z/OS

Language Environment Concepts Guide

Version 2 Release 1

SA38-0687-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 45.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this document

IBM® z/OS® Language Environment® (also called Language Environment) provides common services and language-specific routines in a single runtime environment for C, C++, COBOL, Fortran (z/OS only; no support for z/OS UNIX System Services, or CICS®), PL/I, and assembler applications. It offers consistent and predictable results for language applications, independent of the language in which they are written.

This document supports z/OS (5650-ZOS).

Language Environment is the prerequisite runtime environment for applications generated with the following IBM compiler products:

- z/OS XL C/C++
- OS/390® C/C++
- C/C++ Compiler for MVS/ESA
- AD/Cycle C/370™ Compiler
- VisualAge for Java, Enterprise Edition for OS/390
- Enterprise COBOL for z/OS
- Enterprise COBOL for z/OS and OS/390
- COBOL for OS/390 & VM
- COBOL for MVS & VM (formerly COBOL/370)
- Enterprise PL/I for z/OS
- Enterprise PL/I for z/OS and OS/390
- VisualAge PL/I for OS/390
- PL/I for MVS & VM
- AD/Cycle PL/I for MVS & VM
- VS FORTRAN and FORTRAN IV (in compatibility mode)

Although not all compilers listed are currently supported, Language Environment supports the compiled objects that they created.

Language Environment supports, but is not required for, an interactive debug tool for debugging applications in your native z/OS environment. The interactive IBM Debug Tool is available with the latest release of the PL/I compiler or this product can be ordered separately for use with the IBM XL C/C++, COBOL, and PL/I compilers on z/OS. For more information, see the Debug Tool for z/OS home page (http://www.ibm.com/software/products/us/en/debugtool).

Language Environment supports, but is not required for, VS Fortran Version 2 compiled code (z/OS only).

Language Environment consists of the common execution library (CEL) and the runtime libraries for C/C++, COBOL, Fortran, and PL/I.

For more information on VisualAge® for Java™, Enterprise Edition for OS/390, program number 5655-JAV, see the product documentation.

This book introduces you to the Language Environment architecture, a system of constructs and interfaces that provides a common runtime environment and runtime services for all Language Environment-conforming programming language products (those products that adhere to Language Environment's common interface).

Language Environment is offered on z/OS.

The book contains an overview of Language Environment, descriptions of Language Environment's full program model, callable services, and a glossary of Language Environment terms. This is not a programming manual, but rather a conceptual introduction to Language Environment.

Language Environment Concepts Guide should be read by those who design systems installations and develop application programs. This high-level guide will show how best to plan for systems to support your enterprise.

Terms that may be new to you are *italicized* on their first use. Definitions of these terms can be found in "Language Environment glossary" on page 49.

Using your documentation

The publications provided with Language Environment are designed to help you:

- Manage the runtime environment for applications generated with a Language Environment-conforming compiler.
- Write applications that use the Language Environment callable services.
- Develop interlanguage communication applications.
- Customize Language Environment.
- Debug problems in applications that run with Language Environment.
- Migrate your high-level language applications to Language Environment.

Language programming information is provided in the supported high-level language programming manuals, which provide language definition, library function syntax and semantics, and programming guidance information.

Each publication helps you perform different tasks, some of which are listed in Table 1.

Table 1. How to use z/OS Language Environment publications

To ...	**Use ...**
Evaluate Language Environment	*z/OS Language Environment Concepts Guide*
Plan for Language Environment	*z/OS Language Environment Concepts Guide* *z/OS Language Environment Runtime Application Migration Guide*
Install Language Environment	*z/OS Program Directory*
Customize Language Environment	*z/OS Language Environment Customization*
Understand Language Environment program models and concepts	*z/OS Language Environment Concepts Guide* *z/OS Language Environment Programming Guide* *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*
Find syntax for Language Environment runtime options and callable services	*z/OS Language Environment Programming Reference*
Develop applications that run with Language Environment	*z/OS Language Environment Programming Guide* and your language programming guide

Table 1. How to use z/OS Language Environment publications (continued)

To ...	Use ...
Debug applications that run with Language Environment, diagnose problems with Language Environment	*z/OS Language Environment Debugging Guide*
Get details on runtime messages	*z/OS Language Environment Runtime Messages*
Develop interlanguage communication (ILC) applications	*z/OS Language Environment Writing Interlanguage Communication Applications* and your language programming guide
Migrate applications to Language Environment	*z/OS Language Environment Runtime Application Migration Guide* and the migration guide for each Language Environment-enabled language

Product information on the web

For information about the z/OS product and elements, see z/OS home page (http://www.ibm.com/systems/z/os/zos/).

For information about z/OS Language Environment, see z/OS Language Environment (http://www.ibm.com/systems/z/os/zos/features/lang_environment/).

z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

To find the complete z/OS library, including the z/OS Information Center, see z/OS Internet Library (http://www.ibm.com/systems/z/os/zos/bkserv/).

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html).
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS V2R1.0 Language Environment Concepts Guide
  SA38-0687-00
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (http://www.ibm.com/systems/z/support/).

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Overview

Today, enterprises need efficient, consistent, and less complex ways to develop quality applications and to maintain their existing inventory of applications. The trend in application development is to modularize and share code, and to develop applications on a workstation-based front end. Language Environment gives you a common environment for all Language Environment-conforming high-level language (HLL) products. A HLL is a programming language above the level of assembler language and below that of program generators and query languages.

In the past, programming languages also have had limited ability to call each other and behave consistently across different operating systems. This restriction has constrained those who wanted to use several languages in an application. Programming languages have had different rules for implementing data structures and condition handling, and for interfacing with system services and library routines.

Language Environment establishes a common runtime environment for all participating HLLs. It combines essential runtime services, such as routines for runtime message handling, condition handling, and storage management. All of these services are available through a set of interfaces that are consistent across programming languages. You can either call these interfaces yourself, or use language-specific services that call the interfaces. With Language Environment, you can use one runtime environment for your applications, regardless of the application's programming language or system resource needs.

Language Environment consists of:

- Basic routines that support starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling conditions.
- Common library services, such as math services and date and time services that are commonly needed by programs running on the system. These functions are supported through a library of callable services.
- Language-specific portions of the runtime library. Because many language-specific routines call Language Environment services, behavior is consistent across languages.

Figure 1 on page 2 shows the separate components that make up Language Environment. POSIX support is provided in the Language Environment base and in the C language-specific library.

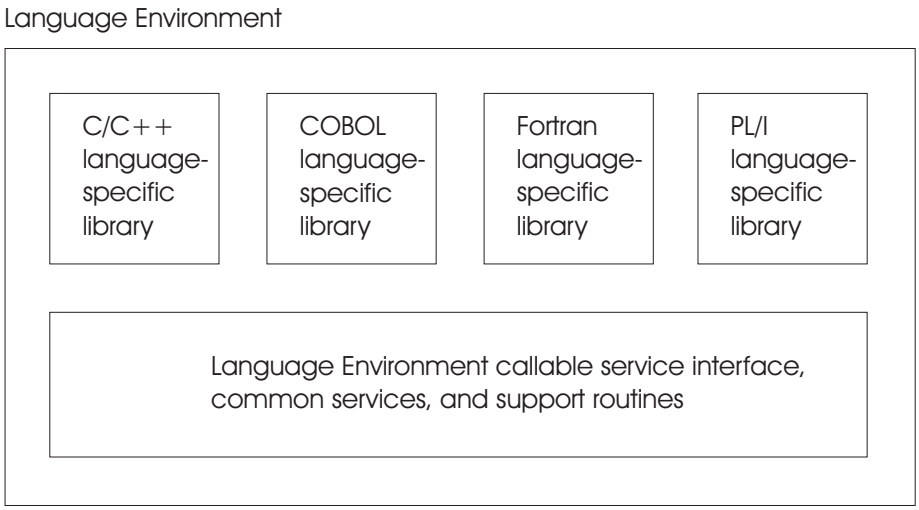

Figure 1. Components of Language Environment

z/OS Language Environment is the prerequisite runtime environment for applications that are generated with the following IBM compiler products:

- z/OS XL C/C++
- OS/390 C/C++
- C/C++ Compiler for MVS/ESA
- AD/Cycle C/370 Compiler
- VisualAge for Java, Enterprise Edition for OS/390
- Enterprise COBOL for z/OS
- Enterprise COBOL for z/OS and OS/390
- COBOL for OS/390 & VM
- COBOL for MVS™ & VM (formerly COBOL/370)
- Enterprise PL/I for z/OS
- Enterprise PL/I for z/OS and OS/390
- VisualAge PL/I for OS/390
- PL/I for MVS & VM
- AD/Cycle PL/I for MVS & VM
- VS FORTRAN and FORTRAN IV (in compatibility mode)

Although not all compilers listed are currently supported, Language Environment supports the compiled objects that they created.

Language Environment supports, but is not required for, VS Fortran Version 2 compiled code (OS/390 only).

In many cases, you can run compiled code generated from the previous versions of the above compilers. A set of assembler macros is also provided to allow assembler routines to run with Language Environment.

For more information on IBM VisualAge for Java, Enterprise Edition for OS/390, program number 5655-JAV, refer to the product documentation.

Figure 2 illustrates the common environment that Language Environment creates.

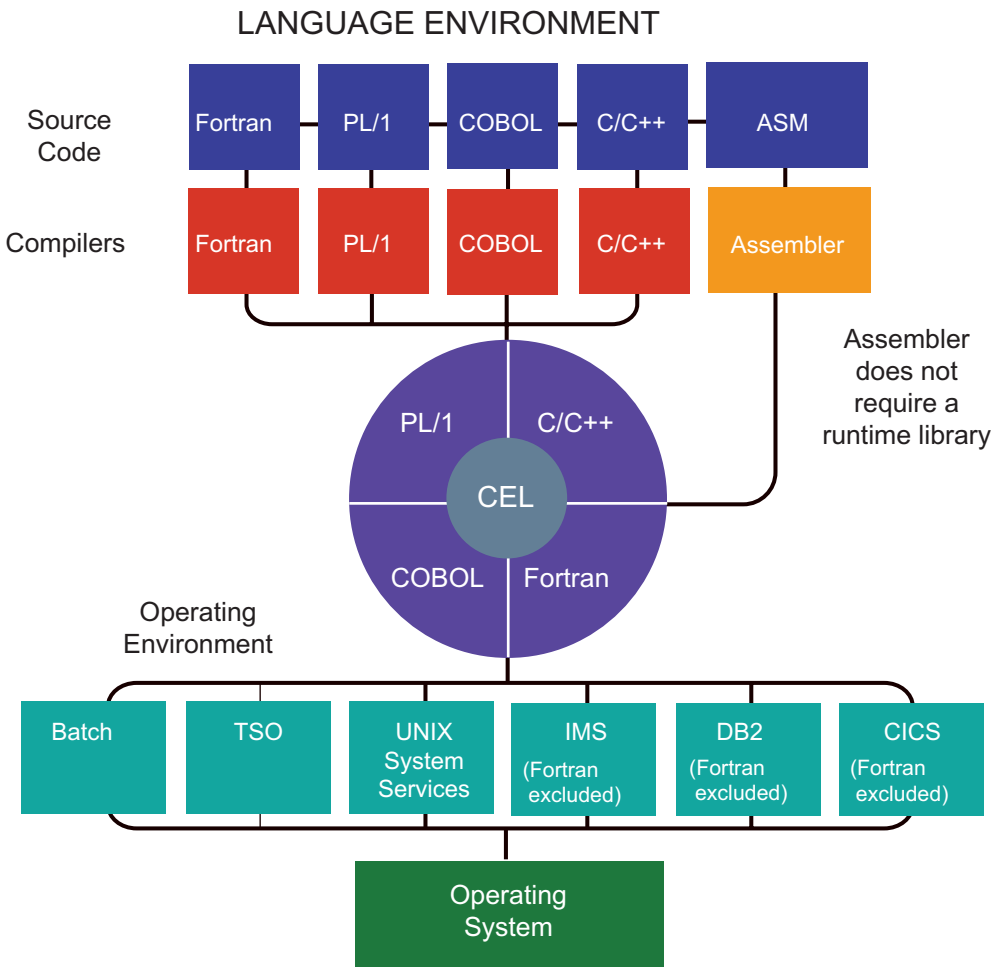

Figure 2. The common runtime environment

Figure 3 on page 4 illustrates the common environment that Language Environment creates for AMODE 64.

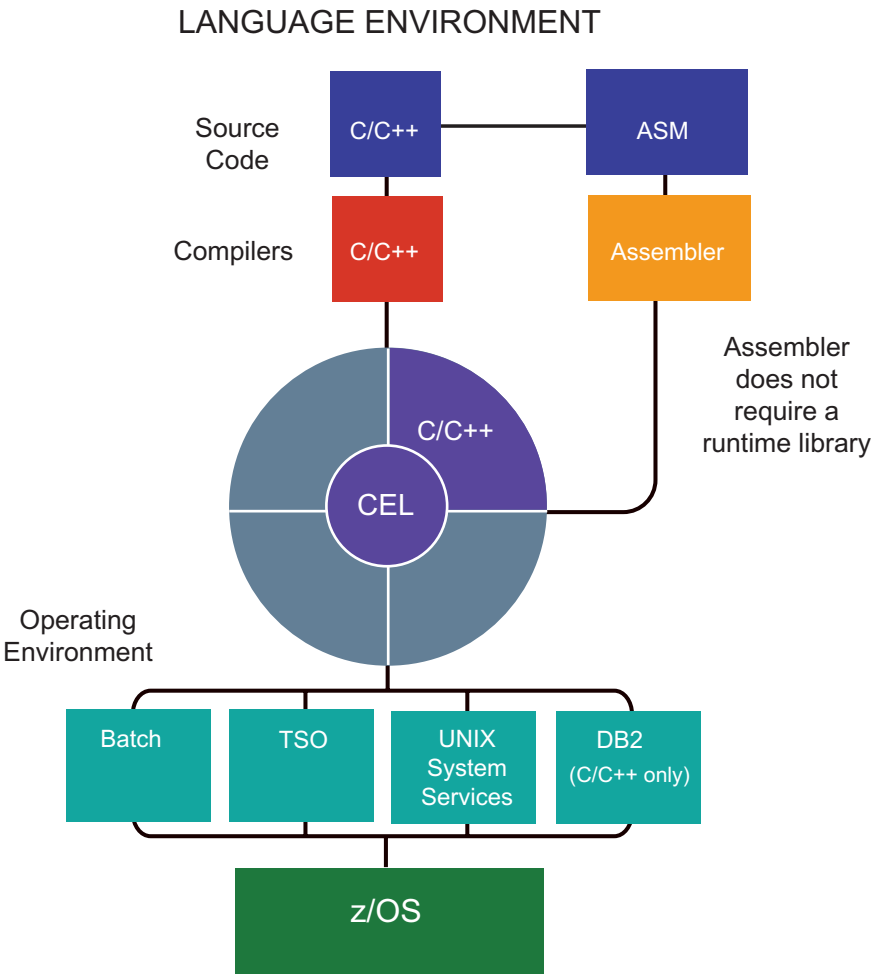

Figure 3. The common runtime environment for AMODE 64

For more information, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

Language Environment supports 64-bit addressing for applications that are written in C, C++, or Language Environment-conforming Assembler.

