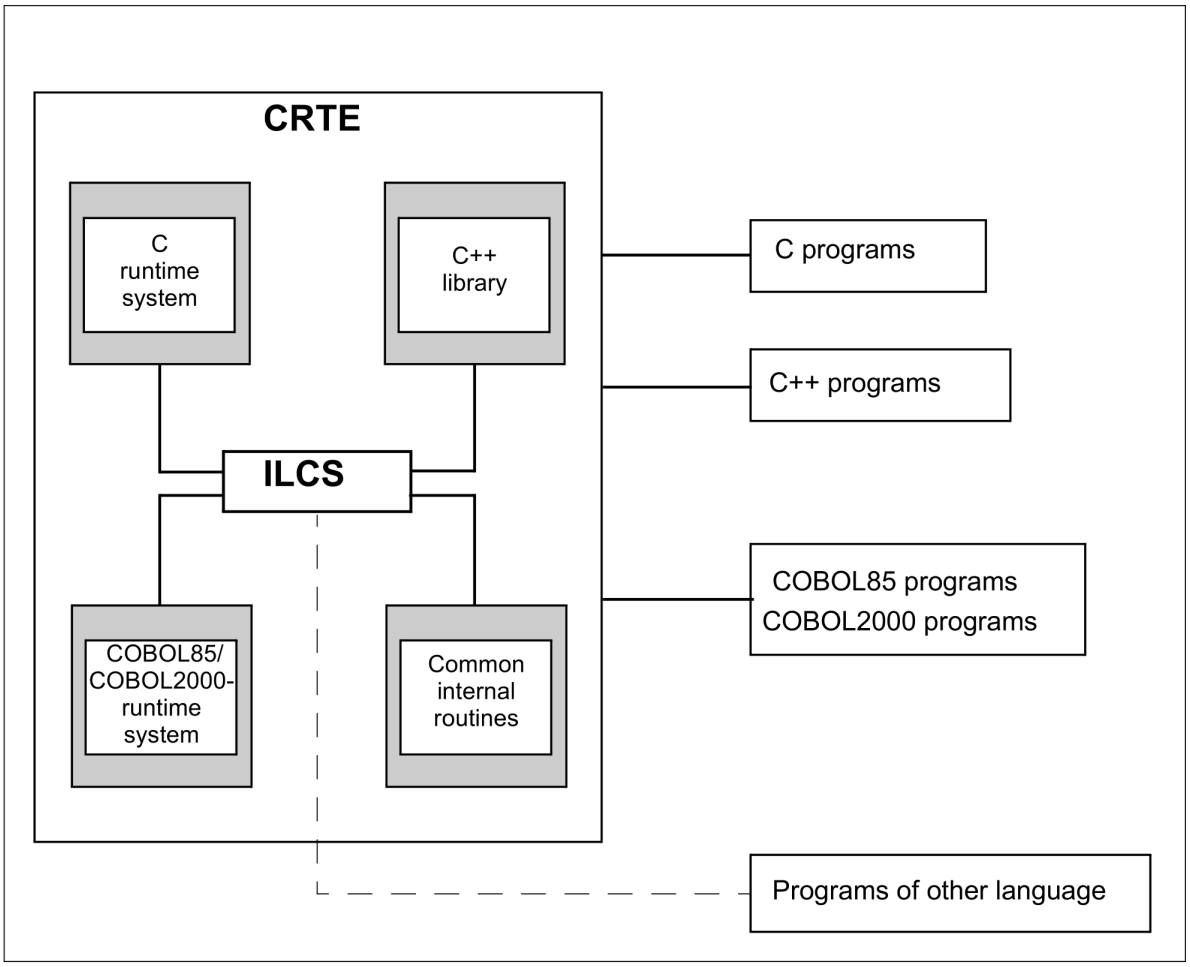
To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001 .

Copyright and Trademarks

Table of Contents

1 Preface

CRTE is the **C**ommon **R**un**T**ime **E**nvironment for C, C++ and COBOL85 or COBOL2000 programs in BS2000.
In addition to the runtime systems for C, C++ and COBOL85/COBOL2000, CRTE includes all routines of the Inter-Language Communication Services (ILCS) program communication interface.
Accordingly, CRTE must be used in order to ensure that program systems comprising programs in various languages will run successfully.



1.1 Objectives and target groups of this manual

This manual is intended for BS2000 programmers and system administrators who want to use the common runtime environment for COBOL85, COBOL2000, C and C++ objects as well as for combinations of other languages.

1.2 Summary of contents

This manual describes the components, the installation and the operation of CRTE.

The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the user ID under which the product's files are stored.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at http://manuals.ts.fujitsu.com.

1.3 Changes since the last edition of the manual

This manual contains the following changes in repect to the last edition for version V10.0B:

- New libraries to support the standard ISO/IEC C++ 2017.
- The names of the delivered files have been adapted to the current CRTE version.

1.4 Notational conventions

The following notational conventions are used in this manual for the presentation of important elements in the text:

> ¡ For comments

> ! **CAUTION!**
>
> For warnings

Italic font

Identifies a variable for which you must specify a value.

`Typewriter font`

Used for input into the system, system output and file names in examples.

command

In the command syntax descriptions, those items (command and parameter names) that have to be entered unchanged are printed in bold.

2 Selectable unit, installation and shareability of CRTE

This chapter provides information concerning the following topics:

2.1 CRTE V10.1A selectable unit

The CRTE V10.1A selectable unit comprises the following PLAM libraries and cataloged files:

Library / file name	Contents
SYSLNK.CRTE.101	Individual modules for complete linking of the C runtime system COBOL85/COBOL2000 runtime system modules ILCS program communication interface modules and macros Modules for shareable internal runtime routines
SYSLNK.CRTE.101.PARTIAL-BIND	Linkage modules to the dynamically linkable C and COBOL runtime system Modules of the COBOL85/COBOL2000 and C/C++ runtime system that cannot be preloaded ILCS program communication interface modules Modules for shared, dynamically loadable, internal runtime routines
SYSLNK.CRTE.101.COMPL	Linkage modules to the dynamically linkable C and COBOL runtime system
SYSLNK.CRTE.101.CPP	C++ library function modules and include headers for C++ V2.1 or higher (Cfront)
SYSLNK.CRTE.101.CFCPP	Runtime system for C/C++ as of V3.0 (Cfront C++)
SYSLNK.CRTE.101.COMPV2	Modules for running linked C V2.0 programs
SYSLNK.CRTE.101.COMPV1	Modules for relinking C V1.0 objects
SYSLNK.CRTE.101.POSIX	Link switch for POSIX library functions (incl. time functions)
SYSLNK.CRTE.101.TIME	Link switch for POSIX time functions
SYSLNK.CRTE.101.TIMESHIFT	Link switch for shifting the reference day (epoch) for time functions (in the current version without effect, is supported for compatibility reasons only)
SYSLNK.CRTE.101.TIME38	Time functions for special applications
SYSLNK.CRTE.101.TIME50	Link switch for processing of time stamps, which have been created with 1/1/1950 as the reference date (epoch)
SYSLNK.CRTE.101.STDCPP	C++ standard library for C/C++ as of V3.0 (ANSI C++)
SYSLNK.CRTE.101.SHARE	Shareable COBOL85/COBOL2000 runtime system (main module ITC...) Shareable C runtime system (main module IC@...)
SYSLNK.CRTE.101.RTSCPP	Runtime system for C/C++ as of V3.0 (ANSI C++)

SYSLNK.CRTE.101.CPP-COMPL	Linkable main modules for the C++ runtime system, the standard C++ library and the C++ tool library
SYSLNK.CRTE.101.TOOLS	C++ library Tools.h++ for C/C++ as of V3.0 (ANSI C++)
SYSLNK.CRTE.101.CXX01	Standard C++ library and runtime system for C/C++ as of V4.0 (standard C++ 2017)
SYSLIB.CRTE.101	Include headers and macros for the C/C++ library functions ILCS macros
SYSLIB.CRTE.101.CPP	C++ library function include header (Cfront)
SYSSSC.CRTE.101.C SYSSSC.CRTE.101.C.LOW SYSSSC.CRTE.101.PARTIAL SYSSSC.CRTE.101.PARTIAL.LOW SYSSSC.CRTE.101.COB-PART SYSSSC.CRTE.101.COB-PART.LOW SYSSSC.CRTE.101.COBOL SYSSSC.CRTE.101.COBOL.LOW SYSSSC.CRTE.101.SIS SYSSSC.CRTE.101.SIS.LOW	Subsystem declaration files for C, COBOL85/COBOL2000 and the shared components (PROSOS)
SYSSII.CRTE.101	IMON information file
SINPRC.CRTE.101	Procedures (for example ICXPART) for corrections in the C linkage module if CRTE is installed under a nonstandard user ID
SINLIB.CRTE.101	POSIX and TIME link switch and include header of the C library for the installation of the POSIX file system
SINLIB.CRTE.101.CXX01	Include-Header of the C++ 2017 library functions
SKULNK.CRTE.101	Module library (X86 code) without COBOL runtime system
SKULNK.CRTE.101.PARTIAL-BIND	Modules for standard partial bind (X86 code) without COBOL runtime system
SKULNK.CRTE.101.COMPL	Modules for complete partial bind (X86 code) without COBOL runtime system
SKULNK.CRTE.101.POSIX	Link switch for POSIX (incl. time functions)
SKULNK.CRTE.101.TIME	Link switch for POSIX time functions
SKULNK.CRTE.101.TIMESHIFT	Link switch for shifting the reference day (epoch) for time functions (in the current version without effect, is supported for compatibility reasons only)
SKULNK.CRTE.101.TIME38	Time functions for special applications

SKULNK.CRTE.101.TIME50	Link switch for processing of time stamps, which have been created with 1/1/1950 as the reference date (epoch)
SKULNK.CRTE.101.RTSCPP	X86 variant of the C++ runtime system
SKULNK.CRTE.101.CPP-COMPL	Linkable main modules for the C++ runtime system, the standard C++ library and the C++ tool library
SKULNK.CRTE.101.STDCPP	X86 variant of the standard C++ library
SKULNK.CRTE.101.TOOLS	X86 variant of the library Tools.h++
SKULNK.CRTE.101.CXX01	X86 variant of the standard C++ library and the C++ 2017 runtime system
SKUSSC.CRTE.101.PARTIAL SKUSSC.CRTE.101.SIS	Subsystem declaration files for X86 variant
SYSDOC.CRTE.101.CXX01.OSS	Readme and licenses of the open source software included in the C++ 2017 runtime system
SYSFGM.CRTE.101.D SYSFGM.CRTE.101.E	Release Notice - German Release Notice - English

C V2.0 and ILCS modules (SYSLNK.CRTE-BASYS.101.CLIB and SYSLNK.CRTE-BASYS.101.ILCS) are delivered with CRTE-BASYS.

The *.PTH-files are delivered with CRTE-BASYS.

The CRTE message files are components of BS2000 basic configuration.

2.2 Installing CRTE

This section provides information concerning the following topics:

- CRTE libraries for installation without version specification
- Standard installation under the user ID “$.”
- Installing with IMON under a non-standard user ID
- Installing header files and POSIX link switches in the default POSIX directory
- Installing header files and POSIX link switches in any POSIX directory
- Private installation
- Switching from an earlier CRTE version to CRTE V10.1A

2.2.1 CRTE libraries for installation without version specification

The following CRTE libraries must be installed with names which do not contain version information (“101”):

SKULNK.CRTE.101
SKULNK.CRTE.101.CXX01
SKULNK.CRTE.101.POSIX
SKULNK.CRTE.101.RTSCPP
SKULNK.CRTE.101.STDCPP
SKULNK.CRTE.101.TOOLS
SKULNK.CRTE.101.TIME
SKULNK.CRTE.101.TIMESHIFT
SKULNK.CRTE.101.TIME38
SKULNK.CRTE.101.TIME50
SKULNK.CRTE.101.PARTIAL-BIND
SKULNK.CRTE.101.COMPL
SKULNK.CRTE.101.CPP-COMPL
SYSDOC.CRTE.101.CXX01.OSS
SYSLIB.CRTE.101
SYSLIB.CRTE.101.CPP
SYSLIB.CRTE.101.CXX01
SYSLNK.CRTE.101
SYSLNK.CRTE.101.CXX01
SYSLNK.CRTE.101.COMPV1
SYSLNK.CRTE.101.COMPV2
SYSLNK.CRTE.101.CPP
SYSLNK.CRTE.101.CFCPP
SYSLNK.CRTE.101.PARTIAL-BIND
SYSLNK.CRTE.101.COMPL
SYSLNK.CRTE.101.CPP-COMPL
SYSLNK.CRTE.101.POSIX
SYSLNK.CRTE.101.RTSCPP
SYSLNK.CRTE.101.STDCPP
SYSLNK.CRTE.101.TIME
SYSLNK.CRTE.101.TIMESHIFT
SYSLNK.CRTE.101.TIME38
SYSLNK.CRTE.101.TIME50
SYSLNK.CRTE.101.TOOLS

If installation is performed with the IMON product (**I**nstallation **Mon**itor) then the above libraries are automatically copied to libraries **without** a version specification.

If installation is not performed with IMON, you must rename the libraries and POSIX-HEADER libraries listed above yourself to files with no version specification. To do this, you can use the ICXINST procedure from the SINPRC. CRTE.101 library.

2.2.2 Standard installation under the user ID “$.”

The standard installation of the CRTE and the POSIX-HEADER is normally performed with the product IMON (**I**nstallations **Mon**itor).

It is recommended that you install the CRTE in standard installation mode, i.e. under the standard system ID ($). If you perform installation using IMON then the libraries listed on "CRTE libraries for installation without version specification" are automatically renamed to libraries without a version specification. If you perform installation without IMON, you must rename the libraries “manually” or use the ICXINST procedure (see "CRTE libraries for installation without version specification").

2.2.3 Installing with IMON under a non-standard user ID

If you install CRTE under a non-standard user ID with IMON, IMON will perform all the necessary steps automatically.

2.2.4 Installing header files and POSIX link switches in the default POSIX directory

The POSIX library functions which are provided in CRTE can be used provided that a POSIX subsystem is available.

You should perform the following steps in order to set up the software configuration required for the use of CRTE-POSIX functions:

1. Install CRTE in POSIX. To do this, call:

   ```
   /START-POSIX-INSTALLATION
   ```

2. Install the POSIX headers in POSIX.
3. Copy the library SYSLIB.POSIX-HEADER.101 (BS2000 OSD/BC V10.0 Release Unit) to a library named SYSLIB.POSIX-HEADER (name without version specification). To do this, use the procedure ICXINST from the library SINPRC.POSIX-HEADER.101. If IMON is used for installation then it does this automatically.

2.2.5 Installing header files and POSIX link switches in any POSIX directory

If you work under POSIX, you can install the header files “privately” in a freely selectable directory under POSIX. This allows you to run CRTE V10.1A in POSIX together with a privately installed C/C++ compiler as of V3.0 in parallel with an older CRTE version.

To do this, install the CRTE headers and the POSIX-HEADERs separatelyin the `ufs` under POSIX with the ICXINSTALL procedures:

```
/CALL-PROC *LIB(SINPRC.CRTE.101,ICXINSTALL), PROC-PAR=(directory='directory')

/CALL-PROC *LIB(SINPRC.POSIX-HEADER.101,ICXINSTALL), -

/                          PROC-PAR=(directory='directory')
```

The headers are stored under *directory*`/usr/include`, and the link switches under *directory*`/opt/CRTE/lib`.

The ICXDELETE procedures are provided for deinstallation:

```
/CALL-PROC *LIB(SINPRC.CRTE.101,ICXDELETE), PROC-PAR=(directory='directory')

/CALL-PROC *LIB(SINPRC.POSIX-HEADER.101,ICXDELETE), -

/             PROC-PAR=(directory='directory')
```

2.2.6 Private installation

CRTE allows you to perform a private installation without IMON. So you can run the new CRTE version under a “private” user ID parallel to an “official” version already installed in the system.

You must transfer and rename the files using the procedure ICXINST to and then call the procedure ICXPPART:

```
/CALL-PROC *LIB-ELEM(SINPRC.CRTE.101,ICXPPART),PROC-PAR=('privateUserid')
```

The procedure ICXPPART modifies the C linkage module, the COBOL linkage module and the linkage module for the shared runtime routines in the libraries SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL and SYSLNK.CRTE to ensure that the loadable C runtime system, the COBOL runtime system and the shared components from the library $*privateUserid*.SYSLNK.CRTE are loaded independently of preloaded subsystems and IMON access.

You must also modify the default settings of the USER-INCLUDE-LIBRARY option and the STD-INCLUDE-LIBRARY option of the C and C++ compiler accordingly. The C/C++ compiler must then also be installed without IMON.

The steps required in order to perform a private installation of the CRTE are summarized below:

1. Transfer the files to the required ID.
2. Start the procedures ICXINST for CRTE and POSIX-HEADER.

   The procedure removes the version specifications from the library names:

   ```
   /CALL-PROC *LIB(SINPRC.CRTE.101,ICXINST)
   ```

   ```
   /CALL-PROC *LIB(SINPRC.POSIX-HEADER.101,ICXINST)
   ```
3. Call the procedure ICXPART:

   ```
   /CALL-PROC *LIB(SINPRC.CRTE.101,ICXPPART), PROC-PAR=('privateUserid')
   ```

   In the case of standard partial bind and complete partial bind methods, this procedure replaces the `$.SYSLNK.CRTE` with $*privateUserid*`.SYSLNK.CRTE` in the adapter modules.

   These modifications are required if the PARTIAL-BIND library or COMPL library are used during linking.
4. If you compile your programs under POSIX you must also call the procedure ICXINSTALL to install the header files and link switches under a private path (see section “Installing header files and POSIX link switches in any POSIX directory”).

2.2.7 Switching from an earlier CRTE version to CRTE V10.1A

Proceed as follows when switching from an earlier version to CRTE V10.1A:

1. Before installing the header files supplied with CRTE V10.1A and POSIX-HEADER V10.1A in the POSIX system (see section “Installing header files and POSIX linkswitches in the default POSIX directory”) you must first remove the header files of the earlier CRTE version and POSIX-HEADER version. To do this, you need to use the deinstallation procedures from the old CRTE and POSIX-HEADER product files:
   - In the case of a standard installation, use the POSIX installation tool (/START-POSIX-INSTALLATION) to perform the delete operation and choose the function “Delete packages from POSIX”.
   - In the case of a private installation use the procedures from the earlier version of CRTE and POSIX-HEADER to perform the delete operation:

     `/CALL-PROC *LIB(SINPRC.CRTE.`*`oldversion`*`, ICXDELETE)`
     `/CALL-PROC *LIB(SINPRC.POSIX-HEADER.`*`oldversion`*`, ICXDELETE)`
2. You may now delete all the files from the earlier version.
3. If the subsystems listed below are preloaded, you must stop them and remove them from the subsystem catalog or replace them with the corresponding version V10.1A subsystems before installing CRTE V10.1A:

   CRTEC, CRTECOB, CRTESIS, CRTESIK, CRTEPART, COBPART, CRTECOM, CRTEPARK

   If you are performing a standard CRTE V10.1A installation with IMON then the new subsystem entries are automatically generated.

> ¡ As of CRTE V2.1, ILCS (CRTECOM) is no longer supplied as a preloadable subsystem. If you have been working with a CRTE version earlier than V2.1, you must delete the entry for CRTECOM from the subsystem catalog because it has no equivalent in CRTE V10.1A.

2.3 Shareability of CRTE

With the exception of a few routines, the entire CRTE is shareable.

The shareable CRTE is divided into the following subsystems:

- CRTECOB for COBOL85 and COBOL2000,
- CRTEC for C/C++,
- CRTEPART for the partial bind link method on /390 (C/C++)
- COBPART for the partial bind link method on /390 (COBOL)
- CRTESIS for the shareable components

The system administrator can load these subsystems into class 4 memory.

> If the subsystems are loaded into the high address space of class 4 memory, you must specify PROG-MODE=ANY in the START-PROGRAM command when calling programs which dynamically link the corresponding runtime system.

Generating the subsystem catalog

The following subsystem declaration files are provided as input files for the generation of the subsystem catalog:

SYSSSC.CRTE.101.C

CRTEC for C/C++ in class 4 memory (high address space)

SYSSSC.CRTE.101.C.LOW

CRTEC for C/C++ in class 4 memory (low address space)

SYSSSC.CRTE.101.PARTIAL

CRTEPART for C/C++ PARTIAL in class 4 memory (high address space)

SYSSSC.CRTE.101.PARTIAL.LOW

CRTEPART for C/C++ PARTIAL in class 4 memory (low address space)

SYSSSC.CRTE.101.COB-PART

COBPART for COBOL PARTIAL in class 4 memory (high address space)

SYSSSC.CRTE.101.COB-PART.LOW

COBPART for COBOL PARTIAL in class 4 memory (low address space)

SYSSSC.CRTE.101.COBOL

CRTECOB for COBOL in class 4 memory (high address space)

SYSSSC.CRTE.101.COBOL.LOW

CRTECOB for COBOL in class 4 memory (low address space)

SYSSSC.CRTE.101.SIS

CRTESIS for COBOL in class 4 memory (high address space)

SYSSSC.CRTE.101.SIS.LOW

CRTESIS for COBOL in class 4 memory (low address space)

In BS2000 the program SSCM is provided for generating the subsystem catalog (see the manual “Subsystem Management (DSSM/SSCM)”.

In class 4 memory, the subsystems can be loaded to either the low or high address space (i.e. above or below 16 Mb).

By default, all the subsystems are loaded in the higher 16 Mb of class 4 memory. The files with the suffix LOW permit preloading below 16 Mb.

Activating and deactivating the subsystems

The subsystem names and versions for activating or deactivating subsystems are shown in the following table:

Subsystem	Name	Version
C	CRTEC	10.1
C-PARTIAL[1)] (/390)	CRTEPART	10.1
C-PARTIAL[1)] (X86)	CRTEPARK	10.1
COBOL85/COBOL2000 PARTIAL[1)] (/390)	COBPART	10.1
COBOL85/COBOL2000	CRTECOB	10.1
Shareable internal components	CRTESIS	10.1
Shareable internal components (X86)	CRTESIK	10.1

[1)]The ... PARTIAL subsystems apply in the same way for standard and complete partial bind

2.4 Runtime products required for program linkage

Since the CRTE versions are upwardly compatible, at least the highest of each required CRTE versions must be used for program linkage if the individual programs presuppose different CRTE versions.

For details on the required CRTE version please refer to the release notes of the relevant compilers.

When linking with programs written in other languages (ASSEMBH V1.2 and higher, FOR1 V2.2, Pascal-XT V2.2 and higher, PLI1 V4.2 and higher, RPG3 V4.0 and higher), the runtime system for the languages involved must also be linked.

3 Language-specific components of CRTE

Alongside the shareable CRTE components there are also the following language-specific components:

- COBOL85 and COBOL2000 runtime system
- C runtime system
- C++ runtime system
- C++ runtime system (Cfront)

3.1 CRTE libraries for the partial bind linkage method

There are two variants of the partial bind linkage method:

- Standard partial bind
- Complete partial bind

You will require different libraries depending on the partial bind variant that you use on/390 systems:

- Partial bind on /390 systems:
  - SYSLNK.CRTE.PARTIAL-BIND for standard partial bind
  - SYSLNK.CRTE.COMPL for complete partial bind

> The CRTE libraries for the partial bind linkage method are **only allowed for linking**. However, you may **not use** these libraries as the BSLIB **for loading** programs containing unresolved external references in the event of dynamic binding with DBL (see "Dynamic linking with DBL"), especially not when the CRTE subsystem is preloaded. In this case you may only use the SYSLNK.CRTE library as a BLSLIB.
>
> In the case of COBOL and mixed C/COBOL programs the complete partial bind linkage method can only be used if the objects run under POSIX and shared objects are loaded.

For more information on the partial bind linkage method in general and the specific characteristics of the two partial bind variants, see section “Dynamic loading of the C/COBOLruntime system and the internal routines (partial bind)”.

3.1.1 Libraries for the partial bind linkage method on /390 systems

The libraries SYSLNK.CRTE.PARTIAL-BIND and SYSLNK.CRTE.COMPL make it possible to link C programs, C /COBOL merges and COBOL programs in a way that satisfies the following conditions:

- The C programs, C/COBOL merges or COBOL programs contain no further open external references.
- The C/COBOL runtime system or the common internal routines are not dynamically loaded until runtime.

To this end, the SYSLNK.CRTE.PARTIAL-BIND and SYSLNK.CRTE.COMPL libraries contain linkage modules which are linked in place of the C and COBOL runtime modules or common internal runtime routines and resolve all the open external references in the program to preloadable components of the C / COBOL runtime system or the common internal runtime routines.

The libraries contain the modules which are needed in order to link C and COBOL programs which contain no open external references to CRTE. These include:

- non-preloadable modules of the C runtime system, name adapter modules, ILCS modules, and
- non-preloadable COBOL85/COBOL2000 runtime modules.

The C / COBOL runtime system itself or the actual internal runtime routines are available as dynamically loadable main modules in the libraries SYSLNK.CRTE and as part of the subsystems CRTEPART/COBPART or CRTESIS.

If you have installed CRTE privately and have modified the linkage modules with the ICXPPART procedure, the required modules are loaded dynamically from the “private” library regardless of preloaded subsystems and IMON accesses. This also applies when the programs linked with these linkage modules are used on a different system.

If you are using the official CRTE installation, an initial attempt is made at runtime to dynamically load the main modules from class 4 memory.
If the subsystem CRTEPART / COBPART or CRTESIS has not been started, the `get-path-name` functionality of IMON is used to determine where the library
SYSLNK.CRTE has been stored and the subsystem is subsequently loaded dynamically from this library.
If neither the subsystem nor IMON is available, the library is searched for under the standard user ID $ or under the user ID specified when ICXPART was called.

As neither the entire C / COBOL runtime system nor the common internal runtime routines are linked but only the linkage module (and certain modules that do not belong to the preloadable runtime system), the fully linked program or module needs considerably less disk space than when the C / COBOL runtime modules and the common internal runtime routines are statically linked from the SYSLNK.CRTE library. In addition, the link times and, if the required subsystem or subsystems are preloaded, the load times are also reduced.

3.2 COBOL85/ COBOL2000 runtime system

The COBOL85/COBOL2000 runtime system comprises runtime modules that are used when linking a COBOL85 /COBOL2000 object to an executable program.

The COBOL runtime modules can be recognized by their prefix “ITC”. They are available as elements in the following libraries provided that they were not preloaded with the subsystem:

- Library for static linkage:
  - SYSLNK.CRTE (/390 systems)

    The COBOL runtime modules are present in object module format (type R).

  When static linking is performed with CRTE , all unresolved external references are resolved.
- Library for linking with the partial bind linkage method
  - SYSLNK.CRTE.PARTIAL-BIND and SYSLNK.CRTE.COMPL (/390 systems)

  These libraries support the partial bind linkage method which significantly reduces the size of the final linked COBOL programs. In this case, all unresolved references are resolved by linkage modules. The COBOL85 /COBOL2000 runtime system is not dynamically loaded until execution time (see section “Libraries for the partial bindlinkage method on /390 systems” f).

  The programs linked using the partial bind linkage method use the subsystem COBPART (linkage with SYSLNK. CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL).

  > i In the case of COBOL and mixed C/COBOL programs the complete partial bind linkage method (library SYSLNK.CRTE.COMPL) can only be used if the objects run under POSIX and shared objects are loaded.

For more detailed information and examples of the various linkage methods, refer to chapter “Linkage of CRTE”.

Most of the shareable runtime modules are prelinked in a main module named ITCSR85 in the library SYSLNK. CRTE.SHARE, which can be loaded into class 4 memory by the system administrator using the CRTECOB subsystem (see section “Shareability of CRTE”).

The runtime modules represent subprograms known to the COBOL85 or COBOL2000 compiler. They can be divided into two broad groups:

1. Subprograms for complex COBOL statements
2. Subprograms for interfacing the generated module to operating system functions

   The main purpose of such subprograms is to ensure that code generation of the compiler is completely independent of the operating system.
   Important functions in this context are:
   - interface of COBOL programs to the I/O system
   - interface of COBOL programs to SORT
   - interface of COBOL programs to UDS
   - interface of COBOL programs to executive functions

A list with the names and functions of all COBOL85 and COBOL2000 runtime modules is provided in the COBOL User Guides.

Support for large files (> 2 GB) in COBOL programs

COBOL programs compiled with COBOL85 V2.3A or COBOL2000 and linked with CRTE V10.1A are able to process large BS2000 files (> 2 GB).

The maximal size of a BS2000 file is 4096 Gigabyte (= 4 Terabyte), a POSIX file may be maximal 1024 Gigabyte (= 1 Terabyte) in size.

3.3 C runtime system

The C runtime system is required for compiling and linking C and C++ programs. C++ programs that use the C++ library functions for complex math and/or standard input/output also require the include elements and modules from the library.

The C runtime system essentially consists of the following components:

- Standard include elements for the C library functions
- Source programs for the generation of user-specific locales
- C runtime modules

  The runtime modules contain, among other things, the code for all C library functions, central routines which also permit the implementation of C++ library functions and further routines for implementing the operating system interfaces of C and C++ programs.

  The module and entry names of the C runtime system begin with “IC@”, “ICS”, “ICX” or “ICP”.

  The C runtime modules are available in a number of versions, which means that they can be fully linked (dynamically or statically), dynamically loaded at runtime and preloaded as shareable in class 4 memory. Not all modules of the C runtime system are designed as shareable and capable of being preloaded. This applies, for example, to the name adapter modules (see "SYSLNK.CRTE library" in section “Libraries of the C runtime system").

These and other components of the C runtime system are stored as library elements in various CRTE libraries. The list below presents the CRTE libraries and associated contents.