Before support for 64-bit addressing, Language Environment applications could be written in COBOL, PL/I, C, C++, Fortran, or Language Environment-conforming Assembler. These applications could run in either 24-bit addressing mode (AMODE 24) or 31-bit addressing mode (AMODE 31). Language Environment includes some support for compatibility between these two addressing modes. In AMODE 24, addresses are 24 bits in length, which allows access to virtual storage up to 16 Megabytes. This is often referred to as the *16-megabyte line.* AMODE 31 applications use addresses that are 31 bits in length, which allows access to virtual storage up to 2 gigabytes. This limit on 31-bit addressing is referred to as the *2-gigabyte bar.* Both of these terms can be shortened to the "line" or the "bar" when used in the context of addressable storage.

In the 64-bit addressing mode (AMODE 64) supported by Language Environment, addresses are 64 bits in length, which allows access to virtual storage up to 16 exabytes. While this is an extremely high address, there are a few very important facts to consider:

- Existing or new Language Environment applications that use AMODE 24 or AMODE 31 can continue to run without change. They run using the same Language Environment services that existed before 64-bit addressing was introduced, and these services will continue to be supported and enhanced.
- Language Environment applications that use AMODE 64 are not compatible with applications that use AMODE 24 or AMODE 31. The only means of communication between AMODE 64 and AMODE 24 or AMODE 31 applications is through mechanisms that can communicate across processes or address spaces. However, Language Environment applications that use AMODE 64 can run with existing applications that use AMODE 24 or AMODE 31 on the same physical System Z.
- Where necessary, there are new Language Environment runtime options to support AMODE 64 applications. The new runtime options primarily support of the new stack and heap storage that is located above the bar. All other existing runtime options continue to be supported and enhanced for AMODE 24 and AMODE 31 applications.

What you can do with Language Environment

Language Environment helps you create mixed-language applications and gives you a consistent method of accessing common, frequently used services. Building mixed-language applications is easier with Language Environment-conforming routines because Language Environment establishes a consistent environment for all languages in the application.

Common use of system resources gives you greater control

Language Environment provides the base for future IBM language library enhancements in the z/OS environment. Many system dependencies have been removed from Language Environment-conforming language products.

Because Language Environment provides a common library, with services that you can call through a common callable interface, the behavior of your applications will be easier to predict. Language Environment's common library includes common services such as messages, date and time functions, math functions, application utilities, system services, and subsystem support. The language-specific portions of Language Environment provide language interfaces and specific services that are supported for each individual language.

Language Environment is accessed through defined common calling conventions, described in Chapter 3, “Language Environment callable services,” on page 25.

Consistent condition handling simplifies error recovery

Language Environment establishes consistent condition handling for HLLs, debug tools, and assembler language routines. For languages with little or no condition handling function, like COBOL, Language Environment provides a user-controlled method that was not available before for predictable, robust error recovery. Language Environment condition handling honors single- and mixed-language semantics and is integrated with message handling services to provide you with specific information about each condition.

This language-independent condition handler, unlike some existing HLL condition semantics, is stack frame-based and delivers predictable behavior at a given stack

frame. Language Environment condition handling enables you to construct applications out of building blocks of modules and control which modules will handle certain conditions.

A complete description of Language Environment's condition handling model and message services is described in Chapter 2, “The model for Language Environment,” on page 11.

Language Environment protects your programming investment

Language Environment provides compatible support for existing HLL applications. Applications linked with the migration tools provided with libraries that predate Language Environment do not need to be linked with the Language Environment library routines. For more information, see *z/OS Language Environment Writing Interlanguage Communication Applications*. For mixed-language applications, however, relinking with Language Environment may be required if the application was not previously relinked using migration tools available with pre-Language Environment libraries. Routines compiled with the new Language Environment-conforming compilers can be mixed with old routines in an application. Thus, applications can be enhanced or maintained selectively, without recompiling the whole application when a change is made to a single routine. Some modifications of existing applications may be required. See *z/OS Language Environment Runtime Application Migration Guide* for more information.

ILC capability offers greater efficiency and flexibility

Language Environment eliminates incompatibilities among language-specific runtime environments. Routines call one another within one common runtime environment, eliminating the need for initialization and termination of a language-specific runtime environment with each call. This makes interlanguage communication (ILC) in mixed-language applications easier, more efficient, and more consistent.

This ILC capability also means that you can share and reuse code easily. You can write a service routine in the language of your choice (C/C++, COBOL, PL/I, or assembler) and allow that routine to be called from C/C++, COBOL, PL/I, or assembler applications. Similarly, vendors can write one application package in the language of their choice, and allow the application package to be called from C/C++, PL/I, and assembler routines or from Fortran or COBOL programs.

In addition, Language Environment lets you use the best language for any task. Some programming languages are better suited for certain tasks. Language Environment's improved interlanguage communication (ILC) allows the best language to be used for any given application task. Many programmers, each experienced in a different programming language, can work together to build applications with component routines written in a variety of languages. Language Environment's enhanced ILC allows you to build applications with component routines written in a variety of languages. The result is code that runs faster, is less prone to errors, and is easier to maintain.

Common dump puts all debugging information in one place

Language Environment provides a common dump for all conforming languages. The dump includes, in an easy-to-read format, a description of any relevant conditions and information on error location, variables, and storage.

With a common dump, you can locate precisely the module where an error occurred, saving you many hours of debugging, especially if your module is built with several languages. A common dump also allows programmers of differing language skills to collaborate effectively in determining the location of a problem that involves modules of different languages.

POSIX-conforming application support enhances code portability

The IEEE *Portable Operating System Interface* (POSIX) standard is a series of industry standards for code and user interface portability. POSIX support allows applications written for a UNIX-like operating system to be run on z/OS. C language programmers can access operating system services through a set of standard language bindings. C language programmers who install z/OS UNIX System Services (z/OS UNIX) and z/OS Language Environment can call C language functions defined in the POSIX standard from their C applications and can run applications that conform to ISO/IEC 9945-1:1990. [1] C language programmers with z/OS UNIX installed can also call a subset of the proposed programming interface for thread management (a subset of draft 6 of *POSIX.4a*). Through C interfaces, Language Environment functions conform to XPG4.2 specifications and are branded by X/Open.

Applications that call POSIX functions can perform limited ILC under Language Environment (see *z/OS Language Environment Writing Interlanguage Communication Applications* for details). In addition, C POSIX-conforming applications may use all Language Environment services.

For an overview of z/OS UNIX, see *z/OS Introduction and Release Guide*.

Locale callable services enhance the development of internationalized applications

Demand is steadily increasing in global markets for software products, and application developers are seeking to make their products available in multiple countries. While marketing their products globally, however, programmers must also make their applications function with the specific language and cultural conventions of the individual user's locale. With locale callable services, application developers can build programs that can be marketed globally, and still meet end users' needs to work with specific languages, cultures, and conventions.

Language Environment provides pre-defined locales, previously available to C/370 routines only, that your PL/I routines and COBOL programs can access at run time through the locale callable services. You can also create your own locales, or modify the IBM-supplied locales, using the C locale definition utility available with the C/C++ compiler.

While C routines can use the locale callable services, it is recommended that they use the equivalent native C library services instead for portability across platforms.

For a complete description of Language Environment locale support, see *z/OS Language Environment Programming Guide*.

1. ISO/IEC 9945-1:1990, which is also ANSI-IEEE 1003.1-1990, is based on the POSIX.1 standard.

Debug tool in your common environment

Language Environment supports Debug Tool for z/OS, an interactive source-level debugger. Debug Tool enables you to examine, monitor, and control the execution of Assembler, C, C++, COBOL, and PL/I programs on z/OS systems. The execution environments that it supports include batch, TSO, CICS, DB2®, DB2 stored procedures, IMS™, and UNIX System Services. Debug Tool offers additional productivity enhancements when used with the GUI provided in IBM Rational® Developer for System z®, WebSphere® Developer for System z, or WebSphere Developer Debugger for System z (all available separately).

Debug Tool also includes tools to help you identify OS/VS and VS COBOL II source code and to upgrade the code to Enterprise COBOL. In addition, it provides tools that can help you quickly identify and convert OS/VS COBOL code to ANSI 85 standard, as well as tools to help you determine how thoroughly your code has been tested.

For more information about Debug Tool for z/OS, see the Debug Tool for z/OS home page (http://www.ibm.com/software/products/us/en/debugtool).

IBM C/C++ productivity tools for OS/390

With the IBM C/C++ Productivity Tools for OS/390 product, you can expand your z/OS application development environment out to your workstation, while remaining close to your familiar host environment.

IBM C/C++ Productivity Tools for OS/390 include the following workstation-based tools to increase your productivity and code quality:

- A Performance Analyzer to help analyze, understand, and tune your C and C++ applications for improved performance. (References to the Performance Analyzer in this section refer to the Performance Analyzer included in the C/C++ Productivity Tools for OS/390 product.)
- A Distributed Debugger that allows you to debug C or C++ programs from the convenience of your workstation.
- A workstation editor to improve the productivity of your C and C++ source entry.
- Advanced online help, with full text search and hypertext topics as well as printable, viewable, and searchable Portable Document Format (PDF) documents.

In addition, IBM C/C++ Productivity Tools for OS/390 include the following host components:

- Debug Tool
- Host Performance Analyzer

Use the Performance Analyzer on your workstation to graphically display and analyze a profile of the running of your host z/OS C or C++ application. Use this information to time and tune your code so that you can increase the performance of your application.

Use the Distributed Debugger to debug your z/OS C/C++ application remotely from your workstation.

Set a break point with a click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

Use the workstation-based editor to quickly develop C and C++ application code that runs on z/OS. Context-sensitive help information is available to you when you need it.

Chapter 2. The model for Language Environment

This topic describes the Language Environment architecture, a system of user conventions, product conventions, and processing models that, when followed by HLL application programmers, provides a common, consistent runtime environment.

Models for program management, condition handling, message services, and storage management are outlined.

The Language Environment program management model

The Language Environment program management model provides a framework within which an application runs. It is the foundation of all of the component models—condition handling, runtime message handling, and storage management—that comprise the Language Environment architecture. The program management model defines the effects of programming language semantics in mixed-language applications and integrates transaction processing and multithreading.

Language Environment program management model terminology

Some terms used to describe the program management model are common programming terms; other terms are described differently in other languages. It is important that you understand the meaning of the terminology in a Language Environment context as compared to other contexts.

For more detailed definitions of these and other Language Environment terms, consult the "Language Environment glossary" on page 49.

General programming terms

Application program
A collection of one or more programs cooperating to achieve particular objectives, such as inventory control or payroll.

Environment
In Language Environment, normally a reference to the runtime environment of HLLs at the enclave level.

Language Environment terms and their HLL equivalents:

Routine
In Language Environment, refers to either a procedure, function, or subroutine.

Equivalent HLL terms: COBOL—program; C/C++—function; PL/I—procedure, BEGIN block.

Enclave
The enclave defines the scope of HLL semantics. In Language Environment, a collection of routines, one of which is named as the main routine. The enclave contains at least one thread.

Equivalent HLL terms: COBOL—run unit, C/C++—program, consisting of a main C function and its subfunctions, PL/I—main procedure and its subroutines, and Fortran—program and its subroutines.

Process
The highest level of the Language Environment program management model. A process is a collection of resources, both program code and data, and consists of at least one enclave.

Thread
An execution construct that consists of synchronous invocations and terminations of routines. The thread is the basic runtime path within the Language Environment program management model, and is dispatched by the system with its own runtime stack, instruction counter, and registers. Threads may exist concurrently with other threads.

Terminology for data

Automatic data
Data that does not persist across calls. It is allocated with the same value on entry and reentry into a routine.

External data
Data that can be referenced by multiple routines and data areas. External data is known throughout an enclave.

Local data
Data that is known only to the routine in which it is declared.

Equivalent HLL terms: C/C++—local data, COBOL—WORKING-STORAGE data items and LOCAL-STORAGE data items, PL/I—data declared with the PL/I INTERNAL attribute.

Program management

Program management defines the program execution constructs of an application, and the semantics associated with the integration of various components management of such constructs.

Three entities (*process*, *enclave,* and *thread*) are at the core of the Language Environment program management model. They are described in this section.

Refer to Figure 4 on page 13 as you read the following discussion about processes, enclaves, and threads. This figure illustrates the simplest form of the Language Environment program management model and how resources such as storage are managed.

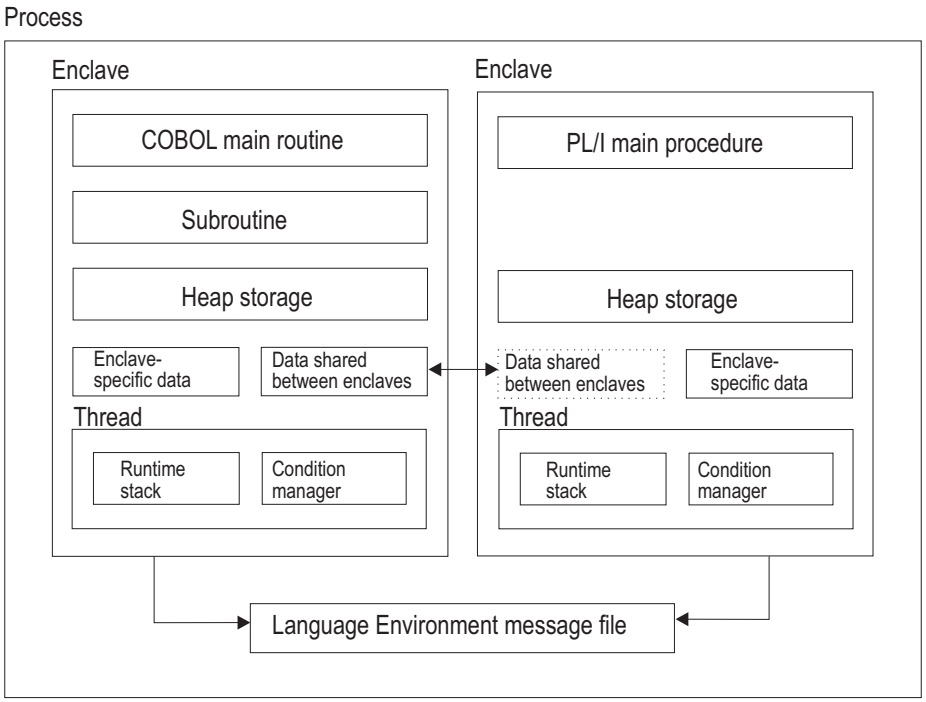

Figure 4. Language Environment resource ownership

Processes

The highest level component of the Language Environment program model is the process. A process consists of at least one enclave and is logically separate from other processes. Processes do not share storage and are independent of and equal to each other; they are not hierarchically related.

Language Environment generally does not allow language file sharing across enclaves nor does it provide the ability to access collections of externally stored data.