3.3.1 Libraries of the C runtime system

This section describes the following libraries:

- SYSLNK.CRTE library
- SYSLNK.CRTE.SHARE library
- SYSLNK.CRTE.POSIX library
- SYSLNK.CRTE.COMPV1 and SYSLNK.CRTE.COMPV2 libraries
- SYSLNK.CRTE.TIMESHIFT library
- SYSLNK.CRTE.TIME50 library
- SYSLIB.CRTE library

SYSLNK.CRTE library

This library contains:

- Source programs for the generation of user-specific locales (type S)

USLOCA	Assembler source program
USLOCC	C source program

- Individual modules of the C runtime system (object modules, type R and link-and-load modules, type L)

  These modules can be statically linked using BINDER or dynamically linked using the DBL. The C runtime system can also be dynamically loaded (partial bind linkage method). In this case, one of the following libraries must be specified at link time in place of the library SYSLNK.CRTE.

  - SYSLNK.CRTE.PARTIAL-BIND (standard partial bind)
  - SYSLNK.CRTE.COMPL (complete partial bind)
- Name adapter modules for new C library functions:
  - Object modules (OMs, type R)
  - Link-and-load modules (LLMs, type L)

  The adapter modules are non-preloadable components of the C runtime system and must consequently be linked to the application program.
- C runtime system as a dynamically loadable main module (object module, type R) and link-and-load module (LLM, type L)

  This main module is dynamically loaded at runtime if the library SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL is included instead of the library SYSLNK.CRTE and the dynamically loadable C runtime system has not been preloaded as a CRTEPART subsystem into class 4 memory.

SYSLNK.CRTE.SHARE library

This library contains the C runtime system as the shareable main module IC@RTSXS. The system administrator can load this main module into class 4 memory as a CRTEC subsystem.

SYSLNK.CRTE.POSIX library

This “link switch” library must always be included if the POSIX functions of the C runtime system are used. It contains the modules ICSS44Y@ and ICSTISP@ (object module, type R) which must, **as a priority**, be linked before the modules in the library SYSLNK.CRTE or SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL. When linking is performed with BINDER it is necessary to include the library by means of an INCLUDE-MODULES statement (without specifying a module name), because the link sequence must be observed when linking is performed by the Autolink mechanism (RESOLVEs) and the content of the libraries may also change. In the case of dynamic linking using the DBL, the link switch library must be assigned a lower link name BLSLIB*nn* than the CRTE libraries which are included after it. The POSIX link switch is automatically included when programs developed in the POSIX shell use the command cc, c89 or CC.

SYSLNK.CRTE.COMPV1 and SYSLNK.CRTE.COMPV2 libraries

Objects generated by the C compiler V1.0 are executable with CRTE if they are relinked with the library SYSLNK. CRTE or SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL and with the library SYSLNK.CRTE.COMPV1.

C modules or C programs for which the linkage module has been linked from CLIB V2.0 require the dynamically loadable C runtime system compatible with C V2.0 in order to run. After installation of CRTE V10.1A, this runtime system is contained in the $.CLIB library. This means that it is no longer necessary (cf. CRTE up to and including V1.0B) to assign the library SYSLNK.CRTE.COMPV2 with the link name IC@CLIB before calling a C V2.0 program. Procedures containing this assignment need not be changed.

SYSLNK.CRTE.TIMESHIFT library

This “link switch” library is supported fo compatibilty reasons only.

In previous versions it was used to shift the reference date (epoch) for the time functions `ctime`, `difftime`, `ftime` , `gmtime`, `localtime`, `mktime` and `time` from 1/1/1950 to 1/1/1970 00:00:00.

In the current CRTE version the reference date by default is 1/1/1970 00:00:00. Therefor it using this link switch is not necessary anymore.

SYSLNK.CRTE.TIME50 library

This “link switch” library shifts the reference date (epoch) for the time functions `ctime`, `dif-ftime`, `ftime`, `gmtime`, `localtime`, `mktime` and `time` to 1/1/1950 00:00:00. Thus the behaviour of the mentioned time funktions can be set to the default behavior of CRTE versions up to V2.9 and in V10.0A and V11.0A.

This link switch is necessary if time stamps are to be processed which have been created with 1/1/1950 as the reference date (epoch).

However, the time functions mentioned above don‘t supply correct result anymore for time stamps after 1/19/2038 03:14:07. Especially, a call of the functions `time` and `ftime` cause the termination of the program, as the current date is higher than 1/19/2018.

To use the bind option, enter the following statement when binding:

```
INCLUDE-MODULE LIB=<library>,ELEMENT=*ALL
```

SYSLIB.CRTE library

The source components of the C runtime system (standard include elements and source programs for user-defined locales) are supplied in the library SYSLIB.CRTE. This library also contains the ILCS macros.

3.3.2 Support for large files > 2 GB by 64-bit functions

C/C++ programs are able to process large files (> 2 GB) if they have been compiled with a C/C++ compiler supporting the long long data type. The long long data type is part of any C/C++ compiler >= V3.1.

The maximal size of a BS2000 file is 4096 Gigabyte (= 4 Terabyte), a POSIX file may be maximal 1024 Gigabyte (= 1 Terabyte) in size.

3.4 Cfront-C++-library functions and runtime system

The Cfront-C++ library functions for complex mathematics and standard I/O operations are available in the libraries SYSLNK.CRTE.CFCPP and SYSLNK.CRTE.CPP:

- SYSLNK.CRTE.CFCPP contains the runtime system for Cfront programs as of C++ V3.0B.
- The runtime system for C++ programs < V3.0 is present in SYSLNK.CRTE.CPP.

3.5 ANSI-C++ libraries and runtime systems

The following libraries are provided for the C/C++ compiler V3.0 or higher:

- For use in ANSI-C++ compiler mode, the standard C++ library (SYSLNK.CRTE.STDCPP), the C++ runtime system (SYSLNK.CRTE.RTSCPP) and the Tools.h++ library (SYSLNK.CRTE.TOOLS).
- The library SYSLNK.CRTE.CPP-COMPL exists for the complete partial bind linkage method. This contains all the modules of the libraries listed above. In each case, all the modules in a library are grouped together to form an LLM main module.
- For use in Cfront-C++ compiler mode, the Cfront-C++ runtime system (SYSLNK.CRTE.CFCPP), in addition to the C++ library functions for complex math and stream-oriented I/O (see preceding section).

The use of these libraries is described in detail in the “C/C++ User Guide” [9 (Related publications)].

3.6 Common internal runtime routines

CRTE contains common routines that can be used internally by both the COBOL85/COBOL2000 and the C/C++ runtime system. With the exception of memory management and the mathematical routines, the modules for these internal routines are located in the CRTESIS subsystem. Memory management is part of the subsystems CRTEC and CRTEPART (for information on subsystems, see "Shareability of CRTE").

The modules can be recognized by the following prefixes:

IML...	Mathematical routines
IT0...	ILCS routines (see chapter “ILCS program communication interface”)
IT1...	Service routines, e.g. for data and memory management, for messages and for garbage collection for COBOL2000 programs

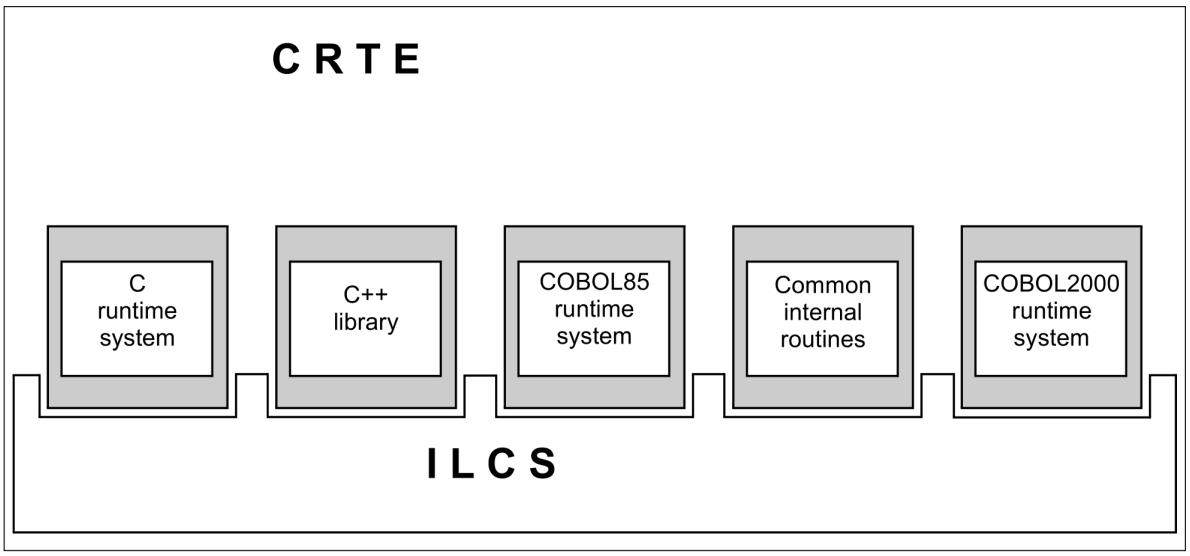
By using SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL, programs which use common internal runtime routines can be linked without external references. The runtime routines themselves are then dynamically loaded during program execution.

4 ILCS program communication interface

The ILCS (**I**nter **L**anguage **C**ommunication **S**ervices) program communication interface is the basis for implementing the Common RunTime Environment CRTE. It standardizes and simplifies, across the various programming languages, important communication functions between the programs of a run unit and between run unit and operating system.

4.1 Architecture and functionality of ILCS

A range of ICLS basic functions can also be used directly by the language runtime systems and the applications.



ILCS is a combination of software and interface conventions:

- ILCS comprises runtime routines which are grouped together in the CRTE library
- It also defines the communication interface corresponding to the “standard linkage conventions in BS2000”, i.e. any module generated by an ILCS-compatible compiler is ready for linking with programs in the same or a different language in accordance with the standard linkage conventions.

The ILCS routines are part of the CRTE selectable unit and the latest version of the routines is always shipped with CRTE. They can be recognized by the prefix “IT0”

ILCS provides the following functions:

- program initialization
- program termination
- convention for linking programs in different languages
- standard guidelines for event handling
- memory management (stack and heap memory)
- uniform handling of program mask
- processing of non-local branches
- support for C++ exception handling in C/C++ V3.0 or higher

All recent versions of BS2000 compilers, as listed below, are ILCS-compatible.

Compiler	as of Version
Ada	2.1
ASSEMBH	1.1
C	2.0
C++	2.1A
COBOL85	1.1

COBOL2000	1.0
FOR1	2.2
Pascal-XT	2.2
PLI1	4.1
RPG3	3.0
UX-Basic	3.0B

In addition, openUTM is ILCS-compatible as of Version 3.3.

With the aid of ILCS, it is a simple matter to link programs compiled with ILCS-compatible compilers to form a program system. If a program system contains programs that do not behave as required by ILCS conventions, they must be modified to comply with the conventions. Otherwise incompatibility may be the result, particularly when linking programs in different languages.

The ILCS comprises

- the code module IT0SL@ and
- the data module IT0SL#.

ILCS (CRTECOM) is no longer supplied as a preloadable subsystem as of CRTE V2.1. When linking applications which require CRTE you must make sure that ILCS is always linked from CRTE. In particular, no old IT0INITS modules from the language runtime system libraries and no modules from $.SYSLNK. ILCS may be dynamically linked. For this reason, when resolving external references, the $.SYSLNK. CRTE library should be searched before any other of the language runtime system libraries. In addition, it must be noted that ILCS and runtime system modules may only be linked into an application once. This does not apply for encapsulated subsystems.

4.2 ILCS conventions

This section provides information about the following topics:

- Register conventions
- Data structures
- Program mask handling by ILCS

4.2.1 Register conventions

Register contents on program call

The following table provides an overview of register contents as supplied by the calling program (before passing control to the called program).

Register number	Contents
0	Number of parameters
1	Start address of the parameter address list
2 - 12	Undefined
13	Start address of the save area of the calling program
14	Address of the return point to the calling program
15	Address of the entry point in the called program
PM	Program mask: value of PCD field “program mask” (default X'0C')

Register contents on return to the calling program

The following table provides an overview of register contents as supplied by the called program when returning control to the calling program.

Register number	Contents
0 and 1	Return values of integer functions (or undefined)
Floating-point register 0	Return values of floating-point functions (or undefined)
2 - 14	As in table above
15	Undefined
PM	Program mask: value from PCD field “program mask”

4.2.2 Data structures

Save area

The calling program supplies (in register 13) the address of a save area in which the called program can store the current contents of the registers.
The called program creates a new save area and chains the two areas together.
The save area is structured as follows:

Bytes	Contents
1-4	1st byte: 1st bit: activity bit (1: program active, 0: program inactive) 2nd - 8th bit: reserved 2nd byte: version = X' 01' 3rd and 4th bytes: X' FEFF'
5-8	Start address of the save area of the calling program. The contents of the field in the **first** calling program are -1.
9-12	Start address of the next save area (if any).
13-16	Contents of register 14
17-20	Contents of register 15
21-24	Contents of register 0
25-28	Contents of register 1
29-32	Contents of register 2
. .	
69-72	Contents of register 12
73-76	Reserved for FOR1
77-80	Address of the PCD
81-84	Address of the EHL (Event Handler List).If no EHL is defined, the field contains -1.
85-128	Reserved

Prosys Common Data Area (PCD)

The PCD is a common data area used by all programming languages. The first part contains the data areas used by ILCS, including, for example, the “program mask” field which is initialized to the value X’0C’. The second part of the PCD contains the programming language areas, each of 128 bytes, that are available to the runtime systems of the various languages.

4.2.3 Program mask handling by ILCS

The program mask for program execution is set to the value of the PCD field “program mask” (default X’0C’: fixed-point and decimal overflow abort the program, exponent underflow and mantissa zero do not) at initialization. If the mask is changed during the program run, it must be reset to the value of the PCD field “program mask” before return of program control or before the next program call.
In C/C++ the program mask can be changed with the aid of the RUNTIME-OPTIONS option (see the C and C++ User Guides); in Assembler it can be changed with the aid of the ASSEMBH structure macro @SETPM (see the ASSEMBH Reference Manual [12 (Related publications)]).

4.3 Initialization

This section provides information about the following topics:

- Initialization of ILCS
- Initialization on dynamic program loading

4.3.1 Initialization of ILCS

Initialization of ILCS is normally automatic.
If ILCS has not been initialized for some reason, it can be initialized before branching to the first ILCS module by calling the initialization routine **IT0INITS** in an Assembler program (see the example in the appendix).
IT0INITS is called without parameters. In addition, register 0 must be supplied the value 0 (number of parameters = 0), since the routine can have up to 9 parameters for internal purposes. IT0INITS supplies the address of the PCD in register 0.
Register 15 contains the following return codes:

Return code	Meaning
0	Initialization of ILCS completed successfully
1	IT0INITS has already been called
2	BS2000 version is not supported
3	Version incompatibility between code and data
4	Shortage of memory for initialization of stack administration
5	Shortage of memory for initialization of heap administration
6	Standard event handler could not be initialized
7	A language initialization routine returned an error

4.3.2 Initialization on dynamic program loading

On program startup, ILCS performs the necessary initializations for all languages involved in the program system. The involvement of a language is only detected if the modules of a language are linked statically or if they are recognized by the dynamic linking loader (DLL).