However, in PL/I, SYSPRINT can be shared across enclaves if all the code in all the enclaves has been compiled either with PL/I for MVS & VM or with Enterprise PL/I for z/OS, but not both.

The Language Environment message file also can be shared across enclaves, since it is managed at the process level. The Language Environment message file contains messages from all routines running within a process, making it a useful central location for messages that are generated during run time.

Processes can create new processes and communicate to each other by using Language Environment-defined communication for such things as indicating when a created process has been terminated.

Enclaves

A key feature of the program management model is the *enclave*, a collection of the routines that make up an application. The enclave is the equivalent of any of the following:

- A *run unit*, in COBOL

- A *program*, consisting of a *main C function* and its *subfunctions*, in C and C++
- A *main procedure* and all of its *subroutines*, in PL/I
- A and its *subroutines*, in Fortran

The enclave consists of one main routine and zero or more subroutines. (However, a POSIX application might not have a main routine active at a given time.) The main routine is the first to execute in an enclave; all subsequent routines are named as subroutines.

Characteristics of the enclave

The enclave logically owns resources normally associated with the running of a program. Some resources are owned directly, such as heap storage; some are owned indirectly, such as the runtime stack, which is owned by a thread. Heap storage, the runtime stack, and threads are discussed in the following sections.

Heap storage is shared among all routines in an enclave and can be allocated by a routine in one language and be freed by a routine in another language. For a discussion on stack and heap storage, see "Language Environment storage management model" on page 21.

The enclave defines the scope—how far the semantic effects of language statements reach—of the language semantics for its component routines, just as a COBOL run unit defines the scope of semantics of a COBOL program.

The enclave defines the following in a Language Environment-conforming application:

- Scope of shared external data, such as COBOL EXTERNAL data and PL/I external data
- Scope of external files, such as COBOL EXTERNAL files [2]
- Scope of the effect of language statements, for example, STOP-like constructs, such as STOP RUN in COBOL or other terminating mechanisms
- Lifetime of heap storage, in its last-used state

Threads

Each enclave consists of at least one thread, the basic instance of a particular routine. A thread is created during enclave initialization with its own runtime stack, which keeps track of the thread's execution, as well as a unique instruction counter, registers, and condition-handling mechanisms. Each thread represents an independent instance of a routine running under an enclave's resources.

Threads share all of the resources of an enclave. A thread can address all storage within an enclave. All threads are equal and independent of one another and are not related hierarchically. A thread can create a new enclave. Because threads operate with unique runtime stacks, they can run concurrently within an enclave and allocate and free their own storage. Because they may execute concurrently, threads can be used to implement parallel processing applications and event-driven applications.

Figure 5 on page 15 illustrates the full Language Environment program model, with its multiple processes, enclaves, and threads.

2. The sharing of files across languages is not permitted in z/OS Language Environment.

As Figure 5 shows, each process is within its own address space. An enclave consists of one main routine, with any number of subroutines. A main routine might not be active at all times in a POSIX application, if the thread in which the main routine executes terminates before the other threads it created.

External data is available only within the enclave where it resides; notice that even though the external data may have identical names in different enclaves, the external data is unique to the enclave. The scope of external data is the enclave. The threads can create enclaves, which can create more threads, and so on.

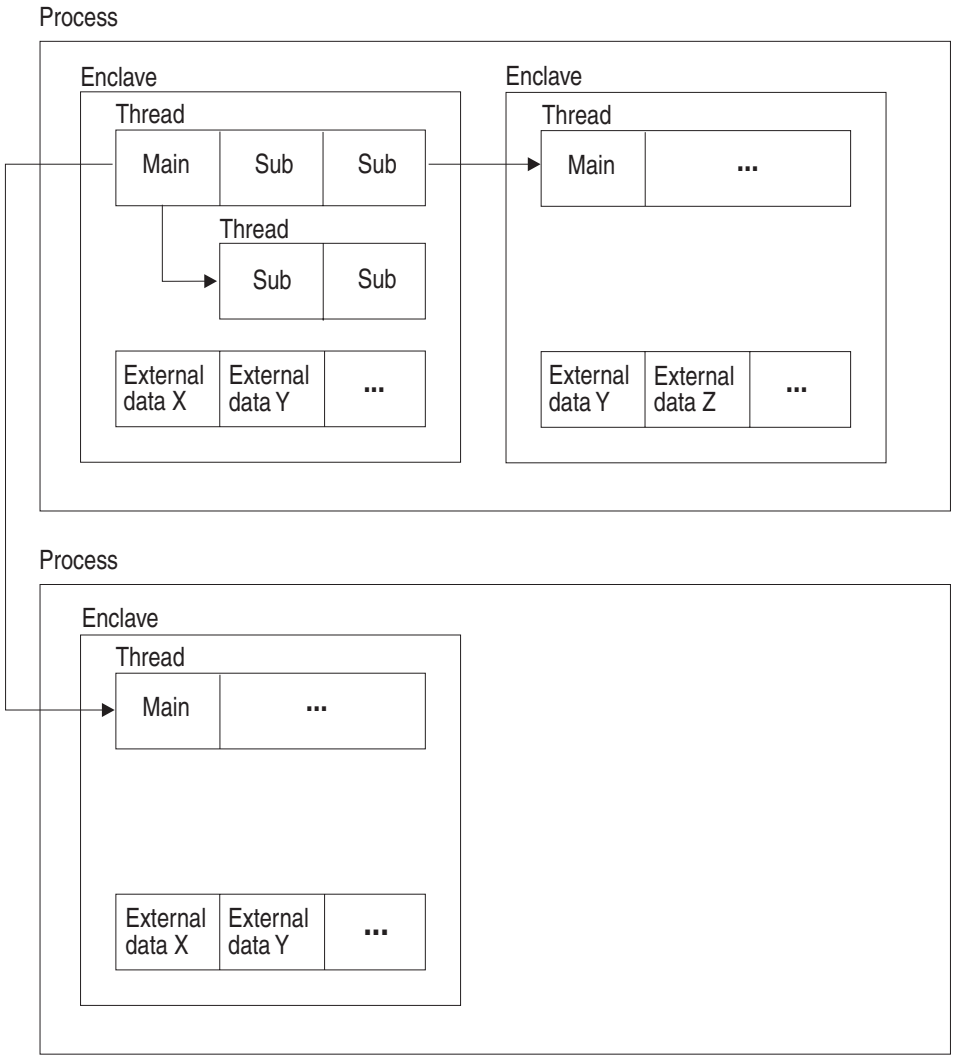

Figure 5. Language Environment program management

Language Environment condition-handling model

For single- and mixed-language applications, the Language Environment runtime library provides a consistent and predictable condition-handling facility. It does not replace current HLL condition handling, but instead allows each language to respond to its own unique environment as well as to a mixed-language environment.

Language Environment condition management gives you the flexibility to respond directly to conditions by providing callable services to signal conditions and to interrogate information about those conditions. It also provides functions for error diagnosis, reporting, and recovery.

Language Environment condition handling is based on the *stack frame*, an area of storage that is allocated when a routine runs and that represents the history of execution of that routine. It can contain automatic variables, information on program linkage and condition handling, and other information. Using the stack frame as the model for condition handling allows conditions to be handled in the stack frame in which they occur. This allows you to tailor condition handling according to a specific routine, rather than handle every possible condition that could occur within one global condition handler.

A unique feature of Language Environment condition handling is the condition token. The token is a 12-byte data type that contains an accumulation of information about each condition. The information can be returned to the user as a feedback code when calling Language Environment callable services. It can also be used as a communication vehicle within the runtime environment.

Serviceability is improved with interactive problem control system (IPCS) exits.

Condition-handling terminology

For more detailed definitions of these and other Language Environment terms, see the "Language Environment glossary" on page 49.

Condition
: Any change to the normal programmed flow of a program. In Language Environment, a condition can be generated by an event that has historically been called an exception, interruption, or condition.

Condition handler
: A routine invoked by Language Environment that responds to conditions in an application. Condition handlers are registered through the CEEHDLR callable service, or provided by the language libraries, by such constructs as PL/I ON statements.

Condition token
: In Language Environment, a data type consisting of 12 bytes with structured fields that indicate various aspects of a condition, including severity, associated message number, and information that is specific to a given instance of the condition.

Feedback code
: A condition token value used to communicate information when using the Language Environment callable services.

Resume cursor
: Contains the address where execution resumes after a condition is handled. Initially, it will be the point in the application where a condition occurred when it is first reported to Language Environment.

Stack frame
: The physical representation of the activation of a routine. The stack frame is allocated on a last in, first out (LIFO) basis and can contain automatic variables, information on program linkage and condition handling, and other information.

A stack frame is conceptually equivalent to a dynamic save area (DSA) in PL/I, or a save area in assembler.

Condition-handling model description

The Language Environment condition handler is based on a stack frame model. A stack frame is an area of storage that can contain automatic variables, information on program linkage and condition handling, and other information. The stack frame is allocated using Language Environment-managed storage, either HEAP or STACK, depending on the language being used. It is created through any of the following,

- A function call in C or C++
- Entry into a compile unit in COBOL
- Entry into a procedure or begin block in PL/I
- Entry into an ON-unit in PL/I

Each routine adds a unique stack frame, in a LIFO manner, to the Language Environment storage, either HEAP or STACK. User-written condition handlers (registered through CEEHDLR) are associated with each stack frame. In addition, HLL handling semantics can affect the processing conditions at each stack frame. For an illustration of the Language Environment runtime stack and its divisions into stack frames, see Figure 6.

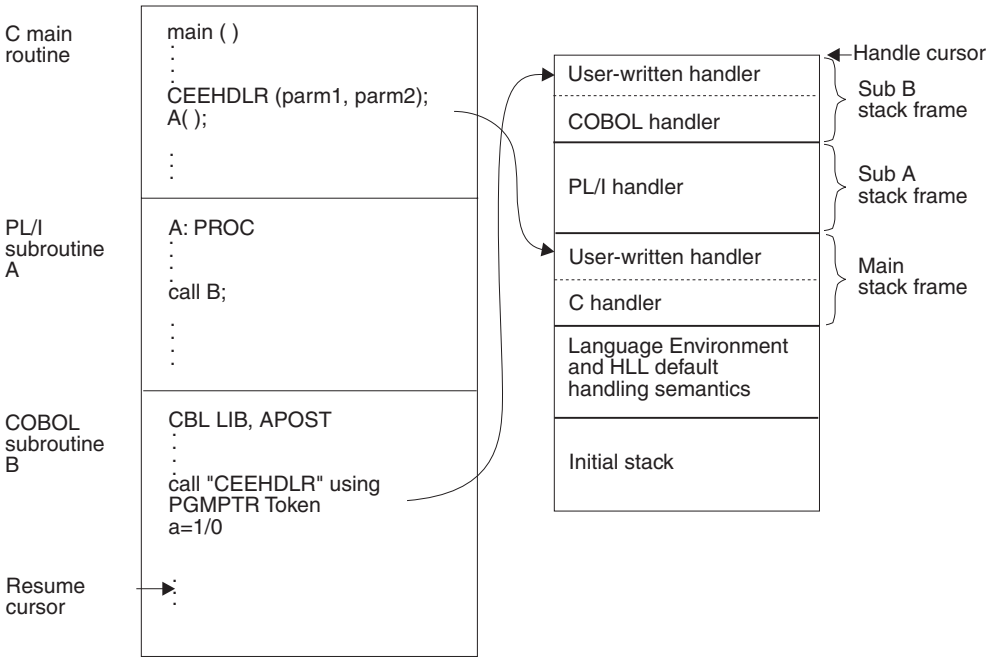

Figure 6. Condition-handling stack configuration

Each Language Environment user condition handler is explicitly registered through the callable service CEEHDLR or through the USRHDLR runtime option. Language-defined handling mechanisms are registered through language-provided constructs, such as the PL/I ON statement or the C `signal()` function. When a routine returns to its caller, its stack frame is removed from the stack and the associated handlers are automatically unregistered. Semantics associated with a routine are honored; for example, PL/I semantics on a return specify that any ON-units within a routine will be unregistered. If the USRHDLR runtime option is used, the user-written condition handler is registered at stack frame 0.

A condition is signaled within Language Environment as a result of one of the following occurrences:

- A hardware-detected interrupt
- An operating system-detected exception
- A condition generated by Language Environment callable services
- A condition explicitly signaled within a routine

The first three types of conditions are managed by Language Environment and signaled if appropriate. The last may be signaled by user-written code through a call to the service CEESGL or signaled by HLL semantics such as SIGNAL in PL/I or `raise` in C.

When a condition is signaled, whether by a user routine, by Language Environment in response to an operating system or hardware detected condition, or by a callable service, Language Environment directs the appropriate condition handlers in the stack frame to handle the condition. Condition handling proceeds first with user-written condition handlers in the queue, if present, then with any HLL-specific condition handlers, such as a PL/I ON-unit or a C signal handler, that may be established. The process continues for each frame in the stack, from the most recently allocated to the least recently allocated.

If a condition remains unhandled after the stack is traversed, the condition is handled by either Language Environment or by the default semantics of the language where the condition occurred.

How conditions are represented

A condition token is used to communicate information about a condition to Language Environment, message services, callable services, and routines. The token is a 12-byte data type with fields that indicate the following information about a condition:

- Severity of a condition
- Associated message number
- Facility ID: This field identifies the owner of the condition (Language Environment, Language Environment component, or user-specified). It is also used to identify a file containing message text that is unique for the condition.
- Instance specific information: This field is created if the condition requires that data or text be inserted into a message, for example, a variable name. This field also contains qualifying data, which can be used to specify data (input or output) to be used when a routine resumes processing after a condition occurs.

How condition tokens are created and used

If the condition is detected by the operating system or by the hardware, Language Environment will automatically build the condition token and signal the condition. With Language Environment callable services, you can create a condition token with corresponding message or data inserts and then signal the condition to the application running within Language Environment by returning the token.

When used in Language Environment callable services, the entire condition token represents a value called the feedback code. You can include a feedback parameter to Language Environment callable services, and check the result of the call; or, in PL/I and C, you can omit the feedback parameter, and any errors in the call are signaled to you. Figure 7 on page 19 shows how condition tokens are created and used.