If, in Assembler programs, program sections containing objects in languages not yet called are loaded by means of the BIND macro, the necessary initializations will not have been carried out for these languages.
In such instances the ILCS routine IT0ININ must be called in a user routine after dynamic loading in an Assembler routine. The ASSEMBH structure macro @ININ is available for calling IT0ININ (see the example in the appendix and the “ASSEMBH” Reference Manual [12 (Related publications)]). ILCS must already have been initialized when IT0ININ is called.

When dynamically loading subprograms in COBOL85/COBOL2000 programs with “CALL identifier”, the COBOL85 /COBOL2000 runtime system automatically calls IT0ININ internally.

Special aspect of C++

Even if C++ modules are already present, the ILCS routine IT0ININ must be called after each dynamic loading of a C++ modules. This is necessary to ensure that calls for global constructors are made.

4.4 Encapsulated subsystems

A subsystem which works with its own global data can encapsulate itself.

This has the following advantages:

- The subsystem is independent of the various versions of a component which may be present both in the environment and in the subsystem.
- The internal statuses of the language runtime systems of the application and the subsystem do not interfere with one another.

At the same time, encapsulation imposes the following restrictions:

- Global jumps (e.g. longjump in C) from the subsystem to the application are not possible.
- Events from the subsystem cannot be propagated to the application.

4.4.1 Requirements placed on an encapsulated subsystem

The characteristics listed above mean that an encapsulated subsystem must meet the following requirements:

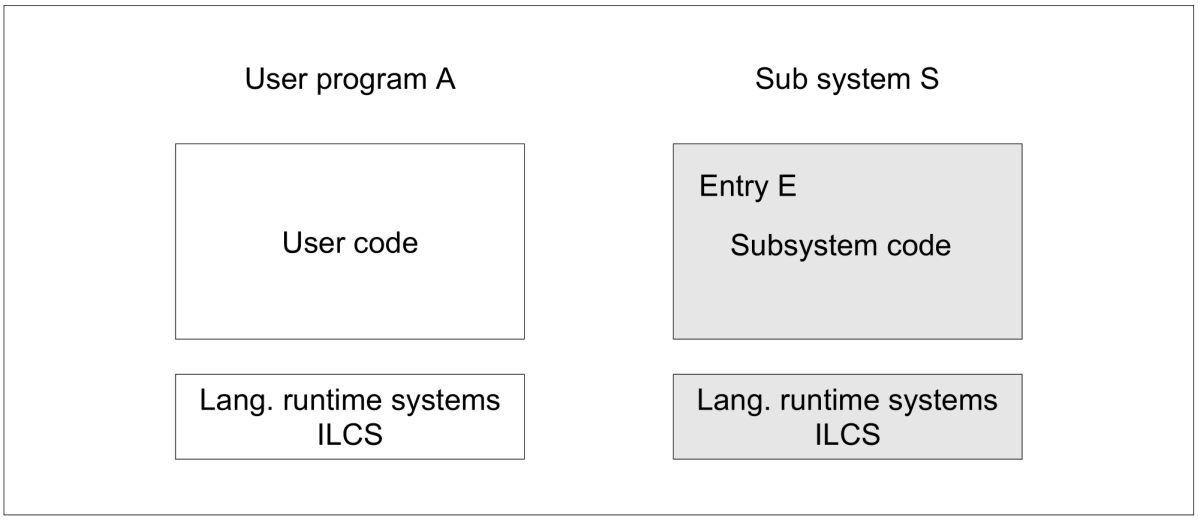
- The subsystem needs “private” copies of the runtime systems used and the ILCS. The entries of these components must no longer be visible to the outside.
- The subsystem must include a switch which controls the initialization of the subsystem the first time it is called.
- The subsystem needs variables for the addresses of the save areas of the application and the first save area of the subsystem. The PSA and EHL fields in the first save area of the subsystem must contain the value F‘-1‘.

In addition, certain considerations must be borne in mind when linking subsystems, switching to the subsystem environment and performing event and contingency handling. These are described in the following sections.

4.4.2 Architecture of an encapsulated subsystem

A subsystem is prelinked with all the required components. If the subsystem is to be preloadable, the code and data parts must be linked separately.

Below is a diagrammatic representation of an encapsulated subsystem



4.4.3 Actions required for encapsulation

A called subsystem function must carry out the following actions

1. Save the registers of the caller in the save area of the caller and the address of this save area in the appropriate variable.
2. The first time a subsystem is called (switch set), call IT0INITS to initialize the ILCS and call the language runtime system and enter the PCD address returned by IT0INITS in the save area of the subsystem (then deactivate the switch).
3. Deactivate SEH event handling of the subsystem with IT0SEHEN if the subsystem does not perform any explicit event handling.
   Alternatively, the event handling of the called program can be deactivated with IT0CRDEA and the ILCS of the subsystem activated with IT0CRACT if the switch is not set.
4. Load the address of the first save area of the subsystem in register 13. This switches to the ILCS of the subsystem, and the code of the called function can be executed.

Before returning to the application, the following actions must be carried out:

1. If the event handling of the subsystem was activated with IT0CACT, it should be deactivated with T0CDEA before the subsystem is called again provided that no asynchronous, instance-specific events are to be delivered to the subsystem while it is inactive.
   The ILCS of the calling program should be reactivated (IT0CACT).
2. The registers of the application must be loaded, and the address of the save area of the application must be loaded in register 13.
3. The results of the called subsystem function must be passed on to the application.

4.4.4 Using POSIX in subsystems

POSIX interfaces can only be used by *one* subsystem or by the application since anything else would result in reciprocal influence on the standard streams.

4.4.5 STXIT events and contingencies

The forwarding of events to the various ILCS instances is controlled by activating or deactivating the appropriate PCDs.
A distinction is made here between instance-specific and task-specific events and contingencies.

If no precautionary measures are taken, the subsystem is informed of program errors (STXIT class PROCHK and ERROR) even if the error did not occur in the subsystem.
If the application contains several components with private ILCS instances, behavior in the event of an error is influenced by the order of the SEH logons to the operating system, something which is normally not desired.
If the subsystem may ignore all events, the SEH should be deactivated with IT0SEHEN. This does however have the disadvantage of also deactivating the error handling of the language runtime system.

4.4.6 Unloading a subsystem

A subsystem can be terminated and subsequently unloaded without having to terminate the entire application. This is done by calling IT0CRTRM.

4.5 Linking ILCS programs with different languages

This section provides information about the following topics:

- Parameter passing
- Passing function return values
- Event handling by ILCS

4.5.1 Parameter passing

Data type semantics vary significantly in the programming languages that can be linked by means of ILCS. The data types that have the same data representation in the individual programming languages (and can therefore be passed as parameters without problems) are listed below. When other data types are used as parameters, a detailed knowledge of the type of data storage is required in order to ensure correct program execution.

The parameters are passed “by reference”, i.e. the address of the data item is passed.

Compiler	**Data types**			
	Binary word	Floating-point word	Floating-point doubleword	String
C / C++	long	float	double	char <var> [<size>]
COBOL2000 COBOL85	PIC S9(i) COMP-5 SYNC 5 <= i <= 9	COMP-1	COMP-2	USAGE DISPLAY
Ada	INTEGER		FLOAT	EBCDIC_STRING (<static_INDEX_ CONSTRAINT>)
ASSEMBH (@ macros)	F	E	D	C
FOR1	INTEGER*4	REAL*4	REAL*8	CHARACTER*i
Pascal-XT	long_integer	short_real	long_real	packed array [<range>] of char
PLI1	BIN FIXED(31) ALIGNED	BIN FLOAT(21) DEC FLOAT(6)	BIN FLOAT(53) DEC FLOAT(16)	CHAR(i)
RPG3	binary item 0 decimal points			alphanum. item, fixed length
UX-Basic	%%,?	!	!!	$

Compatible data types for parameter passing

The calling program compiles a list of passed addresses. The number of parameters is passed in register 0, while the address of the list is passed in register 1 (see section “Register conventions”).

Parameter passing in Assembler

The data must always be aligned when stored, i.e. 32-bit integers represented in binary form are aligned on word boundaries, floating-point numbers are aligned on word or doubleword boundaries, and strings are aligned on byte boundaries. The length of strings is constant and is known to the called program.

Parameter passing in C and C++

In C/C++ parameters are passed “by value” as standard in compliance with language definition. This type of parameter passing is possible without restrictions only in program systems in which only C and C++ are involved.

If objects in other languages are linked, the following steps must be taken:

- When the non-C or non-C++ routine is called, the address of the data element to be passed must be given in C /C++, for example &par. Technically speaking, the address of the data element is then passed “by value”.
- If a C/C++ routine is called from a non-C/C++ routine, the formal parameters in C/C++ must be declared as pointers to the data elements to be passed, e.g. f (T *par).

Parameter passing in COBOL

In COBOL85 and COBOL2000, parameters are passed as standard “by reference” (see the table "Compatible data types for parameter passing").

Parameter passing “by value”

COBOL2000 can pass any numeric parameters “by value” if a prototype exists for the called program. If no prototype exists, parameter passing “by value” is only possible with restrictions (see next page).

The position and alignment of the data in the parameter list are dependent on the data description in the prototype. The following tables indicate the effects of the data descriptions:

Binary data PIC[S]9(i) {USAGE COMP | COMP-5 | BINARY}	**Representation in parameter list**
Halfword (1 <= i <= 4)	right-justified in aligned whole word (bytes 1 and 2 undefined)
Whole word (5 <= i <= 9)	aligned whole word
Doubleword (10 <=i <= 18)	aligned doubleword (bytes inserted for alignment are undefined)

Floating point data	**Representation in parameter list**
Word (USAGE COMP-1)	aligned whole word
Doubleword (USAGE COMP-2)	aligned doubleword (bytes inserted for alignment are undefined)

Numeric data that is packed (USAGE COMP-3/PACKED DECIMAL) or printable (USAGE DISPLAY) should only be passed between COBOL2000 programs “by value”. The data is stored in the parameter list flush left on word boundary (length <8 bytes) or doubleword boundary (length 8 bytes).

If no prototype exists, only data items with the following definitions can be passed “by value” with COBOL2000:

- COMP-5 PIC[S]9(i) (1 <= i <= 9) or
- 1-byte data item of any type

If necessary, the parameter is expanded to 4 bytes by padding with binary zeros flush left.

This restricted parameter passing “by value” is also possible with COBOL85 Version 2.1B or later.

Passing OMITTED (only possible with COBOL2000)

Parameters that are omitted when a subroutine is called (specification of OMITTED in the call or OPTIONAL in the prototype) are stored as an aligned whole word with the value X’FFFFFFFF’ in the parameter list.

In an OMITTED test in a called COBOL2000 program, each parameter that appears in the parameter list with the address X’FFFFFFFF’ is evaluated as not passed (OMITTED).

Parameter passing in Java

Although it is possible to transfer data between C and Java programs, this cannot be done via the ILCS interface. For information on transferring data between C/C++ and Java please refer to your Java documentation.

4.5.2 Passing function return values

Return values of integer functions are passed in registers 0 and 1, return values of floatingpoint functions in floating-point register 0.
It is possible to pass return values with other data types in registers 0 and 1, but this is not defined by ILCS. Their representation is left to the individual programming languages.COBOL subprograms as of COBOL85 V2.0 and COBOL2000 V1.0 always behave like integer functions and return the contents of the COBOL special register RETURN-CODE to the calling program in registers 0 and 1.

As of COBOL85 Version 2.1B and COBOL2000 V1.0, the return value of a called C subprogram can be accessed via the COBOL special register RETURN-CODE by means of a control statement (see the “COBOL85 or COBOL2000 User Guide” [2 (Related publications), 4 (Related publications)]).

4.5.3 Event handling by ILCS

ILCS differentiates between implicit language-specific event handling and event handling that can be explicitly enabled and disabled. The ILCS interface SEH (Standard Event Handler) is available for implicit event handling, and SSH (Standard STXIT Handler) is available for explicit event handling.

Standard event handler (SEH)

Event handling using the SEH interface is possible only for the STXIT events ERROR and PROCHK. This interface is used internally by the runtime system either to implement the language resources to be used for exception handling in the various programming languages (e.g. Pascal, Pl/1, Ada) or to take certain measures - such as the output of error messages - in response to an ERROR or PROCHK event before the program aborts (e.g. COBOL85).

Standard STXIT handler (SSH)

The SSH interface can be used to explicitly enable and disable user-defined routines for all BS2000-STXIT events. This interface is currently used by the C/C++ and ASSEMBH runtime systems: C/C++ programs can use the C library functions signal and cstxit to enable user-defined routines for event handling; Assembler programs do this with the ASSEMBH structure macros @STXEN and @STXDI.

The explicit enabling of user-defined routines for the STXIT event classes ERROR and PROCHK (SSH) disables the implicit handling of these events by other languages (SEH). If implicit event handling of other languages (COBOL85, Pascal-XT, etc.) is to be reactivated, the event handling routines for ERROR and PROCHK must be explicitly disabled when leaving the C or ASSEMBH language environment. In C/C++ the routines are disabled by allocating SIG_DFL in the C library function signal, in Assembler it is done with the aid of the ASSEMBH structure macro @STXDI.

STXIT routines directly enabled for BS2000

In Assembler programs, use of the BS2000 macro STXIT makes it possible to enable STXIT routines directly for BS2000. However, these routines cannot be coordinated by ILCS. For example, if STXIT routines are enabled directly for PROCHK and ERROR events, the implicit handling of these events via the SEH interface remains active.

4.6 Linking ILCS and non-ILCS programs

This section provides information about the following topics:

- Linking ILCS programs and non-ILCS programs of the same language
- Linking ILCS programs to non-ILCS programs of a different language

4.6.1 Linking ILCS programs and non-ILCS programs of the same language

There are different rules for linking ILCS programs and same-language non-ILCS programs (i.e. programs not generated by an ILCS-compatible compiler); these rules depend on the language (compiler and associated runtime system). In the following, the rules to be observed when linking ILCS and non-ILCS programs are described only for the CRTE languages COBOL and C and also for Assembler.
The rules for other languages (e.g. Fortran, Pascal) are described in the relevant user guides.

COBOL85

The linking of non-ILCS programs (i.e. programs created with COBOL85 V1.0) to ILCS programs is basically possible. Program communication via ILCS can only be ensured, however, if the main program is an ILCS program. If the main program is a non-ILCS program, all called ILCS subprograms are seen as non-ILCS programs.

COBOL2000

The linking of non-ILCS programs (i.e. programs created with COBOL85 V1.0) to ILCS programs is basically possible.
Since COBOL2000 always uses the ILCS infrastructure internally, ILCS is initialized when COBOL2000 modules are called, even if the main program is a non-ILCS program.

> ¡ Using COBOL programs with USE procedures requires a valid backwards concatenation in the save area. X'FFFFFFFF' or X'00000000' are interpreted as the end of the backwards concatenation.

C and C++

Non-ILCS programs (i.e. programs created with C V1.0) and ILCS programs in C can be linked to one another without difficulty. If the calling program is a C V1.0 program, the ILCS environment is initialized automatically by the C runtime system. Linking is possible up to C Version 2.0 without precautionary measures, but from C or C++ Version 2.1A onward it is possible only under the following conditions:
At compilation, the default setting INLINE-OPTIMAL of the LINKAGE parameter must be changed to OUT-OF-LINE or V1-COMPATIBLE-INLINE in the COMPILER-ACTION option.Compilation of a C program with the C++ compiler must be performed in KR or ANSI mode. Programs compiled in CPLUSPLUS mode cannot be linked to C V1.0 objects.