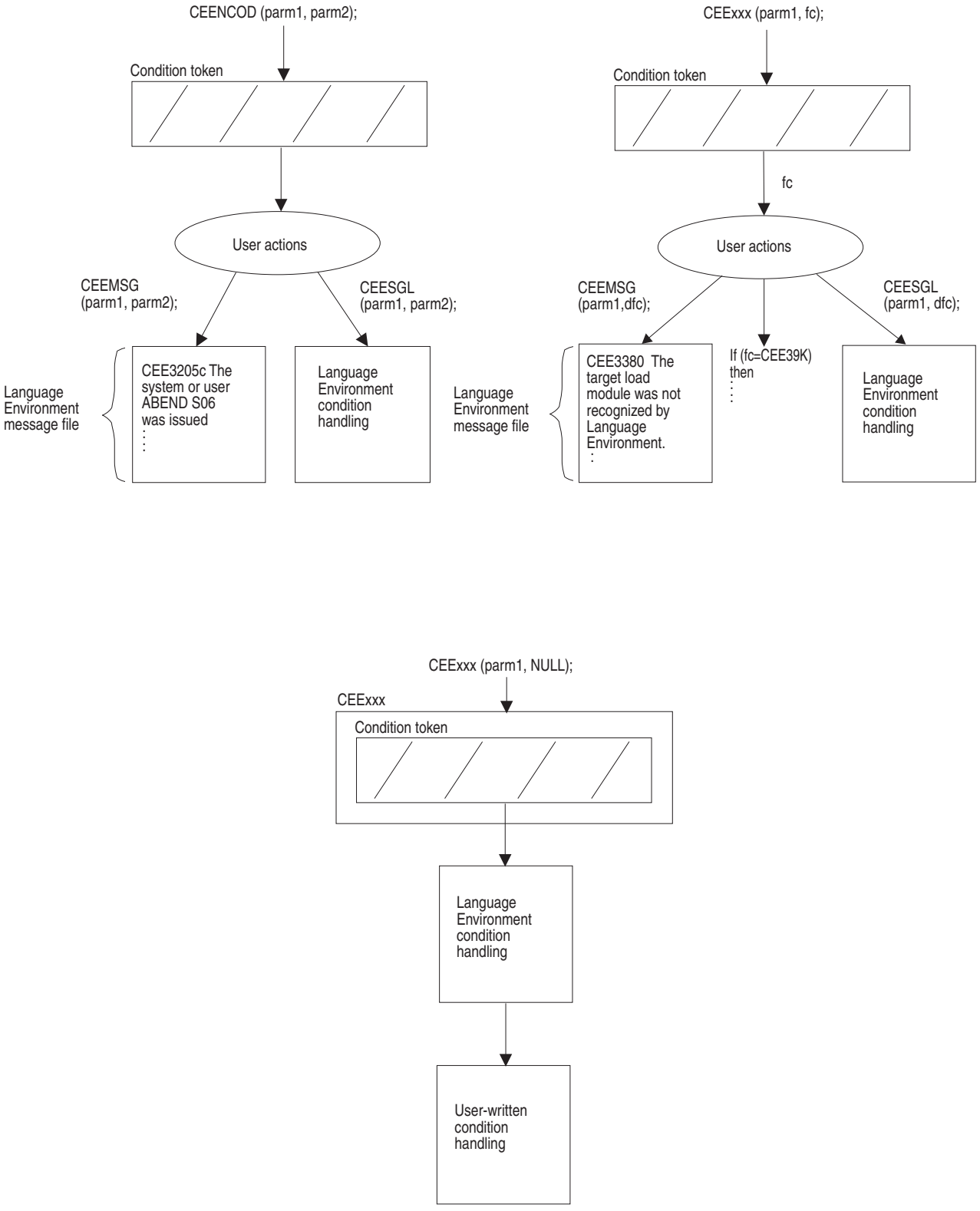

Figure 7. How condition tokens are created and used

Condition-handling responses

Conditions are responded to in one of the following ways:

- *Resume* terminates condition handling and transfers control usually to the location immediately following the point where the condition occurred.

  A resume cursor points to the place where a routine should resume; it can be moved by the callable service CEEMRCR to point to another resume point.
- *Percolate* defers condition handling for an unchanged condition. Condition handling continues at the next condition handler.
- *Promote* is similar to percolate in that it passes the condition on to the next condition handler; however, it transforms a condition to another condition, one with a new meaning. Condition handling then continues, this time with a new type of condition.

Runtime dump service provides information in one place

The Language Environment callable service CEE3DMP dumps the runtime environment of Language Environment into one easily readable report. CEE3DMP can be called directly from an application to produce a dump that is formatted for printing. Depending on the options you choose, the dump report may contain information on conditions, tracebacks, variables, control blocks, stack and heap storage, file status and attributes, and language-specific information. The report can also be requested with the TERMTHDACT runtime option when a program terminates due to an unhandled condition.

Serviceability is improved with a traceback section in CEEDUMP.

Language Environment message handling model and national language support

A set of common message handling services that create and send runtime informational and diagnostic messages is provided by Language Environment.

With the message handling services, you can use the condition token that is returned from a callable service or from some other signaled condition, format it into a message, and deliver it to a defined output device or to a buffer.

National language support

Messages can be formatted according to national language support specifications for the following languages:

- Mixed-case American English (ENU)
- Uppercase American English (UEN)
- Japanese (JPN)

National language support callable services allow you to set a national language that affects the language of the error messages and the names of the day, week, and month. It also allows you to change the country setting, which affects the default date format, time format, currency symbol, decimal separator character, and thousands separator.

Language Environment storage management model

Common storage management services are provided for all Language Environment-conforming programming languages; Language Environment controls stack and heap storage used at run time. It allows single- and mixed-language applications to access a central set of storage management facilities, and offers a multiple-heap storage model to languages that do not now provide one. The common storage model removes the need for each language to maintain a unique storage manager and avoids the incompatibilities between different storage mechanisms.

Condition-handling terminology

For more detailed definitions of these and other Language Environment terms, see the "Language Environment glossary" on page 49.

Condition
Any change to the normal programmed flow of a program. In Language Environment, a condition can be generated by an event that has historically been called an exception, interruption, or condition.

Condition handler
A routine invoked by Language Environment that responds to conditions in an application. Condition handlers are registered through the CEEHDLR callable service, or provided by the language libraries, by such constructs as PL/I ON statements.

Condition token
In Language Environment, a data type consisting of 12 bytes with structured fields that indicate various aspects of a condition, including severity, associated message number, and information that is specific to a given instance of the condition.

Feedback code
A condition token value used to communicate information when using the Language Environment callable services.

Resume cursor
Contains the address where execution resumes after a condition is handled. Initially, it will be the point in the application where a condition occurred when it is first reported to Language Environment.

Stack frame
The physical representation of the activation of a routine. The stack frame is allocated on a last in, first out (LIFO) basis and can contain automatic variables, information on program linkage and condition handling, and other information.

A stack frame is conceptually equivalent to a dynamic save area (DSA) in PL/I, or a save area in assembler.

Stack storage

In Language Environment, a runtime stack, or stack storage, is automatically created when a thread is created, and freed when the thread terminates. When a thread is created, Language Environment allocates an initial stack, which can have stack increments added to it as needed. Users can specify the sizes of the initial stack and additional stack increments; they can also tune the stack for better performance. For more information, see*z/OS Language Environment Programming Guide*.

For AMODE 64 support, users can specify a stack size above the bar, and can specify the maximum stack size. For more information, see the information on stack and heap storage in *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

In AMODE 31, each stack segment is allocated separately. In AMODE 64, the maximum possible stack size can be specified. A contiguous block of storage is allocated above the bar and each segment is unguarded as needed.

Heap storage

Heap storage can be allocated and freed in no particular order. (Stack storage, in contrast, is allocated when a routine is entered and freed when the routines ends.) Language Environment provides multiple heaps that may be dynamically created and discarded by using Language Environment callable services. Language Environment's heap storage is reliable because it provides a level of isolation and prevents common errors such as attempting to free a heap element that has already been freed.

Heap storage is shared among all program units and all threads in an enclave. Allocated heap storage remains allocated until it is explicitly freed by a thread or until the enclave terminates. Heap storage is typically controlled by the programmer through Language Environment runtime options and callable services.

Heap storage consists of an initial heap segment that is allocated when the first heap element is allocated by a call to CEEGTST. The Language Environment storage manager allocates heap increments as previously allocated segments become full.

Figure 8 illustrates heap storage.

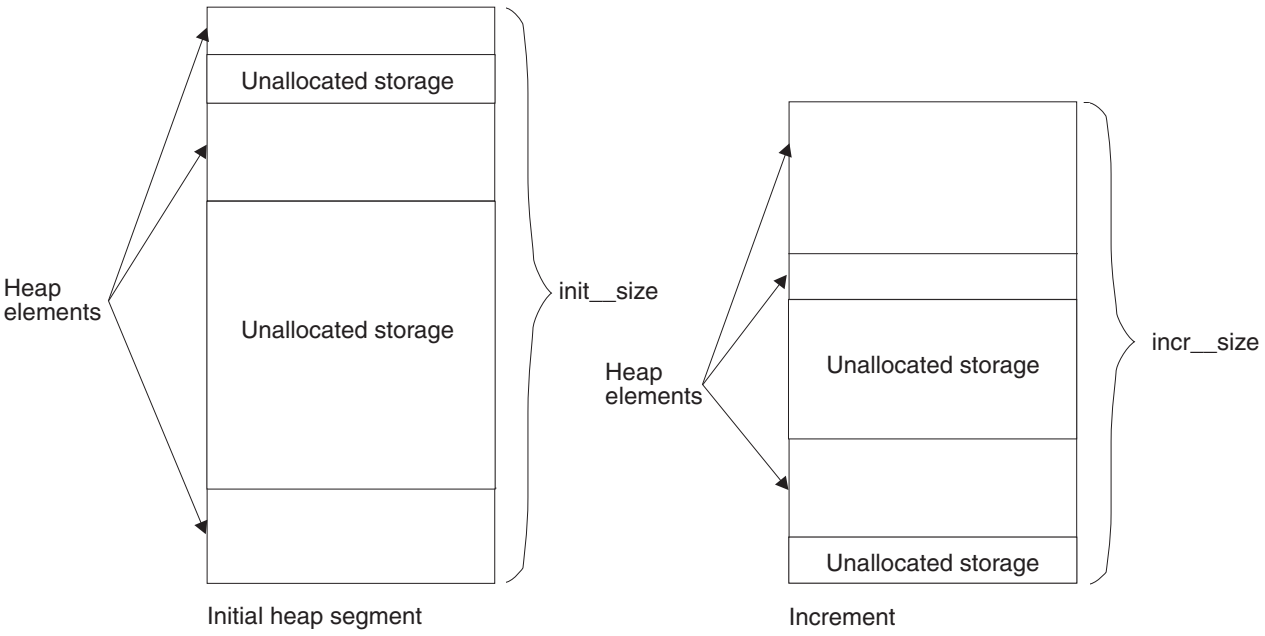

Figure 8. Language Environment heap storage

Storage management options

Storage report

You can write a storage report using the runtime option RPTSTG. The report summarizes all heap and stack activity, including total amount of storage used, number of heap elements allocated and freed, number of operating system calls performed, and recommended heap and stack sizes. Proper setting of heap and stack sizes can significantly improve performance by reducing the number of operating system calls made to allocate and free storage.

Storage option

In Language Environment, the runtime option STORAGE may be used to automatically initialize all heap and stack storage to a specified character. This is useful as a debugging aid to find references to uninitialized program variables.

For AMODE 64 support, you must specify MEMLIMIT. The overall storage above the bar is controlled in MVS. For more information, see *z/OS MVS Programming: Extended Addressability Guide*.

Chapter 3. Language Environment callable services

This topic gives an overview of Language Environment callable services and the common calling procedure required to invoke them from C/C++, COBOL, PL/I, Fortran, and assembler.

This common set of callable services is designed to supplement your programming language's intrinsic capability. For example, COBOL application developers will find Language Environment's consistent condition handling services especially useful. All languages can benefit from the rich set of Language Environment common math services, as well as the date and time services.

The listed callable services are for AMODE 31 only. For AMODE 64, none of the application writer interfaces (AWIs) will be supported in their present form. There may be C functions that provide similar functionality for some of the AWIs. A few nonstandard C functions have been added to provide the functionality of some of the AWIs. For details, see *z/OS XL C/C++ Runtime Library Reference*.

Language Environment callable services are divided into the following groups:
- Communicating Conditions Services
- Condition Handling Services
- Date and Time Services
- Dynamic Storage Services
- General Callable Services
- Initialization/Termination Services
- Locale Callable Services
- Math Services
- Message Handling Services
- National Language Support Services

Direct invocation of Language Environment callable services is not supported from Fortran. However, support is provided to use callable services using a Fortran library subroutine service. For more information, see *Language Environment for MVS & VM Fortran Run-Time Migration Guide*. Alternatively, a Fortran program can call another Language Environment-enabled high-level language or an assembler program that can invoke a Language Environment callable service.

Language-specific services, including those that call Language Environment callable services, are documented in the language publications.

Language Environment calling conventions

Language Environment services can be invoked by HLL library routines, other Language Environment services, and user-written HLL calls. In many cases, services will be invoked by HLL library routines, as a result of a user-specified function, such as a COBOL intrinsic function.

Language Environment-conforming languages exhibit consistent behavior because language functions call Language Environment services. For example, C `malloc()` and PL/I ALLOCATE each directly or indirectly call CEEGTST to obtain storage.

The sections below show examples of the syntax used to invoke Language Environment callable services.

Invoking callable services from C

In C, invoke a Language Environment callable service (with feedback code) using the following syntax:

```
#include <leawi.h>
main ()
{
 CEESERV(parm1, parm2, ... parmn, fc);
 }
```

Figure 9. Sample invocation of a callable service from C

leawi.h is a header file shipped with Language Environment that contains declarations of Language Environment callable services and OMIT_FC, which is used to explicitly omit the feedback code parameter.

```
#include <leawi.h>
main ()
{
 CEESERV(parm1, parm2, ... parmn, OMIT_FC);
 }
```

Figure 10. Omitting the feedback code when calling a service from C

Invoking callable services from COBOL

In COBOL, invoke a Language Environment callable service using the following syntax:

```
01 Feedback.
COPY CEEIGZCT
⋮
CALL "CEESERV" USING parm1 parm2 ... parmn fc
```

Figure 11. Sample invocation of a callable service from COBOL

CEEIGZCT is an include file shipped with Language Environment that contains declarations of Language Environment symbolic feedback codes.

You can omit the feedback code parameter in COBOL for OS/390 & VM and COBOL for MVS & VM as shown in the following syntax:

```
01 Feedback.
COPY CEEIGZCT
CALL "CEESERV" USING parm1 parm2 ... parmn OMITTED
```

Figure 12. Omitting the feedback code when calling a service from COBOL

Invoking callable services from PL/I

In PL/I, invoke a Language Environment callable service (with feedback code) using the following syntax:

```
%INCLUDE CEEIBMAW
:
CALL CEESERV (parm1, parm2, ... parmn, fc);
```

Figure 13. Sample invocation of a callable service from PL/I

CEEIBMAW is an include file shipped with Language Environment that contains declarations of Language Environment callable services.