Assembler

For a non-ILCS program to be able to call an ILCS program, it is necessary to adapt the save area of the non-ILCS program to comply with the ILCS convention. This adaptation can be carried out by the user as follows:

- The save area of the non-ILCS object must be created with the IT0VSA macro.
- Before the ILCS object is called, the field &P.VPCD must be supplied with the address of the current IT0PCD module. This address can be ascertained from address constant A(IT0PCD).

For an example, see the appendix.

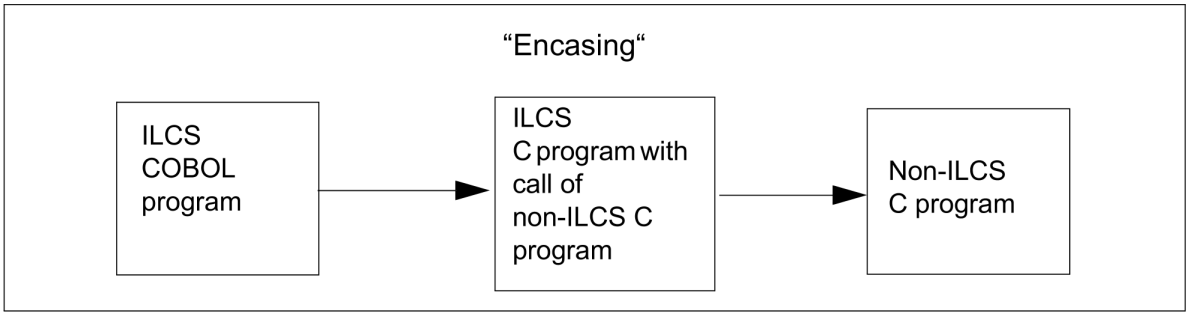
Java

There are various methods for linking C/C++ programs and Java programs, for example linkage via the Java Native Interface (JNI). For more information on this subject, refer to your Java documentation.

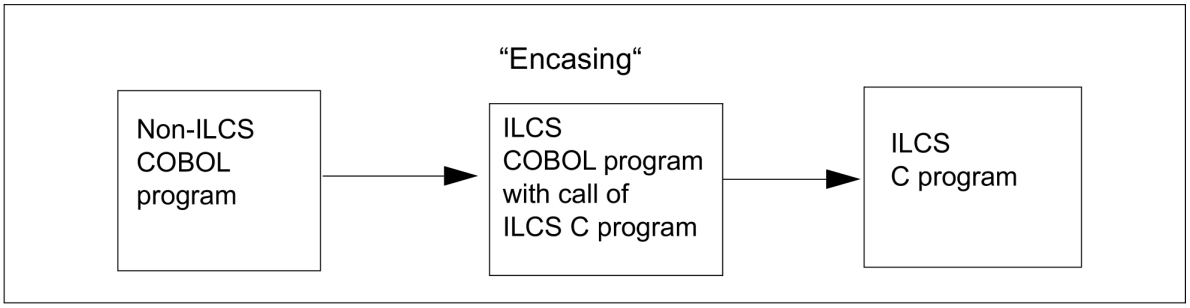
4.6.2 Linking ILCS programs to non-ILCS programs of a different language

Linking non-ILCS programs to ILCS programs in a different language is only possible if the non-ILCS programs are “encased” (or “encapsulated”) by ILCS programs of the same language, i.e. the user must write a program which calls the non-ILCS program and must compile this program by means of an ILCS-compatible compiler

Example: ILCS-COBOL program calls non-ILCS-C program.



Example: Non-ILCS COBOL program calls ILCS C program.



4.7 Error handling

ILCS normally reports the occurrence of an error to the calling program via a return code. In the following situations, however, this is not possible since there is no caller:

- An exception occurs in a contingency routine (e.g. no resources).
- A return to the co-routine adapter is effected from IT0CREND.

These situation are treated as *internal* errors in ILCS.

The following error situations are treated by the standard event handler (SEH) as internal errors for which normal SEH error recovery - triggering the error again - is not possible:

- Data exception. These are SEH exceptions that have changed the data in such a way that the exceptions cannot be triggered again.
- An SEH error recovery routine reports that an STXIT event is to be treated like a data exception.

In the event of internal errors:

- an error message is issued (prerequisite for issuing error messages is CRTE-MSG V1.3)
- debugging information is stored in the Prosys Common Data Area (PCD), and
- the application is terminated abnormally with the TERM macro.

Values are entered in the fields in the PCD the first time an error occurs which causes abnormal termination. Sequence errors are ignored.
The PCD fields are also supplied with values if abnormal termination is handled by the TERM macro by specifying a user exit.

The possible values of the PCD fields M7TYP, M7DIA and M7ADR are listed below.

The user can also control the TERM macro via the contents of certain PCD fields. You will find a description of how to initialize the corresponding fields in section “Control of the TERM macro by the user”.

4.7.1 Debugging information in the PCD

The following information is stored in the fields of the PCD (the entries refer to the first error which caused abnormal termination. Sequence errors are ignored):

1. In the field M7TYP, the error type:

Value	Message number	Meaning
0	CCM0200	Data exception
1	CCM0201	SEH signals a data exception
2	CCM0202	Return from the co-routine adapter
3	CCM0203	Insufficient memory SSH (Standard STXIT Handler)
4	CCM0204	Insufficient memory SCH (Standard Contingency Handler)
5	CCM0205	Internal system error (error when switching heap or stack, or something similar)
6	CCM0205	LEVCO delivers return code to SSH
7	CCM0205	LEVCO delivers return code to SCH

2. In M7DIA field: the STXIT interrupt weight or the return code if a SVC.
3. In M7ADR field: address in the ILCS at which the error occurred.

4.7.2 Control of the TERM macro by the user

When the TERM macro is called by the ILCS, the UNIT, DUMP, MODE and URETCD parameters are supplied with values which are located in the PCD fields AUNT, ADMP, AMDE and AMJV.

The table below indicates how the PCD fields AUNT, ADMP, AMDE and AMJV are initialized. You, as the user, can modify these PCD fields (by writing the desired value in the PCD field using an Assembler instruction) and thus influence the behavior of the TERM macro.

Contents of the PCD field	Corresponding parameter value for the TERM macro
AUNT = X‘02‘	UNIT = STEP
ADMP = X‘01‘	DUMP = Y
AMDE = X‘04‘	MODE = ABNORMAL
AMJV = CL4'3'	URETCD = CL4‘3‘

4.7.3 Message texts

CCM0200 DATA EXCEPTION: EW=‘(&00)‘

Meaning
A data exception that has not been handled has occurred such as, for example, division by zero, OVERFLOW, etc. UG indicates the interrupt weight.

Response
Correct the error in the program.

CCM0201 DATA EXCEPTION REPORTED BY SEH-Routine: EW=‘(&00)‘

Meaning
An SEH routine reports via a return code that the SEH event is to be handled like an untreated data exception.

Response
Correct the error in the program.

CCM0202 ILLEGAL RETURN FROM IT0CREND

Meaning
IT0CREND returns to the co-routine adapter.

Response
Please contact system support.

CCM0203 NOT ENOUGH MEMORY SPACE AVAILABLE ON STARTING A STXIT PROCESS

Meaning
Resource bottleneck.

Response
Check for nested interrupt calls or a call loop. If this is not the case, contact your system administrator so that you can be allocated more memory.

CCM0204 NOT ENOUGH MEMORY SPACE AVAILABLE ON STARTING A CONTINGENCY PROCESS

Meaning
Resource bottleneck.

Response
Check for nested interrupt calls or a call loop. If this is not the case, contact your system administrator so that you can be allocated more memory.

CCM0205 INTERNAL ILCS ERROR

Meaning
An internal error has occurred in the ILCS.

Response
Please contact systems support.

5 Linkage of CRTE

A source program that has been compiled using a compiler is present either in object module or link-and-load module format. Object modules (OMs) and link-and-load modules (LLMs) are input objects for the Binder Loader Starter system (BLS) that generates executable programs from these objects.

The link editor combines a compiled source program together with all the object modules and link-and-load modules required for program execution to create an executable unit. In doing so, in each module it modifies addresses that reference areas external to the module and which could not therefore be entered by the compiler. These addresses are known as external references.

In the sections below describe linking with /390 systems. Section “Input examples” contains the example inputs.

> ¡ Link-and-load modules (LLMs) with linked-in runtime systems **must not** be stored in libraries from which non-prelinked link-and-load modules can be loaded directly.

5.1 Linking with BINDER and DBL

The CRTE modules should always be linked using the Autolink mechanism, i.e. by means of RESOLVE-BY-AUTOLINK statements (BINDER), or by means of the AUTOLNK specification in the DBL's (Dynamic Binder Loader) BIND macro as the names of the CRTE modules are not guaranteed external interfaces and may change in future versions.

BINDER and DBL all belong to the **Binder Loader Starter (BLS)** system.

5.1.1 The BINDER linkage editor

The BINDER linkage editor combines modules to form a loadable unit. This unit is known as a link-and-load module (LLM). BINDER stores link-and-load modules as type L library elements in a library.

BINDER can use the following modules to create a link-and-load module:

- Object modules (OMs) and prelinked object modules (main modules) from a library or from the temporary EAM object module file
- Link-and-load modules (LLMs) that were created by the compiler or BINDER and come from the same library.

5.1.2 The Dynamic Binder Loader (DBL)

The Dynamic Binder Loader (DBL) has the task of combining modules to form a load unit which it then loads. However, linkage with DBL is only temporary as long as there is no executable program.

The DBL can create a load unit from the following modules:

- Link-and-load modules (LLMs) linked by BINDER or generated by a compiler and saved in a library (type L).
- Object modules (OMs) generated by a compiler and saved in a library (type R) or in the temporary EAM object module file.

5.1.3 Linking with the Autolink mechanism

When the Autolink mechanism is used, BINDER and DBL attempt to resolve all unresolved external references in the modules of a load unit.

The order in which the libraries to be searched are specified is of particular importance:
To ensure that the external references are satisfied from the correct modules, the **logical** order shown below must be observed when specifying the libraries:

1. user-specific libraries
2. the CRTE libraries SYSLNK.CRTE (or SYSLINK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL) and SYSLNK.CRTE.CPP plus SYSLNK.CRTE.CFCPP
3. language-specific runtime libraries of non-CRTE languages (in the case of a language mix)

- It should be noted that BINDER searches in the libraries in the exact order in which they are specified.
- SYSLNK.CRTE.COMPL does not support Cfront mode.

For detailed information on the use of the autolink mechanism, refer to the “BINDER” and “Binder Loader/Starter” manuals.

5.1.4 Linking with nested external references

The C++ runtime library and the Fortran90 runtime library contain external references to the C runtime library. This means that not all external references can be resolved in a single chronological search in the specified libraries, but libraries that have already been searched must be searched again.

> It should be noted that BINDER automatically repeats the search procedure only if all runtime libraries to be searched are specified as a list in a single RESOLVE-BY-AUTOLINK statement.

5.1.5 Using time functions

The time functions `ctime`, `difftime`, `ftime`, `gmtime`, `localtime`, `mktime` and `time` by default use 1/1/1970 00:00:00 as the reference date (epoch). As a result, the specified time functions supply correct results from 12/13 /1901 20:45:52 up to 1/19/2038 03:14:07. This is the behavior when using the link switch TIMESHIFT in CRTE versions since V2.9A (see also “SYSLNK.CRTE.TIMESHIFT library” in section "Libraries of the C runtime system").

If you require the default behaviour of CRTE versions up to V2.9 or of V10.0A or 11.0A for the mentioned time functions, you have to use the link switch TIME50 (see also “SYSLNK.CRTE.TIME50 library” in section “Libraries of the C runtime system").

5.1.6 Masking of symbols

There is no standard masking of symbols (CSECTs, ENTRYs) when linking with BINDER; the symbols remain visible for subsequent linkage runs with BINDER or DBL.
When linking with DBL, this has the following effects: If a library contains individual modules created by the compiler and also LLMs with a linked runtime system, external references to the runtime system may, in certain circumstances, be resolved from any prelinked module and not from the runtime library when dynamically linking individual modules. In this event, a number of “DUPLICATES” warnings are received from the DBL. This is due to the Autolink mechanism first searching the library containing the individual module and then the runtime libraries assigned with the link names BLSLIB*nn*.

In order to ensure that, when performing dynamic linkage, the external references are always resolved from the current runtime library and not from any arbitrary module, it is advisable

- either to keep individual modules and prelinked modules in different libraries or
- to mask the symbols, for example with the statement MODIFY-SYMBOL-VISIBILITY VISIBLE=NO when linking with BINDER.

5.1.7 Language-specific considerations

Special aspects of C and C++

Modules generated in one of the ways listed below exist only in LLM format. Program systems containing program sections of this kind must be linked with DBL in RUN-MODE=ADVANCED or with BINDER:

- C modules generated with C/C++ Compiler V3.0 or higher
- C++ modules

C++ modules generated with C/C++ Compiler >= V3.0 in ANSI-C++ mode must be linked with a specially provided compiler statement (BIND). This is partly due to the template instantiation. This manual does not describe linkage of ANSI-C++ modules in depth. For more information, please refer to the “C/C++ User Guide”[9 (Related publications)].

Special aspects of COBOL

See the chapter on **linking, loading, and starting** in the “User Guide to COBOL85 or COBOL2000” [2 (Related publications), 4 (Related publications)].

5.2 Linkage techniques

It is necessary to distinguish between three basic linkage techniques:

- Static linking
- Dynamic loading of the C/COBOL runtime system and the internal routines (partial bind linkage technique)
- Dynamic linking with the DBL

For examples of the input for the linkage techniques explained here, refer to section “Input examples”.

5.2.1 Static linking with BINDER

You use BINDER to combine object modules and link-and-load modules (LLMs) to form a link-and-load module (LLM) which is then saved as an element of type “L” in a library.

If you use static linking with CRTE there are no unresolved references. You use the library SYSLNK.CRTE when linking.

5.2.2 Dynamic loading of the C/COBOL runtime system and the internal routines (partial bind)

When linking with BINDER it is possible to link not the entire C, C++ or
COBOL runtime system and the internal routines but instead simply linkage modules which resolve all open external references to the preloadable components of the CRTE. Unless they are preloaded, the required modules themselves are then not dynamically loaded until runtime.

There are two variants of the partial bind linkage method:

- Standard partial bind
- Complete partial bind

In view of the considerable savings in disk space, it is advisable always to use the partial bind linkage technique when linking the following programs:

- C programs
- COBOL programs
- Programs in other languages which contain components written in C or COBOL or which use common internal routines.

In all cases, CRTE must be installed in the executing system.

i
- The corresponding CRTE must be available when the program is called.
- If you use shared libraries under POSIX then successful linkage is only guaranteed if you use complete partial bind.

Standard partial bind

When linking with standard partial bind, you should use the library SYSLNK.CRTE.PARTIAL-BIND instead of SYSLNK.CRTE (see section “Libraries for thepartial bind linkage method on /390 systems”).