PL/I allows you to omit arguments when invoking callable services. To do so, code an asterisk (*) in place of the argument.

```
%INCLUDE CEEIBMAW
:
CALL CEESERV (parm1, parm2, ... parmn, *);
```

Figure 14. Omitting the feedback code when calling a service from PL/I

Invoking callable services from assembler

In assembler, invoke a Language Environment callable service (with feedback code) using the syntax shown below:

```
        LA    R1,PLIST
        L     R15,=V(CEESERV)
        BALR  R14,R15
        CLC   FC(12),CEE000   Check if feedback code is zero
        BNE   ER1             If not, branch to error routine
:
PLIST   DS    0D
        DC    A(PARM1)
:
                              Parms 2 through n

DC      A(FC+X'80000000')     Feedback code as last parm

PARM1    DC   F'5'            Parm 1
:
                              Parms 2 through n

FC      DS    12C             Feedback code as last parm
CEE000  DC    12X'00'         Good feedback code
```

Figure 15. Sample invocation of a callable service from assembler

Assembler allows you to omit arguments when invoking callable services. To do so, place an X'80000000' in the last parameter address slot if *fc* is the last parameter, or X'00000000' in the corresponding address slot if *fc* is not the last parameter.

```
        LA    R1,PLIST
        L     R15,=V(CEESERV)
        BALR  R14,R15
⋮
PLIST   DS    0D
        DC    A(PARM1)
⋮
                                Parms 2 through n
DC      A(X'80000000')          Omitted feedback code in last slot

PARM1    DC   F'5'              Parm 1
⋮
                                Parms 2 through n
```

Figure 16. Omitting the feedback code when calling a service from assembler

Language Environment callable services

Naming conventions of the callable services are as follows:

- Services starting with CEE are intended to be cross-system consistent; they operate on z//OS systems.
- Services starting with CEE3 are services that exploit unique characteristics of z/OS systems..

Table 2. Language Environment callable services

Service name	Description
Communicating conditions services	
CEEDCOD (Decompose a condition token)	Decomposes an existing condition token.
CEENCOD (Construct a condition token)	Dynamically constructs a condition token. The condition token communicates with message services, condition management, Language Environment callable services, and user applications.
Condition handling services	
CEE3CIB (Return pointer to condition information block)	Given a condition token that is passed to a user-written condition handler, CEE3CIB returns a pointer to the condition information block associated with a condition. Allows access to detailed information about the subject condition during condition handling.
CEE3GRN (Get name of routine that incurred condition)	Obtains the name of the routine that is running when a condition is raised. If there are nested conditions, the most recently signaled condition is used.
CEE3GRO (Return offset)	Returns the offset of the location within the most current Language Environment-conforming routine where a condition occurred.
CEE3SPM (Query and modify Language Environment hardware condition enablement)	Allows the user to manipulate the program mask by enabling or masking hardware interrupts.
CEE3SRP (Set resume point)	Sets a resume point within user application code to resume from a Language Environment user condition handler.
CEEGQDT (Retrieve q_data_Token)	Retrieves the q_data token from the instance-specific information (ISI) to be used by user condition handlers.
CEEHDLR (Register a user condition handler)	Registers a user condition handler for the current stack frame.
CEEHDLU (Unregister a user condition handler)	Unregisters a user condition handler for the current stack frame.

Table 2. Language Environment callable services (continued)

Service name	Description
CEEITOK (Return initial condition token)	Returns the initial condition token for the current condition.
CEEMRCE (Resume user routine)	Resumes execution of a user routine at the location that is established by CEE3SRP.
CEEMRCR (Move resume cursor relative to handle cursor)	Moves the resume cursor. You can either move the resume cursor to the call return point of the routine that registered the executing condition handler, or move the resume cursor to the caller of the routine that registered the executing condition handler.
CEESGL (Signal a condition)	Signals a condition to the Language Environment condition manager. It also can be used to provide qualifying data and create an instance-specific information (ISI) field. The ISI contains information that is used by the Language Environment condition manager to identify and react to conditions.
Date and time services	
CEECBLDY (Convert date to Cobol Lilian format)	Converts a string representing a date into a COBOL Lilian date format. The COBOL Lilian date format represents a date as the number of days since 31 December 1600.
CEEDATE (Convert Lilian date to character format)	Converts a number representing a Lilian date to a date written in character format. The output is a character string such as `1992/07/25`.
CEEDATM (Convert seconds to character timestamp)	Converts a number representing the number of seconds since 00:00:00 14 October 1582 to a character format. The format of the output is a character string, such as "1992/07/26 20:37:00."
CEEDAYS (Convert date to Lilian format)	Converts a string representing a date into a Lilian format. The Lilian format represents a date as the number of days since 14 October 1582, the beginning of the Gregorian calendar.
CEEDYWK (Calculate day of week from Lilian date)	Calculates the day of the week on which a Lilian date falls. The day of the week is returned to the calling routine as a number between 1 and 7.
CEEGMT (Get current Greenwich mean time)	Returns the current Greenwich Mean Time (GMT) as both a Lilian date and as the number of seconds since 00:00:00 14 October 1582. These values are compatible with those generated and used by the other Language Environment date and time services.
CEEGMTO (Get offset from Greenwich mean time to local time)	Returns values to the calling routine which represent the difference between the local system time and Greenwich Mean Time.
CEEISEC (Convert integers to seconds)	Converts separate binary integers representing year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 14 October 1582. Use CEEISEC instead of CEESECS when the input is in numeric format rather than character format.
CEELOCT (Get current local time)	Returns the current local time in three formats: • Lilian date (the number of days since 14 October 1582) • Lilian timestamp (the number of seconds since 00:00:00 14 October 1582) • Gregorian character string (in the form YYYYMMDDHHMISS999)
CEEQCEN (Query the century window)	Queries the century within which Language Environment assumes 2-digit year values lie. Use it with CEESCEN when it is necessary to save and restore the current setting.
CEESCEN (Set the century window)	Sets the century where Language Environment assumes 2-digit year values lie. Use it with CEEDAYS or CEESECS when you process date values that contain 2-digit years (for example, in the YYMMDD format), or when the Language Environment default century interval does not meet the requirements of a particular application.

Table 2. Language Environment callable services (continued)

Service name	Description
CEESECI (Convert seconds to integers)	Converts a number representing the number of seconds since 00:00:00 14 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond. Use CEESECI instead of CEEDATM when the output is needed in numeric format rather than character format.
CEESECS (Convert timestamp to number of seconds)	Converts a string representing a timestamp into a number representing the number of seconds since 00:00:00 14 October 1582. This service makes it easier to do time arithmetic, such as calculating the elapsed time between two timestamps.
CEEUTC (Get Coordinated Universal Time)	CEEUTC is an alias of CEEGMT.
Dynamic storage services	
CEECRHP (Create new additional heap)	Defines additional heaps. The heaps defined by CEECRHP can be used just like the Language Environment initial heap (heap id of 0). However, the entire heap created by CEECRHP may be quickly freed with a single call to the CEEDSHP (discard heap) service.
CEECZST (Reallocate storage)	Changes the size of a previously allocated storage element while preserving its contents. Reallocation of a storage element is accomplished by allocating a new storage element of a new size and copying the contents of the old element to the new element.
CEEDSHP (Discard heap)	Discards an entire heap that you created previously with a call to CEECRHP.
CEEFRST (Free heap storage)	Frees storage previously allocated by CEEGTST. It can be used to free both large and small blocks of storage efficiently because freed storage is retained on a free chain instead of being returned to the operating system.
CEEGTST (Get heap storage)	Allocates storage from a heap whose ID you specify. It can be used to efficiently acquire both large and small blocks of storage.
General services	
CEE3DLY (Suspend processing of an active enclave in seconds)	Suspends processing of an active enclave for a specified number of seconds up to a maximum of one hour.
CEE3DMP (Generate dump)	Generates a dump of the runtime environment of Language Environment and of the member language libraries. The dump can be modified to selectively include such information as number and contents of enclaves and threads, traceback of all routines on a call chain, file attributes, and variable, register, and storage contents.
CEE3INF (Provide enclave information)	Provides current Language Environment information about the enclave.
CEETDLI (Invoke IMS)	Invokes IMS.
CEE3RPH (Set report heading)	Sets the heading that is displayed at the top of the storage or runtime options report. Language Environment generates the storage report when the RPTSTG(ON) runtime option is specified, and the options report when the RPTOPTS(ON) runtime option is specified.
CEE3USR (Set or query user area fields)	Sets or queries one of two 4-byte fields in the enclave data block known as the user area fields. The user area fields are associated with an enclave and are maintained on an enclave basis. A user area might be used by vendor or applications to store a pointer to a global data area or keep a recursion counter.
CEEDLYM (Suspend processing of an active enclave in milliseconds)	Suspends processing of an active enclave for a specific number of milliseconds up to a maximum of one hour.

Table 2. Language Environment callable services (continued)

Service name	Description
CEEENV (Query, set, or delete environment variables)	Allows for querying, setting, and deleting of environment variables.
CEEGPID (Retrieve the Language Environment version and platform ID)	Retrieves the Language Environment version ID and the platform ID of the version and platform of Language Environment that is in use for processing the currently active condition.
CEEGTJS (Retrieves the value of an exported JCL symbol)	Retrieves and returns to the caller the length of an exported JCL symbol value or the symbol value.
CEERAN0 (Calculate uniform random numbers)	Generates a sequence of uniform pseudo-random numbers between 0 and 1 using the multiplicative congruential method with a user-specified seed.
CEETEST (Invoke Debug Tool)	Invokes a debug tool, such as Debug Tool.
Initialization and termination services	
CEE3ABD (Terminate enclave with an abend)	Requests Language Environment to terminate the enclave via an abend. The abend can be issued either with or without cleanup.
CEE3AB2 (Add a reason code to an abend)	Supports the addition of a reason code to the ABEND. This enhances CEE3ABD to allow for more control of diagnostic information collection.
CEE3GRC (Get the enclave return code)	Retrieves the current value of the user enclave return code.
CEE3PRM (Query parameter string)	Returns to the calling routine the parameter string that was specified at invocation of the program. The returned parameter string contains only user parameters. If no user parameters are available, a blank string is returned.
CEE3PR2 (Supports longer parameter lists)	Supports longer parameter lists.
CEE3SRC (Set the enclave return code)	Modifies the user enclave return code. The value set is used in the calculation of the final enclave return code at enclave termination.
Locale callable services	
CEEFMON (Format monetary string)	Converts numeric values to monetary strings.
CEEFTDS (Format time and date into character string)	Converts time and date specifications into a character string.
CEELCNV (Query locale numeric conventions)	Returns information about the LC_NUMERIC and LC_MONETARY categories of the locale.
CEEQDTC (Query locale date and time conventions)	Queries the locale's date and time conventions.
CEEQRYL (Query active locale environment)	Allows the calling routine to query the current locale.
CEESCOL (Compare collation weight of two strings)	Compares two character strings that are based on the collating sequence that is specified in the LC_COLLATE category of the locale.
CEESETL (Set locale operating environment)	Allows an enclave to establish a global operating environment. An enclave's National Language operating environment determines the behavior of character collation, character classification, date and time formatting, numeric punctuation, and message response.
CEESTXF (Transform string characters into collation weights)	Transforms each character in a character string into its collation weight and returns the length of the transformed string.
Mathematical services	
Language Environment math services are scalar routines. *x* is a data type variable.	
CEESxABS	Absolute value
CEESxACS	Arccosine

Table 2. Language Environment callable services (continued)

Service name	Description
CEESxASN	Arcsine
CEESxATH	Hyperbolic arctangent
CEESxATN	Arctangent
CEESxAT2	Arctangent x/y
CEESxCJG	Conjugate of complex
CEESxCOS	Cosine
CEESxCSH	Hyperbolic cosine
CEESxCTN	Cotangent
CEESxDIM	Positive difference
CEESxDVD	Floating complex divide
CEESxERF	Error function
CEESxEXP	Exponential (base e)
CEESxGMA	Gamma function
CEESxIMG	Imaginary part of complex
CEESxINT	Truncation
CEESxLGM	Log gamma function
CEESxLG1	Logarithm base 10
CEESxLG2	Logarithm base 2
CEESxLOG	Logarithm base e
CEESxMLT	Floating complex multiply
CEESxMOD	Modular arithmetic
CEESxNIN	Nearest integer
CEESxNWN	Nearest whole number
CEESxSGN	Transfer of sign
CEESxSIN	Sine
CEESxSNH	Hyperbolic sine
CEESxSQT	Square root
CEESxTAN	Tangent
CEESxTNH	Hyperbolic tangent
CEESxXPx	Exponentiation
Message handling services	
CEECMI (Store and load message insert data)	Stores the message insert data and loads the address of that data into the instance-specific information (ISI) field that is associated with the condition that is being processed after optionally creating an ISI.
CEEMGET (Get a message)	Retrieves, formats, and stores a message in a buffer for manipulation or output by the caller.
CEEMOUT (Dispatch a message)	Dispatches a message to a destination which you specify.
CEEMSG (Get, format, and dispatch a message)	Gets/formats/dispatches a message corresponding to an input condition token received from a callable service. You can use this service to print a message after a call to any Language Environment service that returns a condition token.
National Language Support services	

Table 2. Language Environment callable services (continued)

Service name	Description
CEE3CTY (Set default country)	Allows the calling routine to change or query the current national country setting. The country setting affects the date format, the time format, the currency symbol, the decimal separator character, and the thousands separator.
CEE3LNG (Set national language)	Allows the calling routine to change or query the current national language. The national languages can be recorded on a LIFO national language stack. Changing the national language changes the languages of error messages, the names of the days of the week, and the names of the months.
CEE3MCS (Obtain default currency symbol)	Returns the default currency symbol for the country specified.
CEE3MDS (Obtain default decimal separator)	Returns the default decimal separator for the country specified.
CEE3MTS (Obtain default thousands separator)	Returns the default thousands separator for the country specified.
CEEFMDA (Obtain default date format)	Returns to the calling routine the default date picture string for a specified country.
CEEFMDT (Obtain default date and time format)	Returns to the calling routine the default date and time picture strings for the country specified.
CEEFMTM (Obtain default time format)	Returns to the calling routine the default time picture string for the country specified.

Chapter 4. Sample routines

Sample routines that demonstrate several aspects of Language Environment are included.

- Assembler routine, "Sample assembler routine"
- C/C++ routine, "Sample C/C++ routine"
- C routine with POSIX functions, "Sample C routine with POSIX functions" on page 36
- COBOL program, "Sample COBOL program" on page 38
- PL/I routine, "Sample PL/I routine" on page 39

Sample assembler routine

```
* ================================================================
*
* Shows a simple main assembler routine that brings up the environment,
* returns with a return code of 0, modifier of 0, and prints a
* message in the main routine.
*
* ================================================================
MAIN     CEEENTRY PPA=MAINPPA
*
*
         LA       1,PARMLIST
         L        15,=V(CEEMOUT)
         BALR     14,15
*
*   Terminate the Language Environment environment and return to the caller
*
         CEETERM  RC=0,MODIFIER=0
* ================================================================
*              CONSTANTS AND WORKAREAS
* ================================================================
PARMLIST DC    AL4(STRING)
         DC    AL4(DEST)
         DC    X'80000000'        Omitted feedback code
*
STRING   DC    AL2(STRLEN)
STRBEGIN DC    CL19'In the main routine'
STRLEN   EQU   *-STRBEGIN
DEST     DC    F'2'
MAINPPA  CEEPPA                   Constants describing the code block
         CEEDSA                   Mapping of the dynamic save area
         CEECAA                   Mapping of the common anchor area
         END   MAIN               Nominate MAIN as the entry point
```

Sample C/C++ routine

This routine demonstrates the following Language Environment callable services:

- CEEMOUT—Dispatch a message
- CEELOCT—Get current time
- CEEDATE—Convert Lilian date to character format

```
#include <leawi.h>
#include <string.h>
main ()
{
  _FEEDBACK   fbcode;           /* fbcode for all callable services */
```

```
/*********************************************************************/
/* Parameters passed to CEEMOUT.  Typedefs found in leawi.h.         */
/*********************************************************************/
  _VSTRING    msg;
  _INT4       destination;
/*********************************************************************/
/* Parameters passed to CEELOCT.  Typedefs found in leawi.h.         */
/*********************************************************************/
  _INT4       lildate;
  _FLOAT8     lilsecs;
  _CHAR17     greg;
/*********************************************************************/
/* Parameters passed to CEEDATE.  Typedefs found in leawi.h.         */
/*********************************************************************/
  _CHAR80  str_date;
  _VSTRING pattern;
/*********************************************************************/
/* Starting and ending messages                                      */
/*********************************************************************/
  _CHAR80  startmsg  = "Callable service example starting (C/370).";
  _CHAR80  endingmsg = "Callable service example ending (C/370).";

/*********************************************************************/
/* Start execution. Print the first message.                         */
/*********************************************************************/
  destination = 2;
  strcpy( msg.string, startmsg );
  msg.length  = strlen( msg.string );
  CEEMOUT ( &msg, &destination, &fbcode );
/*********************************************************************/
/* Get the local date and time, format it, and print it out.         */
/*********************************************************************/
  CEELOCT ( &lildate, &lilsecs, greg, &fbcode );
  strcpy  ( pattern.string,\
          "Today is Wwwwwwwwwwwwz, Mmmmmmmmmz ZD, YYYY." );
  pattern.length = strlen( pattern.string );
  memset ( msg.string , ' ' , 80 );
  CEEDATE ( &lildate, &pattern, msg.string, &fbcode );
  msg.length = 80;
  CEEMOUT ( &msg, &destination, &fbcode );
/*********************************************************************/
/* Say goodbye.                                                      */
/*********************************************************************/
  strcpy  ( msg.string, endingmsg );
  msg.length = strlen( msg.string );
  CEEMOUT ( &msg, &destination, &fbcode );
}
```

Sample C routine with POSIX functions

This C routine creates multiple threads by using POSIX functions.

```
#pragma longname
#define _POSIX_SOURCE
#define _OPEN_THREADS
#pragma runopts (POSIX(ON))
#include <leawi.h>
#include <types.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

void * CEPSXT1(void *);

main()
```

```
{
pthread_t CEPSXT1_pid[3];

  int status[2], i, j[2], rc, count=0;

   fprintf(stderr,"\n   Creating two threads..............\n");
   fflush(stderr);
   for(i=0; i<2; i++)
   {
     j[i] = i+1;
     rc = pthread_create(&CEPSXT1_pid[i], NULL, &CEPSXT1, &j[i])
     if (rc)
     {
       fprintf(stderr, "Thread creation unsuccessful;Error: %d",errno);
       fprintf(stderr, "pthread_create() returns %d ",rc);
       fflush(stderr);
       exit(-1);
     }
     else
       fprintf(stderr,"Thread %d created\n",j[i]);
   }
   for(i=0; i<2; i++)
   {
     j[i] = i+1;
     if (!(rc = pthread_join(CEPSXT1_pid[i],(void*) &status[i])))
     {
       if (status[i] == 1)
         count++;
     }
     else
     {
       fprintf(stderr,"pthread_join failed for thread %d\n",j[i]);
       fflush(stderr);
       exit(-1);
     }
   } if (count == 2)
       fprintf(stderr,"\n*****  SUCCESS  *****\n");
     else
       fprintf(stderr,"\n*****   ERROR   *****\n");
     fflush(stderr);
     pthread_exit(0);
}
void * CEPSXT1(void *arg)
{
 int status=0, success=0;
 div_t ans;
 char path = '/';
 int i, rc;

 i = *((int *)arg);

 fprintf(stderr,"\n Call POSIX access() function in Thread %d",i);
 fflush(stderr);
 if (access(path, F_OK) == 0)
  fprintf(stderr,"\nPOSIX access() function succeeds in Thread %d\n",i);
 else
   fprintf(stderr, "Error generated by call to access() is %d", errno);

 fflush(stderr);

 status=1;

 fprintf(stderr,"*****    Thread %d completed         *****\n", i);
 fflush(stderr);
 pthread_exit( (void*) status);
}
```

Sample COBOL program

This program demonstrates the following Language Environment callable services:

- CEEMOUT—Dispatch a message
- CEELOCT—Get current time
- CEEDATE—Convert Lilian date to character format

```
*****************************************************************
    *  This program demonstrates the following Language           *
    *  Environment callable                                       *
    *  services : CEEMOUT, CEELOCT, CEEDATE                       *
    *****************************************************************
    *****************************************************************
    **          I D          D I V I S I O N                     ***
    *****************************************************************
     Identification Division.
     Program-id.    AWIXMP.
    *****************************************************************
    **          D A T A      D I V I S I O N                     ***
    *****************************************************************
     Data Division.
     Working-Storage Section.
    *****************************************************************
    **  Declarations for the local date/time service.
    *****************************************************************
     01   Feedback.
     COPY CEEIGZCT
      02   Fb-severity      PIC 9(4) Binary.
      02   Fb-detail        PIC X(10).
     77   Dest-output       PIC S9(9) Binary.
     77   Lildate           PIC S9(9) Binary.
     77   Lilsecs           COMP-2.
     77   Greg              PIC X(17).
    *****************************************************************
    **  Declarations for messages and pattern for date formatting.
    *****************************************************************
     01   Pattern.
      02                    PIC 9(4) Binary Value 45.
      02                    PIC X(45) Value
          "Today is Wwwwwwwwwwwwz, Mmmmmmmmmz ZD, YYYY.".

     77   Start-Msg         PIC X(80) Value
          "Callable Service example starting.".

     77   Ending-Msg        PIC X(80) Value
          "Callable Service example ending.".

     01 Msg.
       02 Stringlen         PIC S9(4) Binary.
       02 Str               .
        03                  PIC X Occurs 1 to 80 times
                                 Depending on Stringlen.
    *****************************************************************
    **          P R O C      D I V I S I O N                     ***
    *****************************************************************
     Procedure Division.
     000-Main-Logic.
         Perform 100-Say-Hello.
         Perform 200-Get-Date.
         Perform 300-Say-Goodbye.
         Stop Run.
    **
    ** Setup initial values and say we are starting.
    **
     100-Say-Hello.
         Move 80 to Stringlen.
         Move 02 to Dest-output.
```

```
           Move Start-Msg to Str.
           CALL "CEEMOUT" Using Msg   Dest-output Feedback.
           Move Spaces to Str.        CALL "CEEMOUT" Using Msg Dest-output Feedback.
       **
       ** Get the local date and time and display it.
       **
        200-Get-Date.
           CALL "CEELOCT" Using Lildate Lilsecs     Greg      Feedback.
           CALL "CEEDATE" Using Lildate Pattern     Str       Feedback.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
           Move Spaces to Str.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
       **
       ** Say Goodbye.
       **
        300-Say-Goodbye.
           Move Ending-Msg to Str.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
        End program AWIXMP.
```

Sample PL/I routine

This routine demonstrates the following Language Environment callable services:

- CEEMOUT—Dispatch a message
- CEELOCT—Get current time
- CEEDATE—Convert Lilian date to character format

```
/* Declarations for callable services */
   %INCLUDE  CEEIBMAW;
   %INCLUDE  CEEIBMCT;

/* feedback code for all callable services*/
   dcl 01 fc FEEDBACK;
/*************************************************************/
/** Parameters passed to CEEMOUT.                           **/
/**                                                         **/
/*************************************************************/
   dcl startmsg CHAR80
         init('Callable service example starting (PL/I)');
   dcl endmsg CHAR80
         init('Callable service example ending (PL/I)');
   dcl strmsg CHAR80;
   dcl destination real fixed binary ( 31,0 );
/*************************************************************/
/** Parameters passed to CEELOCT.                           **/
/**                                                         **/
/*************************************************************/
   dcl lildate real fixed binary ( 31,0 );
   dcl lilsecs real float decimal ( 16 );
   dcl greg character ( 17 );
/*************************************************************/
/** Parameters for CEEDATE.                                 **/
/**                                                         **/
/*************************************************************/
   dcl pattern CHAR80;
   dcl chrdate CHAR80 init ((80)' ');
/*************************************************************/
/** Start execution.  Print the first message.              **/
/**                                                         **/
/*************************************************************/
   destination = 2;
   call CEEMOUT ( startmsg , destination , fc );
   IF  ¬ FBCHECK( fc, CEE000)  THEN  DO;
      DISPLAY( 'CEEMOUT failed with msg ' || fc.MsgNo );
      STOP;
```

```
      END;
 /*************************************************************/
 /** Get the local date and time. Format it, and print it    **/
 /** out.                                                    **/
 /*************************************************************/
    call CEELOCT ( lildate , lilsecs , greg , fc );
    IF  ¬ FBCHECK( fc, CEE000)  THEN  DO;
       DISPLAY( 'CEELOCT failed with msg ' || fc.MsgNo );
       STOP;
       END;
    pattern = 'Today is Wwwwwwwwwwwz, Mmmmmmmmmz, ZD, YYYY.';
    call CEEDATE ( lildate , pattern , chrdate , fc );
    IF  ¬ FBCHECK( fc, CEE000)  THEN  DO;
       DISPLAY( 'CEEDATE failed with msg ' || fc.MsgNo );
       STOP;
       END;

    strmsg = chrdate;
    call CEEMOUT ( strmsg , destination , fc );
    IF  ¬ FBCHECK( fc, CEE000)  THEN  DO;
       DISPLAY( 'CEEMOUT failed with msg ' || fc.MsgNo );
       STOP;
       END;

/*************************************************************/
 /** Say good bye.                                          **/
 /**                                                        **/
 /*************************************************************/
    call CEEMOUT ( endmsg , destination , fc );
    IF  ¬ FBCHECK( fc, CEE000)  THEN  DO;
       DISPLAY( 'CEEMOUT failed with msg ' || fc.MsgNo );
       STOP;
       END;

 end;
```

Appendix. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually

exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS Language Environment.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml (http://www.ibm.com/legal/copytrade.shtml).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Language Environment glossary

This glossary defines technical terms and abbreviations used in z/OS Language Environment documentation. If you do not find the term you are looking for, refer to the index of the appropriate Language Environment publication or view IBM Glossary of Computing Terms, located at:

http://www.ibm.com/software/globalization/terminology/ This glossary includes terms and definitions from: *Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*, ISO/EIC 9945-1: 1990, IEEE Std 1003.1-1990, copyright 1992 by The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017. These terms are identified by [POSIX.1].

abend Abnormal end of application.

absolute value
The magnitude of a real number regardless of its algebraic sign.

active routine
The currently executing routine.

actual argument
The Fortran term for the data passed to a called routine at the point of call. See also *dummy argument*.

additional floating point registers (AFP)
For IEEE support, 12 additional floating point registers, for a total of 16 floating-point registers.

additional heap
A Language Environment heap created and controlled by a call to CEECRHP. See also *below heap*, *anywhere heap*, and *initial heap*.

addressing mode
An attribute that refers to the address length that a routine is prepared to handle upon entry. Addresses may be 24 or 31 bits long.

address space
Domain of addresses that are accessible by an application.

AFP See *additional floating-point registers (AFP)*.

aggregate
A structured collection of data items that form a single data type. Contrast with *scalar*.

AIB Application interface block.

ALLOCATE command
In MVS, the TSO command that serves as the connection between a file's logical name (the ddname) and the file's physical name (the data set name).

American National Standard Code for Information Interchange (ASCII)
The code developed by the American National Standards Institute (ANSI) for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

AMODE
Provided by the linkage editor, the attribute of a load module that indicates the addressing mode in which the load module should be entered.

AMODE 31
Addressing mode 31.

AMODE 64
Addressing mode 64.

anywhere heap
The Language Environment heap controlled by the ANYHEAP runtime option. It contains library data, such as Language Environment control blocks and data structures not normally accessible from user code. The anywhere heap may reside above 16M. See also *below heap*, *additional heap*, *initial heap*.

APAR Authorized program analysis report.

application
A collection of one or more routines cooperating to achieve particular objectives.

application interface block (AIB)
IMS interface between an application and an IMS database.

application program
A collection of software components used to perform specific types of work on a computer, such as a program that does inventory control or payroll.

argument
1) An expression used at the point of a call to specify a data item or aggregate to be passed to the called routine. 2) The data passed to a called routine at the point of call or the data received by a called routine. See also *actual argument* and *dummy argument*.

array An aggregate that consists of data objects, each of which may be uniquely referenced by subscripting.

array element
A data item in an array.

ASCII American National Standard Code for Information Interchange.

Asian date format
In this book, Asian date format refers to the era picture strings associated with the Japanese or other era. Era picture strings begin with a less than character (<) and end with a greater than character (>). The characters inside are either capital Js or Cs.

assembler
Translates symbolic assembler language into binary machine language. The High Level Assembler is an IBM licensed program.

assembler user exit
A routine to tailor the characteristics of an enclave prior to its establishment. The name of the routine is CEEBXITA.

async safe
An application is able to mask off asynchronous signals when it is working with critical data or structures. The application can request to process the asynchronous signals when it has finished updated the critical data or structure.

atexit list
A list of actions specified in the C `atexit()` function that occur at normal program termination.

authorized program analysis report (APAR)
A request for correction of a problem caused by a defect in a current unaltered release of a program.

automatic call
The process used by the linkage editor to resolve external symbols left undefined after all the primary input has been processed. See also *automatic call library*.

automatic call library
Contains load modules or object modules that are to be used as secondary input to the linkage editor to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library may be:

- Libraries containing object modules, with or without linkage editor control statements
- Libraries containing load modules
- The library containing Language Environment runtime routines (SCEELKED) (SCEELKED and SAFHFORT)

automatic conversion
For Enhanced ASCII functionality, the automatic conversion of text data from EBCDIC to ASCII, or from ASCII to EBCDIC, as part of using internationalized applications developed on (or for) ASCII platforms and ported to z/OS platforms. See also *file tag* and *coded character set ID (CCSID).*

automatic data
Data for a routine that is automatically allocated when the routine is called and automatically freed when the routine returns. Automatic data does not persist from one call of the routine to the next.

automatic library call
Automatic call. See also *automatic call library*.

automatic storage
Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

AWI Application writer interface.

background process
A process that is a member of a background process group. [POSIX.1]

background process group
Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. [POSIX.1]

base The core product, upon which features may be separately ordered and installed.

batch Pertaining to activity involving little or no user action. Contrast with *interactive*.

below heap
The Language Environment heap controlled by the BELOWHEAP runtime option, which contains library data, such as Language Environment control block and data structures not normally accessible from user code. Below heap always resides below 16M. See also *anywhere heap, initial heap, additional heap*.