In the event of a standard partial bind, the linkage modules loaded from SYSLNK.CRTE.PARTIAL-BIND resolve all open unresolved references of the object module that is to be linked. Other external references are ignored.

i If a loaded module requires runtime system entries that are not needed by the module that has already been loaded and have therefore not been linked then unresolved external references will occur. In such cases, you should use the complete partial bind linkage technique (see “Complete partial bind”).

Depending on the mode (C-Front or ANSI), further libraries are required when linking C++ programs (see the sections “Cfront-C++-library functions and runtime system” and “ANSI-C++ libraries and runtime systems”).

Complete partial bind

When linking using complete partial bind, you should use the library SYSLNK.CRTE.COMPL instead of SYSLNK.CRTE (see section “Libraries for the partialbind linkage method on /390 systems”).

In the event of a complete partial bind, the loaded linkage modules in SYSLNK.CRTE.COMPL receive all the entries and external data of the full C or COBOL runtime system. This means that the unresolved external references that may occur in the case of a standard partial bind are excluded when you perform a complete partial bind.

Complete partial bind with C, COBOL and mixed C/COBOL programs

The C and COBOL runtime systems are linked either in their entirety or not at all. This means that:

- If your user program requires at least one C entry then all external references to the entire C runtime system are resolved in the event of a complete partial bind.
- If your COBOL program requires at least one COBOL entry then all external references to the entire COBOL runtime system are resolved in the event of a complete partial bind.
- In the case of pure C programs, no COBOL entries are linked.
- As all entries and all external variables are linked in, they can collide with the entries or external variables of the same name in the application program (duplicate entries). Redefinition of runtime system functions by functions of the same name in the application program is no longer possible, either. Names reserved by the runtime system can be displayed using the following LMS statement:
  `//SHOW-ELEMENT (SYSLNK.CRTE.COMPL,*,L),LLM-INF=PAR(INF=ESVD) .`

> ¡ In the case of COBOL and mixed C/COBOL programs the complete partial bind linkage method can only be used if the objects run under POSIX and shared objects are loaded.

Complete partial bind for C++ programs

When binding C++ programs (ANSI), the library SYSLNK.CRTE.CPP-COMPL is also required.

> ¡ The complete partial bind technique is not supported for C++ programs in Cfront mode.

Special characteristics of complete partial bind

You should remember the following points when linking with SYSLNK.CRTE.COMPL:

- The function *_edt* is not supported.
- The POSIX and TIME link switches are not present. They are automatically included when you use the cc command under POSIX.
- No upwards compatibility is guaranteed for the extension of the C/C++ systems by new functions in the COMPL libraries. If the application is linked using complete partial bind then each new entry may theoretically generate a conflict with an entry of the same name that is already present in the application.

Employed subsystems

The following subsystems are used for both partial bind variants:

- CRTEPART (for C)
- COBPART (for COBOL)
- CRTESIS (for common routines)

> ¡

For performance reasons and in order to gain full advantage from the partial bind linkage method, these subsystems should be preloaded.

5.2.3 Dynamic linking with DBL

With the Dynamic Binder Loader you can perform the following operations in one go:

1. Temporarily link modules to form a loadable unit
2. Store the loadable unit in working memory and start it

The generated load unit is deleted at the end of program execution.

If the specified start module is a prelinked module, it must be linked with open external references to CRTE, i.e. without a RESOLVE-BY-AUTOLINK statement to CRTE.
With BINDER, incomplete linkage is possible without precautionary measures.

If the C and/or COBOL runtime system is loaded in the high address space of class 4 memory (see also section “Shareability of CRTE”) you must specify PROG-MODE=ANY in the START-PROGRAM command.

> The CRTE libraries for the partial bind linkage method are **only allowed for linking**. However, you may **not use** these libraries as the BSLIB **for loading** programs containing unresolved external references in the event of dynamic binding with DBL, especially not when the CRTE subsystem is preloaded. In this case you may only use the SYSLNK.CRTE library as a BLSLIB.

Employed subsystems

The dynamic linkage technique with DBL described here requires the following subsystems:

- CRTEC (for C)
- CRTECOB (for COBOL)

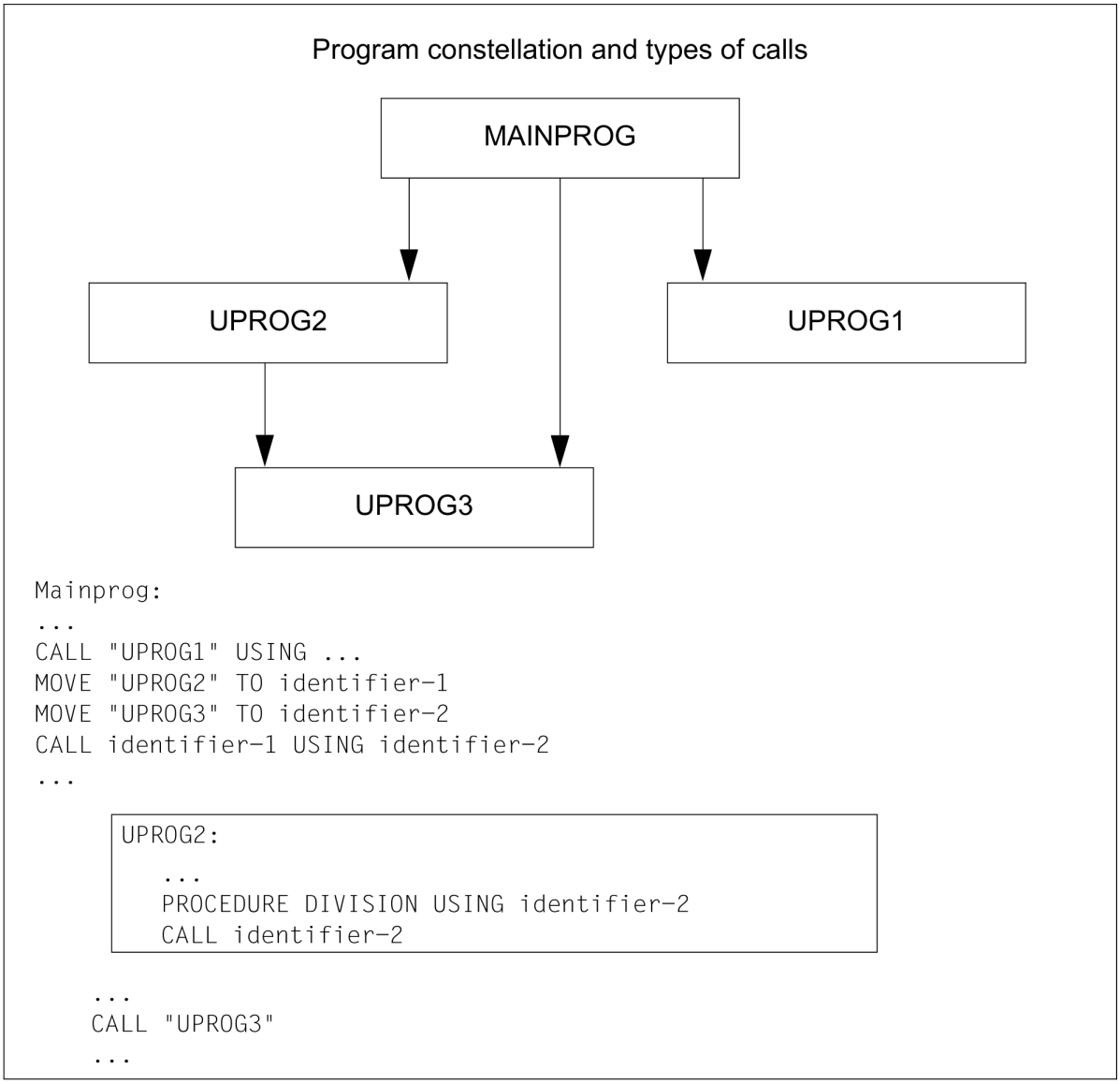
5.2.4 Linking prelinked modules

The runtime system and ILCS modules must not be linked in when prelinked modules are linked if the prelinked modules concerned are to be linked to other objects or prelinked modules at a later time. Some particular runtime system modules such as, e.g. IT0PCD or ITCMPOVH, may only be present once in an application, otherwise the data interchange between the runtime modules will not function correctly. This does not apply for encapsulated subsystems that are described in detail in section “Encapsulated subsystems”.

The runtime system and ILCS may also not be linked statically into assembler or COBOL applications that dynamically load subroutines or main modules at runtime, since the runtime system modules would then not be found by the dynamically loaded subroutines and this would lead to a second runtime system being loaded. The following exception also applies in this case: if only encapsulated subsystems are dynamically loaded, the application can be linked completely.

Example

The following example illustrates linking and loading techniques for program systems containing dynamically loadable subprograms.




```
Mainprog:
...
CALL "UPROG1" USING ...
MOVE "UPROG2" TO identifier-1
MOVE "UPROG3" TO identifier-2
CALL identifier-1 USING identifier-2
...
```

```
UPROG2:
   ...
   PROCEDURE DIVISION USING identifier-2
   CALL identifier-2
```

```
...
CALL "UPROG3"
...
```

SUBPR1 is only called in the form `CALL` *`literal`*.
SUBPR2 is only called in the form `CALL` *`identifier`*.
SUBPR3 is called in either way.

This means that external references are set up for SUBPR1 and SUBPR3; SUBPR2 is loaded dynamically.

The ways of initiating the program run for this program constellation are shown below.

The individual programs are stored as object modules under the element names MAINPROG, SUBPR1, SUBPR2 and SUBPR3 in the library USER-PROGRAMS.

Using the BINDER (linking LLMs)

BINDER keeps all external references and entry points visible by default; this is essential for the succeeding DBL run. Moreover, using BINDER enables external references to remain unresolved. This means that the runtime system does not have to be linked in. This is an advantage if a shareable runtime system is to be used for program execution.

The following example illustrates generating a single link- and-load module.

```
/START-PROGRAM $BINDER
 START-LLM-CREA GROSSMOD .....................................(1)
 INCLUDE-MODULES LIB=BENUTZER-PROGRAMME,ELEM=MAINPROG ........(2)
 INCLUDE-MODULES LIB=BENUTZER-PROGRAMME,ELEM=UPROG2 ..........(3)
 RESOLVE-BY-AUTOLINK LIB=BENUTZER-PROGRAMME ..................(4)
 SAVE-LLM LIB=MODUL.LIB ......................................(5)
 END
/ADD-FILE-LINK BLSLIB00,$.SYSLNK.CRTE ..................... (6)
/START-PROGRAM *MODULE(LIB=MODUL.LIB,ELEM=GROSSMOD,- .........(7)
 RUN-MODE=ADVANCED(ALT-LIB=YES,UNRES-EXT=DELAY,-
 LOAD-INFO=REFERENCE))
```

(1)	Creates a link-load-load module named PRLNKMOD.
(2)	Explicitly links in the main program module MAINPROG from the USER-PROGRAMS library.
(3)	Explicitly links in the module SUBPR2 from the USER-PROGRAMS library in order to avoid dynamic loading. This makes it unnecessary to assign the USER-PROGRAMS library with the link name COBOBJCT in the ensuing link-and-load operation.
(4)	Links in all other modules required (SUBPR1, SUBPR3) from the USER-PROGRAMS library.
(5)	Stores the generated link-and-load module as a type L element in the program library MODUL.LIB.
(6)	Assign the runtime library.
(7)	Call the link-and-load module PRLNKMOD.

5.2.5 Comparison and evaluation of the linkage techniques

The advantages and disadvantages of the individual linkage techniques are presented below.

Advantages and disadvantages of static linking

Advantages

- As far as the application is concerned, it is irrelevant whether a CRTE is installed on the user's system and, if it is, which version of CRTE is installed. This makes it easier to port the application to different systems.
- No link operations are required at runtime.
- No link name is required.

Disadvantages

- Increased disk storage requirements. The runtime system may account for up to an 800 KB increase in code.
- Longer link times
- Longer load times as it is always necessary to load the entire application.
- Corrections in the CRTE do not take effect in an already loaded application. It may be necessary to invalidate corrections in the employed CRTE by means of object corrections in the application.

Advantages and disadvantages of partial bind

Advantages

- Shorter production times due to faster linking with the partial bind libraries.
- Reduced disk storage requirement since no runtime system is linked.
- High-performance loading provided that the corresponding subsystems are loaded.
- Corrections in the CRTE generally take effect automatically, including in linked applications.

Disadvantage

CRTE must be installed in the executing system. (However, this can also be viewed as an advantage.)

Advantages and disadvantages of dynamic linking with DBL

Advantages

- No link run is required for production.
- Reduced disk storage requirement since no runtime system is linked.
- Corrections in the CRTE generally take effect automatically, including in linked applications.

Disadvantages

- CRTE must be installed in the executing system. (However, this can also be viewed as an advantage.)
- It is necessary to specify the location at which the open references can be resolved.
- The link process must be repeated on each load operation.

5.3 Linking with mixed languages

When mixed languages are involved, it is necessary to distinguish between:

- Mix of CRTE languages (COBOL, C, C++)
- Mix including “foreign” languages

5.3.1 Mix of CRTE languages (COBOL, C, C++)

If a language mix involves only CRTE languages, the same linkage techniques (with respect to CRTE) apply as when linking programs in the same language. These techniques are dependent on the language involved and the module format.

If only C and COBOL are involved, the runtime modules from the libraries SYSLNK.CRTE or SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL must be linked. However, if COBOL is involved, the SYSLNK.CRTE.COMPL library can only be used if the objects run under POSIX and shared objects are loaded.

If CFront-C++ is involved, SYSLNK.CRTE.COMPL may not be used. The runtime modules from the libraries SYSLNK.CRTE.CPP and SYSLNK.CRTE.CFCPP must also be linked.

> ¡ In the case of Cfront-C++, the complete partial bind linkage method is not supported.

5.3.2 Mix including “foreign” languages

“Foreign” languages are all ILCS-compatible languages that are not CRTE languages (Assembler, Fortran, Pascal, etc.). Compilers for these languages are shipped with language-specific runtime systems. These systems and CRTE must be considered when performing linkage.

The CRTE libraries SYSLNK.CRTE or SYSLINK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL (see also "Dynamic loading of the C/COBOL runtime system and the internal routines (partial bind)") must always be linked before the appropriate language-specific runtime libraries.

> When linking with SYSLNK.CRTE.COMPL, please note "Special characteristics of complete partial bind" in section "Dynamic loading of the C/COBOL runtime system and the internal routines (partial bind)".

5.4 Input examples

The input examples below illustrate how CRTE is linked statically and dynamically with BINDER and DBL. Each example uses a different program constellation:

- In section “Static linkage of a C, C++ or COBOL program” you will find examples for linking programs which link CRTE from the library SYSLNK.CRTE.
- In section “Linking programs which dynamically load the C or COBOL runtime systems(partial bind linkage technique)” you will find examples for linking programs which link CRTE from the library SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL.
- In section “Dynamic linkage with DBL” you will find examples of how to use the DBL for the dynamic linkage of programs which link CRTE from the library SYSLNK.CRTE.
- In section “Linkage with mixed “foreign” languages” you will find examples of mixed language linkage.