BFP See *binary floating point (BFP).*

binary floating point
For IEEE, binary floating point registers.

binder
The DFSMS component that processes the output of the language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA operating system.

breakpoint
A place in a program, usually specified by a command or a condition, where execution may be interrupted and control given to the workstation user or to a specified debug tool program.

buffer An area of storage into which data is read or from which it is written. Typically, buffers are used only for temporary storage.

by content
See *pass by content*.

by reference
See *pass by reference*.

by value
See *pass by value*.

byte The basic unit of storage addressability. It has a length of 8 bits.

C language
A high-level language used to develop software applications in compact, efficient code that can be run on different types of computers with minimal change.

C++ language
An object-oriented high-level language that evolved from the C language. C++ exploits the benefits of object-oriented technology such as code modularity, portability, and reuse.

C-CAA
C/370-specific common anchor area in the runtime environment.

CAA Common anchor area.

call chain
A trace of all active routines and subroutines that can be constructed by the user from information included in a system dump, such as the locations of save areas and the names of routines.

callable service stub
A short routine that is link-edited with an application and that is used to transfer control from the application to a callable service.

callable services
A set of services that can be invoked by a Language Environment-conforming high-level language using the conventional Language Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

called routine
A routine or program that is invoked by another.

callee A routine or program that is invoked by another.

caller A routine or program that invokes another routine.

calling routine
A routine or program that invokes another routine.

CASE Computer-aided software engineering.

cast In C, an expression that converts the type of the operand to a specified data type (the operator).

cataloged procedure
A set of job control language (JCL) statements placed in a library and retrievable by name.

CBIPO
Custom-Built Installation Process Offering.

CBPDO
Custom-Built Product Delivery Offering.

CCSID
See *coded character set ID (CCSID)*.

CEEDUMP
A dump of the runtime environment for Language Environment and the member language libraries. Sections of the dump are selectively included, depending on options specified on the dump invocation. This is not a dump of the full address space, but a dump of storage and control blocks that Language Environment and its members control.

century window
The 100-year interval in which Language Environment assumes all 2-digit years lie. The Language Environment default century window begins 80 years before the system date.

chained list
Synonym for *linked list*.

character
A letter, digit, or other symbol. A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes.

child enclave
The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CIB Condition information block.

CICS Customer Information Control System.

CICS destination control table (DCT)
A table that contains an entry for each extrapartition, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set. Indirect destination entries contain the information required to locate the queue in the intrapartition data set.

CICS OTE
CICS Open Transaction Environment.

CICS run unit
Consists of a statically and/or dynamically bound set of one or more load modules which can be loaded by a CICS loader. A CICS run unit is equivalent to a Language Environment *enclave*.

CICS translator
A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

CLIST TSO command list.

CLLE COBOL load list entry.

CMS Conversational monitor system.

CMS extended parameter list
A type of parameter list available in the CMS environment consisting of a string composed exactly as the user typed it at the terminal. There is no tokenization performed on the string.

CMS tokenized parameter list
A type of parameter list available in the CMS environment consisting of 8-byte tokens, folded to uppercase, terminating with a double word of X'FF'. Not supported under Language Environment.

COBCOM
Control block containing information about a COBOL partition.

COBOL
COmmon Business-Oriented Language. A high-level language, based on English, that is primarily used for business applications.

COBOL load list entry (CLLE)
Entry in the load list containing the name of the program and the load address.

COBOL run unit
A COBOL-specific term that defines the scope of language semantics. Equivalent to a Language Environment *enclave.*

COBPACK
A collection of individual modules that are packaged into a single load module in order to reduce the time that would otherwise be needed to load the individual load modules.

COBVEC
A COBOL vector table containing the address of the COBOL library routines.

coded character set ID (CCSID)
For Enhanced ASCII functionality, a 16-bit value is a number that represents a character set used by file tagging. It identifies the currrent character set of text strings within a program. This is stored in the file tag of new files or used for the automatic conversion of old files when automatic conversion is in effect. See also *automatic conversion* and *file tag*.

command processor parameter list (CPPL)
The format of a TSO parameter list. When a TSO terminal monitor application attaches a command processor, register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

COMMAREA
A communication area made available to applications running under CICS.

common anchor area (CAA)
Dynamically acquired storage that represents a Language Environment thread. Thread-related storage/resources are anchored off of the CAA. This area acts as a central communications area for the program, holding addresses of various storage and error-handling routines, and control blocks. The CAA is anchored by an address in register 12.

common block
A storage area that may be referenced by one or more compilation units. It is declared in a Fortran program with the COMMON statement. See also *external data*.

compilation unit
An independently compilable sequence of HLL statements. Each HLL product has different rules for what makes up a compilation unit. Synonymous with *program unit*.

compile-time options
Keywords that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, the destination of error messages, and other things.

compiler options
Keywords that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, and the destination of error messages. See also *compiler-time options*.

component
A set of modules that performs a major function within a system.

computer-aided software engineering (CASE)
A software engineering discipline for automating the application development process and thereby improving the quality of application and the productivity of application developers.

condition
An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition code
A code that reflects the result of a previous input/output, arithmetic, or logical operation.

condition handler
A user-written routine or language-specific routine (such as a PL/I ON-unit or C `signal()` function call)

invoked by the Language Environment *condition manager* to respond to conditions.

condition handling
In Language Environment, the diagnosis, reporting, and/or tolerating of errors that occur while a routine is running.

condition information block (CIB)
The platform-specific data block used by the Language Environment condition manager as a repository for data about conditions raised in the Language Environment runtime environment.

condition manager
Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition step
The step of the Language Environment condition handling model that follows the enablement step. In the condition step, user-written condition handlers, C signal handlers, and PL/I ON-units are first given a chance to handle a condition. See also *enablement step* and *termination imminent step*.

condition token
In Language Environment, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

condition variable
A data object that is used for waiting for long durations of time. An application can wait for the variable to become true before continuing processing. [POSIX.1]

conflicting name
One of 20 names that exist in both the Fortran and the C/C++ libraries. See also *conflicting reference*.

conflicting reference
An external reference from a Fortran or assembler language routine to a Fortran library routine with a name that is the same as the name of a C/C++ library routine. The reference is considered to be a conflicting reference only when the intended resolution is to the Fortran library routine rather than to the corresponding C/C++ library routine.

constructed reentrancy
The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

control block
A storage area used by a computer program to hold control information.

control section (CSECT)
The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

control statement
In programming languages, a statement that is used to alter the continuous sequential execution of statements; a control statement can be a conditional statement, such as IF, or an imperative statement, such as STOP. In JCL, a statement in a job that is used in identifying the job or describing its requirements to the operating system.

conversational monitor system (CMS)
A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities, and operates only under the control of the VM/370 control program.

CPPL Command processor parameter list.

CSECT
Control section.

cumulative service tape
A tape sent with a new function order, containing all current PTFs for that function.

cursor One of two pointers managed by the condition manager as it processes a condition. See *handle cursor* and *resume cursor*.

Custom-Built Installation Process Offering (CBIPO)
A CBIPO is a tape that has been specially prepared with the products (at the appropriate release levels) requested by the customer. A CBIPO simplifies installing various products together.

Custom-Built Product Delivery Offering (CBPDO)
A CBPDO is a tape that has been specially prepared for installing a particular product and the related service requested by the customer. A CBPDO simplifies installing a product and the service for it.

Customer Information Control System (CICS)
CICS is an OnLine Transaction Processing (OLTP) system that provides specialized interfaces to databases, files and terminals in support of business and commercial applications.

CWI Compiler-writer interface.

dangling pointer
A pointer to storage that has been freed.

data, qualifying
See *qualifying data*.

data aggregate
A logical collection of data elements that can be referred to either collectively or individually. In PL/I, an array or a structure.

data division
In COBOL, the part of a program that describes the files to be used in the program and the records contained within the files. It also describes any WORKING-STORAGE data items, LINKAGE SECTION data items, and LOCAL-STORAGE data items that are needed.

data set
Under MVS, a named collection of related data records that is stored and retrieved by an assigned name. Equivalent to a CMS *file*.

data type
The properties and internal representation that characterize data.

datum, qualifying
A single element of qualifying data associated with a condition. See *qualifying data*.

DBCS Double-byte character set.

DB2 DATABASE 2; generally, one of a family of IBM relational database management systems and, specifically, the system that runs under MVS.

DCLCB
Declare control block.

DCT Destination control table.

DD statement
In MVS, the data definition statement. A JCL control statement that serves as the connection between a file's logical name (the ddname) and the file's physical name (the data srt name).

ddname
Data definition name. The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file through a FILEDEF command, DD statement, or ALLOCATE command. DD statement or ALLOCATE command.

decimal overflow
A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declare control block (DCLCB)
Control block containing file information.

default
A value that is used or an action that is taken when no alternative is specified.

dereference
In C, the application of the unary operator (*) to a pointer to access the object the pointer points to. Also known as *indirection*.

descriptor
PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

descriptor, q_data
See *q_data descriptor*.

destination control table (DCT)
In CICS, a table containing an entry for each extrapartition, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Indirect destination entries redirect data to a destination controlled by another DCT entry. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set.

device A computer peripheral or an object that appears to the application as such. [POSIX.1]

direct argument passing
A type of parameter passing in which the value of the argument is placed directly in the argument list body.

directory entry
An object that associates a filename with a file. Several directory entries can associate names with the same file. [POSIX.1]

disabled/enabled
See *enabled/disabled*.

distribution libraries
IBM-supplied partitioned data sets on tape containing one or more components that the user restores to disk for subsequent inclusion in a new system.

distribution zone
In SMP/E, a group of VSAM records that describe the SYSMODs and elements in the distribution libraries.

double-byte character set (DBCS)
A collection of characters represented by a 2-byte code.

downward-growing stack
With Extra Performance Linkage (XPLINK), a stack that grows from high addresses to low addresses in memory.

downwardly compatible
The ability of applications that have been compiled and linked with Language Environment to run on previous releases of OS/390. In order for an application to be downwardly compatible, it must not have exploited any new Language Environment function unavailable in the targeted release.

double-precision
Pertaining to the use of two computer words to represent a number in accordance with the required precision. See also *precision, single-precision*.

doubleword
A sequence of bits or characters that comprises eight bytes (two 4-byte words) and is referenced as a unit.

doubleword boundary
A storage location whose address is evenly divisible by 8.

driving system
The system used to install the program. Contrast with target system.

DSA Dynamic storage area.

dummy argument
The Fortran term for the data received by a called routine. See also *actual argument*.

dynamic call
A call that results in locating a called routine at run time, that is, by loading the routine into virtual storage. Contrast with *static call*.

dynamic loading
See *dynamic call*.

dynamic storage
Storage acquired as needed at run time. Contrast with *static storage*.

dynamic storage area (DSA)
An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment–managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is also known as a *stack frame.*

EBCDIC
Extended binary-coded decimal interchange code.

EIB EXEC interface block.

enabled/disabled
A condition is enabled when its occurrence will result in the execution of condition handlers or in the performance of a standard system action to handle the condition as defined by Language Environment.

A condition is disabled when its occurrence is ignored by the condition manager.

enablement
The determination by a language at run time that an exception should be processed as a condition. This is the capability to intercept an exception and to determine whether it should be ignored or not; unrecognized exceptions are always defined to be enabled. Normally, enablement is used to supplement the hardware for capabilities that it does not have and language enforcement of a language's semantics. An example of supplementing the hardware is the specialized handling of exponent-overflow exceptions based on language standards.

enablement step
The first step of the Language Environment condition handling model. In the enablement step it is determined whether an exception is to be *enabled* and processed as a condition. See also *condition step* and *termination imminent step*.

enclave
In Language Environment, an independent collection of routines, one of which is designated as the main routine and is invoked first. An enclave is roughly analogous to a program or run unit. an executable program.

enterprise
The composite of all operational entities, functions, and resources that form the total business concern.

entry name
In assembler language, a programmer-specified name within a control section that identifies an entry point and can be referred to by any control section. See also *entry point*.

entry point
The address or label of the first instruction that is executed when a routine is entered for execution. Within a load module, the location to which control is passed when the load module is invoked.

entry point name
The symbol (or name) that represents an entry point. See also *entry point*.

environment
A set of services and data available to a program during execution. In Language Environment, environment is normally a reference to the runtime environment of HLLs at the enclave level.

environment variable
A variable that is included in the current software environment and is therefore available to any called program that requests it.

epilog Code generated at the end of a routine, normally causing a return to the caller of the routine.

euro The monetary unit of the European Monetary Union (EMU) that was introduced alongside national currencies on 01 January 1999.

EuroReady product
A product is EuroReady if the product, when used in accordance with its associated documentation, is capable of correctly processing monetary data in the euro denomination, respecting the euro currency formatting conventions (including the euro sign). This assumes that all other products (for example, hardware, software, and firmware) that are used with this product are also EuroReady. IBM hardware products that are EuroReady may or may not have an engraved euro sign key on their keyboards.

EXEC interface block (EIB)
In CICS, a control block containing information useful in the execution of an application, such as a transaction identifier and a time and a date when the transaction is started.

exception
The original event such as a hardware signal, software detected event, or user-signaled event which is a potential condition. This action may or may not include an alteration in a program's normal flow. See also *condition*.

execution time
Synonym for *run time*.

execution environment
Synonym for *runtime environment*.

extended binary-coded decimal interchange code (EBCDIC)
A set of 256 8-bit characters.

exponent-overflow exception
The program interruption that occurs when an overflow occurs during the execution of a floating-point instruction, that is, when the result value from the instruction has a characteristic that is larger than the floating-point data format can handle.

exponent-underflow exception
The program interruption that occurs when the result value from executing a floating-point instruction has a nonzero fraction and a characteristic is smaller than the floating-point data format can handle. This program interruption can be disabled through a program mask bit setting.

extended error handling facility
The VS FORTRAN facility that provided automatic error correction and control over both the handling of the errors and the printing of error messages.

external data
Data that persists over the lifetime of an enclave and maintains last-used values whenever a routine within the enclave is reentered. Within an enclave consisting of a single load module, it is equivalent to any C data objects that have static storage duration, a Fortran common block, and COBOL EXTERNAL data.

external reference
In an object module, a reference to a symbol, such as an entry point name, defined in another program or module.

Extra Performance Linkage (XPLINK)
Extra Performance Linkage (XPLINK) is an enhanced linkage between programs that can significantly improve the performance of your C and C++ programs. The primary goal of XPLINK is to make subroutine calls as fast and efficient as possible by removing all nonessential instructions from the main program path. The XPLINK runtime option controls the initialization of the XPLINK environment.

FCB File control block.

feature
A part of an IBM product that may be ordered separately by a customer.

feature code
A four-digit code used by IBM to process hardware and software orders.

feedback code (fc)
A condition token value. If you specify *fc* in a call to a callable service, a condition token indicating whether the service completed successfully is returned to the calling routine.

fetch The dynamic load of a PL/I procedure.

FIB File information block.

file A named collection of related data records that is stored and retrieved by an assigned name. Equivalent to an MVS *data set*.

file control block (FCB)
Block containing the addresses of I/O routines, information about how they were opened and closed, and a pointer to the file information block.