> ¡ With regard to linkage of C and C++ programs, please read the notes in section “Language-specific considerations”.

5.4.1 Static linkage of a C, C++ or COBOL program

Static linkage with BINDER

1. C or COBOL program (object module or link-and-load module)

```
/START-BINDER
//START-LLM-CREATION INT-NAME=internal-name ----------------------- (1)
//INCLUDE-MODULES LIB=..., ELEM=... ------------------------------- (2)
//RESOLVE-BY-AUTOLINK LIB=userlibrary ----------------------------- (3)
//RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE --------------------------- (4)
//SAVE-LLM LIB=..., ELEM=... -------------------------------------- (5)
//END
```

   (1) A link-and-load module with the name internal-name is generated.

   (2) The program modules are explicitly included.

   (3) The user library `userlibrary` is statically linked using the Autolink function.

   (4) The library $.SYSLNK.CRTE which, among other things, contains the complete C and COBOL runtime system, is statically linked with the Autolink function.

   (5) The generated, current link-and-load module is saved.

2. C++-Program, Cfront mode (link-and-load module) for compiler versions <= V2.2

```
/START-BINDER
//START-LLM-CREATION INT-NAME=internal-name ----------------------- (1)
//INCLUDE-MODULES LIB=..., ELEM=... ------------------------------- (2)
//RESOLVE-BY-AUTOLINK LIB=userlibrary ----------------------------- (3)
//RESOLVE-BY-AUTOLINK LIB=($.SYSLNK.CRTE,$.SYSLNK.CRTE.CPP) -------- (4)
//SAVE-LLM LIB=..., ELEM=... -------------------------------------- (5)
//END
```

   (1) A link-and-load module with the name internal-name is generated.

   (2) The program modules are explicitly included.

   (3) The user library `userlibrary` is statically linked using the Autolink function.

   (1) A link-and-load module with the name internal-name is generated.

   (2) The program modules are explicitly included.

   (3) The user library `userlibrary` is statically linked using the Autolink function.

   (4) The following are also statically linked with the Autolink function:

   - The library $.SYSLNK.CRTE which, among other things, contains the complete C and COBOL runtime system (see "CRTE V10.1A selectable unit").
   - The library $.SYSLNK.CRTE.CPP which contains the Cfront-C++ library functions (see section “Cfront-C++-library functions and runtime system”).

(5) The current link-and-load module is saved.

3. Cfront C++ program (link-and-load module) for compiler versions as of V3.0

```
/START-CPLUS-COMPILER ---------------------------------------------- (1)
//MODIFY-BIND-PROPERTIES INCLUDE=*LIB(LIB=...,ELEM=...)
//MODIFY-BIND-PROPERTIES RESOLVE=*AUTOLINK(userlibrary)
//MODIFY-BIND-PROPERTIES RUNTIME-LANG=CPLUSPLUS(CPP),STDLIB=STATIC
//BIND OUTPUT=*LIB(LIB=...,ELEM=...)
```

(1) The linkage editor is implicitly called by the C++ compiler.The following statements for the C++ compiler have the same functionality as the BINDER statements in Example 2.

5.4.2 Linking programs which dynamically load the C or COBOL runtime systems (partial bind linkage technique)

Static linkage with BINDER

1. C or COBOL program (object module or link-and-load module)

```
/START-BINDER
//START-LLM-CREATION INT-NAME=internal-name ----------------------- (1)
//INCLUDE-MODULES LIB=..., ELEM=... ------------------------------- (2)
//RESOLVE-BY-AUTOLINK LIB=userlibrary ----------------------------- (3)
//RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE.PARTIAL-BIND -------------- (4)
//SAVE-LLM LIB=..., ELEM=... -------------------------------------- (5)
//END
```

   (1) A link-and-load module with the name internal-name is generated.

   (2) The program modules are explicitly included.

   (3) The user library is statically linked using the Autolink function.

   (4) The library $.SYSLNK.CRTE.PARTIAL-BIND (standard partial bind) which, among other things, contains the complete C and COBOL runtime system (see "CRTE V10.1A selectable unit"), is statically linked with the Autolink function.

   Alternatively, you can use the complete partial bind linkage method. To do this, use the library $.SYSLNK.CRTE.COMPL instead of $.SYSLNK.CRTE.PARTIAL-BIND.

   (5) The current link-and-load module is saved.

2. C++ program (link-and-load module) for compiler versions <= V2.2

```
/START-BINDER
//START-LLM-CREATION INT-NAME=internal-name -------------------------- (1)
//INCLUDE-MODULES LIB=..., ELEM=... ---------------------------------- (2)
//RESOLVE-BY-AUTOLINK LIB=user library ------------------------------- (3)
//RESOLVE-BY-AUTOLINK LIB=($.SYSLNK.CRTE.PARTIAL-BIND,$.SYSLNK.CRTE.CPP)(4)
//SAVE-LLM LIB=..., ELEM=... ----------------------------------------- (5)
//END
```

   (1) A link-and-load module with the name internal-name is generated.

   (2) The program modules are explicitly included.

   (3) The user library is statically linked using the Autolink function.

   (4) The following are also statically linked with the Autolink function:

   - The library $.SYSLNK.CRTE.PARTIAL-BIND which, among other things, contains the complete C and COBOL runtime system (see "CRTE V10.1A selectable unit").
   - The library $.SYSLNK.CRTE.CPP which contains the Cfront-C++ library functions (see section “Cfront-C++-library functions and runtime system”).

(5) The current link-and-load module is saved.

3. C++ program (link-and-load module) for compiler versions as of V3.0

```
/START-CPLUS-COMPILER ---------------------------------------------- (1)
//MODIFY-BIND-PROPERTIES INCLUDE=*LIB(LIB=...,ELEM=...)
//MODIFY-BIND-PROPERTIES RESOLVE=*AUTOLINK(userlibrary)
//MODIFY-BIND-PROPERTIES RUNTIME-LANG=CPLUSPLUS(CPP), STDLIB=DYNAMIC
//BIND OUTPUT=*LIB(LIB=...,ELEM=...)
```

(1) The linkage editor is implicitly called by the C++ compiler.The following statements for the C++ compiler have the same functionality as the BINDER statements in Example 2.

Loading and starting link and load modules

```
/START-PROGRAM *MODULE(LIB=...,ELEM=...,RUN-MODE=ADVANCED)
```

5.4.3 Dynamic linkage with DBL

1. C or COBOL program (object module)

```
/SET-TASKLIB $.SYSLNK.CRTE ---------------------------------------- (1)
...
/START-PROGRAM *MODULE(LIB=..., ELEM=..., PROG-MODE=ANY) ------------ (2)
```

   (1) The library $.SYSLNK.CRTE is assigned as Tasklib.

   (2) If the specified start module is a prelinked module then this module must be linked with open external references to the CRTE, i.e. without a RESOLVE statement for the CRTE (see section “The Dynamic Binder Loader (DBL)”).

2. C or COBOL program (object module or link-and load module)

```
/ADD-FILE-LINK LINK-NAME=BLSLIB00,FILE-NAME=userlibrary ------------ (1)
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE
...
/START-PROGRAM *MODULE(LIB=..., ELEM=..., PROG-MODE=ANY, ----------- (2)
/RUN-MODE=ADVANCED(ALT-LIB=YES,AUTO=ALT-LIB))
```

   (1) The link names BLSLIB00 and BLSLIB01 are assigned to the libraries userlibrary and $.SYSLNK.CRTE respectively. This means that `userlibrary` is searched before SYSLNK.CRTE during the resolution of unresolved references. The link names must be assigned **before** the linkage editor run and are not released after the linkage editor has executed.

   (2) If the specified start module is a prelinked module then this module must be linked with open external references to the CRTE, i.e. without a RESOLVE statement for the CRTE (see section “The Dynamic Binder Loader (DBL)”).

3. C++ program (link-and-load module) for compiler versions <= V2.2

```
/ADD-FILE-LINK LINK-NAME=BLSLIB00,FILE-NAME=userlibrary
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.CPP ------ (1)
...
/START-PROGRAM *MODULE(LIB=..., ELEM=..., PROG-MODE=ANY,-  --------- (2)
/RUN-MODE=ADVANCED(ALT-LIB=YES,AUTO=ALT-LIB))
```

   (1) Taking example 2 as our starting point, the link name BLSLIB02 is additionally assigned to the Cfront-C++ library $.SYSLNK.CRTE.CPP (see section “Cfront-C++-library functions and runtime system”). This means that the program can also use the C++ library functions for complex math and standard I/O. Please note that in the case of C++ programs which use a C++ version <= V2.2, the link names must be assigned in the specified order.During the resolution of unresolved external references, the sequence in which the individual libraries are searched corresponds to the order of their link names BLSLIB00 -> BLSLIB01 -> BLSLIB02.

   (2)

If the specified start module is a prelinked module then this module must be linked with open external references to the CRTE, i.e. without a RESOLVE statement for the CRTE (see section “The Dynamic Binder Loader (DBL)”).

4. C++ program, Cfront mode (link-and-load module) for compiler versions as of V3.0

```
/ADD-FILE-LINK LINK-NAME=BLSLIB00,FILE-NAME=userlibrary
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE.RTSCPP
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.CFCPP ---- (1)
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE
...
/START-PROGRAM *MODULE(LIB=..., ELEM=..., PROG-MODE=ANY,------------ (2)
 RUN-MODE=ADVANCED(ALT-LIB=YES,AUTO=ALT-LIB))
```

(1) Taking example 2 as our starting point, a link name is additionally assigned to the library SYSLNK.CRTE.CFCPP (Cfront runtime system, see see section “Cfront-C++-library functions and runtime system”). In the case of C++ programs as of C++ version V3.0, the sequence in which the link names BLSLIBnn are assigned is no longer mandatory.During the resolution of unresolved external references, the sequence in which the individual libraries are searched corresponds to the order of their link names BLSLIB00 -> BLSLIB01 -> BLSLIB02.

(2) If the specified start module is a prelinked module then this module must be linked with open external references to the CRTE, i.e. without a RESOLVE statement for the CRTE (see section “The Dynamic Binder Loader (DBL)”).

5.4.4 Linkage with mixed “foreign” languages

Static linking with BINDER

```
/START-BINDER
//START-LLM-CREATION INT-NAME=... ------------------------------------ (1)
//INCLUDE-MODULES LIB=..., ELEM=... ---------------------------------- (2)
//RESOLVE-BY-AUTOLINK LIB=userlibrary -------------------------------- (3)
//RESOLVE-BY-AUTOLINK LIB=($.SYSLNK.CRTE,$.rts-foreignlanguage1,//
                         rts-foreignlanguage2) ----------------------- (4)
// ...
//SAVE-LLM LIB=..., ELEM=...
//END
```

(1) The names of the load module and the output file are specified.

(2) The program modules are explicitly included.

(3) The user library is statically linked with the Autolink function.

(4) The library $.SYSLNK.CRTE and the runtime systems rts-foreignlanguage1 and rts-foreignlangauage2 are statically linked with the Autolink function.

Dynamic linkage with DBL

```
/ADD-FILE-LINK LINK-NAME=BLSLIB00,FILE-NAME=userlibrary ---------------- (1)
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE -------------- (1)
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=rts-foreignlanguage1 ------- (1)
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=rts-foreignlanguage2 ------- (1)
...
/START-PROGRAM *MODULE(LIB=..., ELEM=..., PROG-MODE=ANY,-
/RUN-MODE=ADVANCED(ALT-LIB=YES,AUTO=ALT-LIB)) -------------------------- (2)
```

(1) The link names BLSLIB00 to BLSLIB03 are assigned in sequence to the libraries $.SYSLNK.CRTE, rts-foreignlanguage1 and rts-foreignlanguage.During the resolution of unresolved external references, the sequence in which the individual libraries are searched corresponds to the order of their link names BLSLIB00 -> BLSLIB01 -> BLSLIB02 -> BLSLIB03.

(2) If the specified start module is a prelinked module then this module must be linked with open external references to the CRTE, i.e. without a RESOLVE statement for the CRTE (see section “The Dynamic Binder Loader (DBL)”).

6 Appendix: use of the ILCS routines IT0INITS and IT0ININ

This section contains an example illustrating the following problems:

- Calling an ILCS Assembler program by a non-ILCS Assembler program which does not use any ASSEMBH structure macros. In this case, ILCS must be dynamically initialized using the ILCS routine IT0INITS.

  When an ILCS Assembler program is called by a structured non-ILCS Assembler program (specification ILCS=NO in the @ENTR macro) it is not necessary to call IT0INITS.
- The dynamic loading of ILCS objects in languages which were hitherto not involved. In this case the relevant language environment must be initialized with the ILCS routine IT0ININ. The ASSEMBH structure macro @ININ is available for calling IT0ININ.

Following the general description of the example you will find a representation of the source programs as of "Reproduction of the source programs" and a complete trace listing dealing with compilation, linkage and program execution as of "Trace listings".

6.1 General description of the example

Call schema

```
Non-ILCS Assembler program CALLINTF
  -->
    ILCS Assembler program ILCSINTF
      -->
        ILCS-C program CUPRO
           -->
             ILCS-COBOL program COBUPRO
```

The C and COBOL programs are dynamically loaded in the Assembler program ILCSINTF by means of the BIND macro.

Non-ILCS Assembler program CALLINTF

Reproduction of the source program:

> "*Non-ILCS Assembler program CALLINTF*" in "Reproduction of the source programs"

Trace listing for compilation:

> "*Compilation of the Assembler programs CALLINTF (non-ILCS main program) and ILCSINTF (ILCS subprogram)I"* in "Trace listings"

In the example, CALLINTF is the main program. If it is to be possible to call CALLINTF in turn by a non-ILCS program, it must be modified as follows:

- when it is called, save the registers that will be required later
- before returning, restore the previously saved registers
- return to the calling non-ILCS program (RETURN)

Description of CALLINTF

1. The initialization routine IT0INITS is called in order to initialize ILCS dynamically. Register 0 must contain the value 0 (number of parameters = 0).
2. IT0INITS returns in register 0 the address of the generated PCD. This address be saved in the save area. The save area itself is generated by means of the CRTE macro IT0VSA.
3. Register 13 is loaded with the address of the save area (ILCS convention).
4. An ILCS Assembler program is called; in this example this program has the name “ILCSINTF”.

ILCS Assembler program ILCSINTF

Reproduction of the source program:

> "*ILCS Assembler program ILCSINTF*" in "Reproduction of the source programs"

Trace listing for compilation:

> "*Compilation of the Assembler programs CALLINTF (non-ILCS main program) and ILCSINTF (ILCS subprogram)I"* in "Trace listings"

Description of ILCSINTF

1. In order to prevent the ILCS-C and COBOL modules from being loaded in memory more than once, a test is carried out to see whether the C and COBOL modules have already been loaded.
2. If the modules have already been loaded, only the C module is called.
3. If the modules have not been loaded:

   - the BIND macros are executed
   - the ILCS routine IT0ININ is called with the structure macro @ININ in order to initialize the runtime environment for the languages C and COBOL
   - the C module is called.

ILCS-C program CUPRO and ILCS-COBOL program COBUPRO

Reproduction of the source program:

> "*ILCS-C program CUPRO*" and "*IILCS-COBOL program COBUPRO*" in "Reproduction of the source programs"

Trace listing for compilation:

> "*Compilation and prelinking of the COBOL program COBUPRO*" and *"Compilation and prelinking of the C subprogram CUPRO"* in "Trace listings"

Description of the programs

The C program merely calls the COBOL program. The COBOL program issues a message at the terminal acknowledging that the call has been successful.

Since the programs are to be dynamically loaded with the BIND macro and then preloaded in class 4/5 memory, the following option is specified for compilation with the C++ or COBOL85 compiler:

```
COMPILER-ACTION=MODULE-GEN(MOD-FORM=LLM,SHAREABLE-CODE=YES)
```

In a subsequent linkage run with BINDER a public slice is formed from the code CSECT and a private slice is formed from the data CSECT.

6.2 Reproduction of the source programs

This section contains the following source programs:

- *Non-ILCS Assembler program CALLINTF*
- *ILCS Assembler program ILCSINTF*
- *ILCS-C program CUPRO*
- *ILCS-COBOL program COBUPRO*

Non-ILCS Assembler program CALLINTF

```
CALLINTF CSECT
         TITLE 'CALL TO ILCSINTF'
CALLINTF AMODE ANY
CALLINTF RMODE ANY
R15      EQU   15
R14      EQU   14
R12      EQU   12
R13      EQU   13
R3       EQU   3
R4       EQU   4
R0       EQU   0
         PRINT NOGEN
         BALR  R3,0
         USING *,R3
* Call of IT0INITS
* Register 0 must contain the value 0 (no parameters)
         XR    R0,R0
         L     R15,INIADDR
         BASR  R14,R15
* Store the PCD address in the save area
         ST    R0,ILCVPCD
* Load the save area address
         LA    R13,ILCVAI
* Call ILCSINTF in order to link the C and COBOL modules
* (with the BIND macro) and to execute them for the first time
         L     R15,INTFADDR
         BASR  R14,R15
         TERM
* Definition of the address of IT0INITS
INIADDR  DC    V(IT0INITS)
* Definition of the address of ILCSINTF
INTFADDR DC    V(ILCSINTF)
         PRINT GEN
* Definition of the save area
         IT0VSA ILC
         DROP  R3
         END
```

ILCS Assembler program ILCSINTF

```
ILCSINTF START
         PRINT GEN
ILCSINTF @ENTR TYP=E,AMODE=ANY,RMODE=ANY,VERS=000,AUTHOR=LEROY,        X
               FUNCT='ILCSINTF',TITLE=YES,ILCS=YES
```

```
* Test whether the modules have already been linked in
         L     R15,CMODADDR
         @IF   ZE
         LTR   R15,R15
         @THEN
****
* A separate BIND macro is issued for each language and module
****
* Load the C module
         BIND  SYMBOL=PRNT,SYMBLAD=CMODADDR,ALTLIB=YES,MSG=ERROR
* Load the COBOL module
         BIND  SYMBOL=COBUPRO,ALTLIB=YES,MSG=ERROR
* Initialize the C and COBOL language environment by calling
* the ILCS routine IT0ININ with the structure macro @ININ
         @ININ
         @BEND
* Call the C module
         @PASS ADDR=CMODADDR
         @EXIT
* Definition of the address of the C module
CMODADDR DS    F
         @END  LTORG=YES,DROP=()
         END
```

ILCS-C program CUPRO

```
#include <stdio.h>
extern void cobupro(void);
void prnt()
{
  printf("CUPRO before\n");
  cobupro();
  printf("CUPRO after1\n");
  cobupro();
  printf("CUPRO after2\n");
}
```

ILCS-COBOL program COBUPRO

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBUPRO.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
           TERMINAL IS T.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       PROCEDURE DIVISION.
       DIALOG SECTION.
       DIAL.
           DISPLAY "COBUPRO WAS HERE" UPON T.
       BYEBYE.
           EXIT PROGRAM.
```

6.3 Trace listings

This section contains the following trace listings:

- *Compilation of the Assembler programs CALLINTF (non-ILCS main program) and ILCSINTF (ILCS subprogram)*
- *Compilation and prelinking of the COBOL program COBUPRO*
- *Compilation and prelinking of the C subprogram CUPRO*
- *Linking, loading and starting the program execution unit*

Compilation of the Assembler programs CALLINTF (non-ILCS main program) and ILCSINTF (ILCS subprogram)

```
(IN)     /START-ASSEMBH
(OUT)    %  BLS0523 ELEMENT 'ASSEMBH', VERSION '013', TYPE 'C' FROM LIBRARY
        ':2OSH:$TSOS.SYSPRG.ASSEMBH.013' IN PROCESS
(OUT)    %  BLS0500 PROGRAM 'ASSEMBH', VERSION '01.3A02' OF '2012-04-04'
         LOADED
(OUT)    %  BLS0552 COPYRIGHT (C) FUJITSU TECHNOLOGY SOLUTIONS   2012.
         ALL RIGHTS RESERVED
(OUT)    %  ASS6010 V01.3A02 OF BS2000 ASSEMBH  READY
(IN)     //COMPILE SOURCE=*LIB(LIB=PLAM.ASSEMBH,ELEM=CALLINTF.ASM), -
         MACRO-LIB=($.SYSLNK.CRTE,$.SYSLIB.BS2CP.180),MODULE-
LIB=PLAM.ASSEMBH, -
         LISTING=*PAR(OUTPUT=*LIB(LIB=PLAM.ASSEMBH,ELEM=CALLINTF))
(OUT)    %  ASS6011 ASSEMBLY TIME: 202 MSEC
(OUT)    %  ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
(OUT)    %  ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
(OUT)    %  ASS6006 LISTING GENERATOR TIME: 78 MSEC
(IN)     //COMPILE SOURCE=*LIB(LIB=PLAM.ASSEMBH,ELEM=ILCSINTF.ASM),-
         MACRO-LIB=($.SYSLIB.ASSEMBH.013,$.SYSLIB.BS2CP.180, -
         (MODULE-LIB=PLAM.ASSEMBH, -
         LISTING=*PAR(OUTPUT=*LIB(LIB=PLAM.ASSEMBH,ELEM=ILCSINTF))
(OUT)    %  ASS6011 ASSEMBLY TIME: 579 MSEC
(OUT)    %  ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
(OUT)    %  ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
(OUT)    %  ASS6006 LISTING GENERATOR TIME: 126 MSEC
(IN)     //END
(OUT)    %  ASS6012 END OF ASSEMBH
```

Compilation and prelinking of the COBOL program COBUPRO

```
(IN)     /START-COBOL2000-COMPILER -
         SOURCE=*LIB(LIB=PLAM.COBOL,ELEM=COBUPRO.COB),-
         COMPILER-ACTION=MODULE-GEN(SHAREABLE-CODE=YES,MODULE-FORM=LLM),-
         MODULE-OUTPUT=*LIB(LIB=PLAM.COBOL,ELEM=COBUPRO),-
         LISTING=*PAR(OUTPUT=*LIB(LIB=PLAM.COBOL))
(OUT)    %  BLS0500 PROGRAM 'COBOL2000', VERSION '01.5A00' OF '2009-03-12'
LOADED
(OUT)         %  CBL9000 FUJITSU TECHNOLOGY SOLUTIONS 2009
         ALL RIGHTS RESERVED
(OUT)    %  CBL9017 COMPILATION INITIATED, VERSION IS  V01.5A00
(OUT)    %  CBL9097 COMPILATION COMPLETED WITHOUT ERRORS
(OUT)    %  CBL9004 COMPILATION OF COBUPRO USED  0.3494 CPU SECONDS
(IN)     /START-BINDER
(OUT)    %  BND0500 BINDER VERSION 'V02.6A30' STARTED
```

```
(IN)     //START-LLM-CREAT INT-NAME=COB,SLICE-DEFINITION=BY-
ATTRIB(PUBLIC=YES)
(IN)     //INCLUDE-MODULE LIB=PLAM.COBOL,ELEM=COBUPRO
(IN)     //SAVE-LLM LIB=PLAM.COBOL-SHARE,ELEM=COBUPROSHARE
(OUT)    %  BND3101 SOME EXTERNAL REFERENCES UNRESOLVED
(OUT)    %  BND1501 LLM FORMAT: '2'
(IN)     //END
(OUT)    %  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'UNRESOLVED
         EXTERNAL'
```

Compilation and prelinking of the C subprogram CUPRO

```
(IN)     /START-CPLUS-COMPILER
(OUT)    %  BLS0523 ELEMENT 'SDFCC', VERSION '03.2D11', TYPE 'L' FROM LIBRARY
         ':2OSH:$TSOS.SYSLNK.CPP-S.032' IN PROCESS
(OUT)    %  BLS0524 LLM 'SDFCC', VERSION '03.2D11' OF '2012-04-06 08:31:13'
LOADED
(OUT)    %  BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2012.
         ALL RIGHTS RESERVED
(OUT)    %  CDR9992 : BEGIN C/C++(BS2000/OSD) VERSION 03.2C11
(IN)     //MOD-MOD-PROP SHAREABLE-CODE=*YES
(IN)     //MOD-SOU-PROP C
(IN)     //MOD-LIS-PROP OPTIONS=*YES,SOURCE=*YES(MIN-MSG-WEI=*WARN),-
         OUTPUT=*LIB(LIB=PLAM.C)
(IN)     //COMPILE SOURCE=*LIB(LIB=PLAM.C,ELEM=CUPRO.C),-
         MODULE-OUTPUT=*LIB(LIB=PLAM.C)
(OUT)    %  CDR9907 : NOTES: 7  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)    %  CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0113 SEC
(OUT)    %  CDR9992 : END
(OUT)    %  CCM0998 CPU TIME USED: 1.1996 SECONDS
(IN)        /START-PROG $BINDER
(OUT)    %  BLS0500 PROGRAM 'BINDER', VERSION 'V02.6A30' OF '2010-10-19'
LOADED
(OUT)    %  BLS0552 COPYRIGHT (C) FUJITSU TECHNOLOGY SOLUTIONS   2009. ALL
RIGHTS          RESERVED
(OUT)    (MSG) %  %  BLS0519 PROGRAM '$BINDER' LOADED
(IN)     //START-LLM-CREAT INT-NAME=C,SLICE-DEFINITION=BY-ATTRIB(PUBLIC=YES)
(IN)     //INCLUDE-MODULE LIB=PLAM.C,ELEM=CUPRO
(IN)     //SAVE-LLM LIB=PLAM.C-SHARE,ELEM=CUPROSHARE
(OUT)    %  BND3101 SOME EXTERNAL REFERENCES UNRESOLVED
(OUT)    %  BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT)    %  BND1501 LLM FORMAT: '2'
(IN)     //END
```

Linking, loading and starting the program execution unit

```
(IN)     /START-PROG $BINDER
(OUT)    %  BLS0500 PROGRAM 'BINDER', VERSION 'V02.6A30' OF '2010-10-19' LOADED
(OUT)    %  BLS0552 COPYRIGHT (C) FUJITSU TECHNOLOGY SOLUTIONS   2009.
         ALL RIGHTS RESERVED
(IN)     //START-LLM-CREAT INT-NAME=ASS
(IN)     //INCLUDE-MODULE LIB=PLAM.ASSEMBH,ELEM=CALLINTF
(IN)     //INCLUDE-MODULE LIB=PLAM.ASSEMBH,ELEM=ILCSINTF
(IN)     //RESOLVE LIB=$.SYSLNK.CRTE.PARTIAL-BIND
(IN)     //MODIFY-SYMBOL-VISIBILITY VISIBLE=NO
*)
```

```
(OUT)    %  BND1111 '304' SYMBOL(S) PROCESSED IN CURRENT STATEMENT
(IN)     //SAVE-LLM LIB=PLAM.PROGRAM,ELEM='START-CALLINTF'
(OUT)    %  BND3101 SOME EXTERNAL REFERENCES UNRESOLVED
(OUT)    %  BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT)    %  BND1501 LLM FORMAT: '1'
(IN)     //END
(OUT)    %  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'UNRESOLVED
         EXTERNAL'
(IN)     /ADD-FILE-LINK LINK=BLSLIB01,FILE-NAME=PLAM.COBOL-SHARE
(IN)     /ADD-FILE-LINK LINK=BLSLIB02,FILE-NAME=PLAM.C-SHARE
(IN)     /ADD-FILE-LINK LINK=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE
*)
(IN)     /ADD-FILE-LINK LINK=BLSLIB04,FILE-NAME=$.SYSLIB.ASSEMBH.013
(IN)     /START-PROG *MOD(LIB=PLAM.PROGRAM,ELEM=START-CALLINTF,-
         RUN-MODE=ADV(ALT-LIB=YES),PROG-MODE=ANY)
(OUT)    %  BLS0523 ELEMENT 'START-CALLINTF', VERSION '@' FROM LIBRARY
         ':2OSC:$MANUBSP.PLAM.PROGRAM' IN PROCESS
(OUT)    %  BLS0524 LLM 'ASS', VERSION ' ' OF '2012-05-14 19:42:23' LOADED
(OUT)    (MSG) %  %  BLS0519 PROGRAM 'START-CALLINTF' LOADED
(OUT)    CUPRO before
(OUT)    COBUPRO WAS HERE
(OUT)    CUPRO after1
(OUT)    COBUPRO WAS HERE
(OUT)    CUPRO after2
```

*) These two statements are not mandatory. However, it is recommended that you include them.

7 Related publications

You will find the manuals on the internet at http://manuals.ts.fujitsu.com. You can order printed copies of those manuals which are displayed with an order number.

[1] **BS2000**
Softbooks English
CD-ROM

[2] **COBOL2000** (BS2000)
COBOL Compiler
User's Guide

[3] **COBOL2000** (BS2000)
COBOL Compiler
Reference Manual

[4] **COBOL85 (BS2000)**
COBOL Compiler
User's Guide

[5] **COBOL85 (BS2000)**
COBOL Compiler
Reference Manual

[6] **C** (BS2000/OSD)
C Compiler
User Guide

[7] **C Library Functions** (BS2000)
Reference Manual

[8] **C Library Functions** (BS2000)
for POSIX Applications
Reference Manual

[9] **C/C++** (BS2000)
C/C++ Compiler
User Guide

[10] **C++** (BS2000)
C++ Library Functions

[11] **C/C++** (BS2000)
POSIX Commands of the C/C++ Compiler
User Guide

[12] **ASSEMBH** (BS2000)
User Guide

[13]

POSIX (BS2000)
POSIX Basics for Users and System Administrators
User Guide

[14] **BLSSERV**
Dynamic Binder Loader / Starter in BS2000/OSD
User Guide

[15] **BINDER**
Binder in BS2000/OSD
User Guide

[16] **LMS** (BS2000)
SDF Format
User Guide

[17] **IMON** (BS2000)
Installation Monitor
User Guide

[18] **DSSM/SSCM**
Subsystem Management in BS2000
User Guide

[19] **BS2000 OSD/BC**
Executive Macros
User Guide

[20] **BS2000 OSD/BC**
Introduction to System Administration
User Guide

[21] **BS2000 OSD/BC**
System Installation
User Guide