FILEDEF
File definition statement.

file definition statement (FILEDEF)
In CMS, serves as the connection between the logical name of a file and the physical name of a file.

file descriptor
A per-process unique, nonnegative integer used to identify an open file for the purpose of file access. [POSIX.1]

file information block (FIB)
A read-only block describing the characteristics of an I/O file.

file system
A collection of files and certain of their attributes. A file system provides a name space for file serial numbers referring to those files.

file tag
For Enhanced ASCII functionality, a file attribute that identifies the character set of the text data within a file and indicates whether the file is eligible for automatic conversion. See also *automatic conversion* and *coded character set ID (CCSID).*

fix A correction of an error in a program, usually a temporary correction or bypass of defective code.

fix-up and resume
The correction of a condition either by changing the argument or parameter and running the routine again or by providing a specific value for the result.

fixed decimal
See *packed decimal format*.

fixed-point overflow exception
A program interruption caused by an overflow during signed binary arithmetic or signed left-shift operations. This program interruption can be disabled through a program mask bit setting.

floating point control register (FPC register)
For IEEE, a floating point control register.

FMID Function modification identifier.

Fortran
A high-level language used primarily for applications involving numeric computations. In previous usage, the name of the language was written in all capital letters, that is, FORTRAN.

Fortran signature CSECT
The resident routine that indicates that the load module in which it is present contains a Fortran routine.

FORTRAN 66
The FORTRAN language standard formally known as *American National Standard FORTRAN, ANSI X3.9-1966*. This language standard specifies the form and establishes the interpretation of programs written to conform to it.

FORTRAN 77
The FORTRAN language standard formally known as *American National Standard FORTRAN, ANSI X3.9-1978*. This language standard specifies the form and establishes the interpretation of programs written to conform to it.

FPC See *floating point control register (FPC register).*

fullword
A sequence of bits or characters that comprises four bytes (one word) and is referenced as a unit.

fullword boundary
A storage location whose address is evenly divisible by 4.

function
A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

function modification identifier (FMID)
The value used to distinguish separate parts of a product. A product tape or cartridge has at least one FMID.

GET Global error table.

global error table (GET)
A method employed by some HLLs, for example, C and Fortran, to determine actions for handling conditions. Whereas Language Environment condition handling actions are defined at the stack frame level, actions defined using the global error table apply to an entire application until explicitly changed. See also *extended error handling facility*.

Gregorian calendar
The calendar in use since Friday, 15 October 1582 throughout most of the world. Used as the basis for the *Lilian date* used in many Language Environment date and time services.

GTAB table
Table in C/370 containing error information.

handle cursor
A pointer used by the condition manager as it traverses the stack. The handle cursor points to the condition handler currently being invoked in the stack frame, whether it be a user-written condition handler or an HLL-specific condition handler.

handled condition
A condition that either a user-written condition handler or the HLL-specific condition handler has processed and for which the condition handler has specified that execution should continue.

handler
See *condition handler*.

header file
A file that contains system-defined control information that precedes user data.

heap 0
Synonymous with *initial heap*.

heap An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments. See *anywhere heap, below heap, initial heap,* and *additional heap.*

heap element
A contiguous area of storage allocated by a call to the CEEGTST service. Heap elements are always allocated within a single heap segment.

heap increment
See *increment*.

heap pool
A storage pool that, when used by the storage manager, can be used to improve the performance of heap storage allocation. This can improve the performance of a multi-threaded application.

heap segment
A contiguous area of storage obtained directly from the operating system. The Language Environment storage management scheme subdivides heap segments into individual heap elements. If the initial heap segment becomes full, Language Environment obtains a second segment, or increment, from the operating system.

heap storage
See *heap*.

heavy weight thread
A heavy weight thread has a one-to-one correspondence with an MVS task control block (TCB) in that the lifetime of the thread is the lifetime of the TCB. [POSIX.1]

hexadecimal
A base 16 numbering system. Hexadecimal digits range from 0 through 9 (decimal 0 to 9) and uppercase or lowercase A through F (decimal 10 to 15) and A through F, giving values of 0 through 15.

high-level language (HLL)
A programming language above the level of assembler language and below that of program generators and query languages. Examples are C, C++, COBOL, Fortran, and PL/I.

HLL High-level language.

hook The location in a compiled program where the compiler inserts an instruction that allows the user to later interrupt the program (by setting breakpoints) for debugging purposes.

IBM service representative
An individual in IBM who performs maintenance services for IBM products or systems.

IBM Software Distribution (ISD)
The IBM department responsible for software distribution.

IBM Support Center
The IBM department responsible for software service.

IBM systems engineer (SE)
An IBM service representative who performs maintenance services for IBM software in the field.

implementation defined
An indication that the implementation defines and documents the requirements for correct program constructs and correct data of a value or behavior. [POSIX.1]

ILC Interlanguage communication.

IMS Information Management System, IBM licensed product. IMS supports hierarchical databases, data communication, translation processing, and database backout and recovery.

increment
The second and subsequent segments of storage allocated to the stack or heap.

indirect argument passing
The body of the argument list contains a pointer to the argument value.

indirection
See *dereference*.

initial heap
The Language Environment heap controlled by the HEAP runtime option

and designated by a *heap_id* of 0. The initial heap contains dynamically allocated user data. See also *additional heap*.

initial heap segment
The first heap segment. A heap consists of the initial heap segment and zero or more additional segments or increments.

Initial process thread (IPT)
See *initial thread*.

initial program load (IPL)
The process of loading system programs and preparing a system to run jobs.

initial stack segment
The first stack segment. A stack consists of the initial stack segment and zero or more additional segments or increments.

initial thread
In terms of POSIX, either the thread established by the `fork()` that created the *process*, or the first thread that calls `main()` after an `exec`. Also known as *initial process thread (IPT)*. [POSIX.1]

input procedure
A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance-specific information (ISI)
Located within the Language Environment condition token, information used by a condition handler or the condition manager to interpret and react to a specific occurrence of a condition. Qualifying data is an example of instance-specific information.

integer
A positive or negative whole number or zero.

interactive
Pertaining to a program or system that alternately accepts input and responds. In an interactive system, a constant dialog exists between user and system. Contrast with *batch*.

interactive problem control system (IPCS)
A component of z/OS that permits online problem management, interactive problem diagnosis, online debugging for disk-resident CP abend dumps, problem tracking, and problem reporting.

Interactive System Productivity Facility (ISPF)
A dialog manager for interactive applications. It provides control and services to permit execution of dialogs.

interface validation exit
A routine that, when used with the binder, automatically resolves conflicting references within Fortran routines.

interlanguage communication (ILC)
The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

interrupt
A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.

interruption
Synonym for *interrupt*.

IPCS Interactive problem control system

IPL Initial program load.

ISI Instance specific information.

ISPF Interactive System Productivity Facility.

JCL Job control language.

job control language (JCL)
A sequence of commands used to identify a job to an operating system and to describe a job's requirements.

job step
The job control (JCL) statements that request and control execution of a program and that specify the resources needed to run the program. The JCL statements for a job step include one EXEC statement, which specifies the program or procedure to be invoked, followed by one or more DD statements, which specify the data sets or I/O devices that might be needed by the program.

Julian date
A date format that contains the year in positions 1 and 2, and the day in

positions 3 through 5. The day is represented as 1 through 366, right-adjusted, with zeros in the unused high-order position.

kernel The part of the component that contains programs for such tasks as I/O, management, and communication.

KSDS Key-sequenced data set. See also *VSAM*.

L-name
In C, this is a mixed-case external identifier that is up to 255 characters long. See also *S-name*.

Language Environment
Short form of z/OS Language Environment. A set of architectural constructs and interfaces that provides a common runtime environment and runtime services for C, C++, COBOL, Fortran, PL/I, and Java applications compiled by Language Environment-conforming compilers.

Language Environment-conforming
Adhering to Language Environment's common interface conventions.

Language Environment-enabled
A program that has been link-edited with the routines or stubs provided with Language Environment.

language-sensitive editing
A set of editing functions that are responsive to the programming language, syntax, and environment of source programs as they are being edited. Typical language-sensitive editing features are automatic indenting, token highlighting, syntax checking, and language-sensitive help.

LIBPACK
A collection of individual modules that are packaged into a single load module in order to reduce the time that would otherwise be needed to load the individual load modules.

library
A collection of functions, subroutines, or other data.

library latch
An object similar to a mutex and used within the Language Environment library to synchronize access to resources shared among threads.

library vector table (LIBVEC)
A vector table used to support access to library routines (Language Environment and HLLs) from compiler-generated code, user-written assembly language code, and other subroutines.

library workspace (LWS)
Special register save areas for certain PL/I library routines, preallocated in nonstack storage.

LIBVEC
Library vector table.

LIFO Last in, first out method of access. A queuing technique in which the next item to be retrieved is the item most recently placed in the queue.

Lilian date
The number of days since the beginning of the Gregorian calendar. Day one is Friday, 15 October 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

link pack area (LPA)
In MVS, an area of main storage containing reenterable routines from system libraries. Their presence in main storage saves loading time when a reenterable routine is needed.

link-edit
To create a loadable computer program by means of a linkage editor or binder.

linkage editor
An operating system component that resolves cross-references between separately compiled or assembled modules and then assigns final addresses to create a single relocatable load module. The linkage editor then stores the load module in a load library on disk.

linked list
A list in which the data elements may be dispersed but in which each data element contains information for locating the next. Synonymous with *chained list*.

load module
A collection of one or more routines that have been stored in a library by the linkage or binder after having been

compiled or assembled. External references have usually been—but are not necessarily—resolved. When the external references have been resolved, the load module is in a form suitable for execution.

local data
Data that is known only to the routine in which it is declared. Equivalent to local data in C and both WORKING-STORAGE and LOCAL-STORAGE in COBOL.

locale An identifer that determines the way in which data is processed, printed, and displayed in a particular user community. A locale includes conventions for a specific language and culture, with appropriate numeric representation, date and time formatting, and monetary formatting.

locator
PL/I control block that holds the address of data such as structures or arrays and the address of the *descriptor*.

LPA Link pack area.

LWS Library workspace.

machine readable
Pertaining to data a machine can acquire or interpret (read) from a storage device, a data medium, or other source.

main program
The first routine in an enclave to gain control from the invoker. In Fortran, a main program does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It could have a PROGRAM statement as its first statement. Contrast with *subprogram*.

main task
In the context of MVS multitasking and the C Multitasking Facility (MTF), the main program in a multitasking environment. The main task runs the *main task program*.

main task program
In the context of MVS multitasking and the C Multitasking Facility (MTF), the part of a program that controls overall processing. The main task program is run by the *main task*.

mapped condition
A condition that is generated by one component and converted, or mapped, to another component; for example, some Language Environment conditions, such as attention interrupts or the decimal divide condition, map directly to the PL/I ATTENTION and ZERODIVIDE conditions, respectively.

megabyte (MB)
1,048,576 bytes.

medium weight thread
A medium weight thread has a one-to-one correspondence with an MVS TCB except the lifetime of the TCB may exceed the lifetime of the thread. [POSIX.1]

memory file control block (MFCB)
Block residing at thread level in C/370 containing the memory information about the file.

MFCB Memory file control block.

microfiche
A sheet of microfilm capable of containing microimages in a grid pattern, usually containing a title that can be read without magnification.

module
A language construct that consists of procedures or data declarations and can interact with other such constructs. In PL/I, an external procedure.

MTF Multitasking Facility.

multilevel security
Allows the classification of data and users based on a system of hierarchical security levels, combined with a system of non-hierarchical security categories. The security administrator classifies users and data, and the system then imposes mandatory access controls restricting which users can access data, based on a comparison of the classification of the users and the data.

Multitasking Facility (MTF)
Facility provided separately by C and by Fortran to improve turnaround time on multiprocessor configurations by using MVS multitasking facilities. MTF is provided by C library functions or by Fortran callable services.

multitasking
A mode of operation in which two or more tasks can be performed at the same time.

multithreading
A mode of operation in which the operating system can run different parts of a program, called threads, simultaneously.

mutex A mutual exclusive variable that is intended to serialize access to a shared data object for a short duration of time. [POSIX.1]

MVS Multiple Virtual Storage operating system.

n-way ILC application
An ILC application that includes three or more of the following: a C routine, a COBOL program, a Fortran program, and a PL/I routine.

NAB Next available byte.

name scope
The portion of an application within which a particular declaration of external data applies or is known.

name space
The portion of a load module within which a particular declaration of external data applies or is known.

named heap
A heap set up specifically by the CEECRHP callable service. An identifier is returned when the heap is created.

national language support
Translation requirements affecting parts of licensed programs; for example, translation of message text and conversion of symbols specific to countries.

natural reentrancy
The attribute of applications that contain no static external data and do not require additional processing to make them reentrant. Contrast with *constructed reentrancy*.

nested condition
A condition that occurs during the handling of another, previous condition. Language Environment by default permits 10 levels of nested conditions. This setting may be changed by altering the DEPTHCONDLMT runtime option.

nested enclave
A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

nested program
In COBOL, a program that is directly contained within another program.

next available byte (NAB)
The address of the next available byte of storage on a doubleword boundary. This address is a segment of stack storage.

next sequential instruction
The next instruction to be executed in the absence of any branch or transfer of control.

nonreentrant
A type of program that cannot be shared by multiple users.

null Empty, having no meaning.

null character
A character that represents X'00'.

null string
A string containing no element. A character or bit string with a length of zero.

object module
A collection of one or more control sections produced by an assembler or compiler and used as input to the linkage editor or binder. Synonym for *text deck* or *object deck*.

offset The number of measuring units from an arbitrary starting point in a record, area, or control block, to some other point.

omitted parameter
A parameter not needed in a call.

online Pertaining to a user's ability to interact with a computer. Pertaining to a user's access to a computer via a terminal.

OpenExtensions
VM/ESA services that support an environment within which operating systems, servers, distributed systems, and workstations share common interfaces.

OpenExtensions supports standard application development across multi-vendor systems. It is required if you want to create and use VM/ESA applications that conform to the POSIX standard.

operating system
Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

OS PL/I
See *PL/I*.

out-of-storage condition
A condition signaled when an application has used all of the storage allocated to it. If the STORAGE runtime option is set to a value other than 0, Language Environment adds a reserve stack segment to the overflowing stack, and then signals the out-of-storage condition.

output procedure
A set of statements, to which control is given during the execution of a SORT statement after the sort function is completed, or during the MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow
Exceeding the capacity of the intended unit of storage. See also *fixed-point overflow exception* and *exponent-overflow exception*.

overlay
To write over existing data in storage.

owning stack frame
Given the calling sequence of Routine 1 calling Routine 2 that in turn calls Routine 3, Routine 3 is the owning stack frame if a condition occurs while Routine 3 is executing.

ON-unit
The specified action to be taken upon detection of the condition named in the containing ON statement.

packed decimal format
A format in which each byte in a field except the rightmost digit represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111.

pad
To fill unused positions in a field with dummy data, usually zeros, ones, or blanks.

parallel function
In the context of MVS multitasking and the C Multitasking Facility, those portions of a program that can run independently of the *main task program* and each other. *Subtasks* run the parallel functions.

parallel program
In the context of the Fortran parallel facility (not MTF), a program that uses parallel language constructs, invokes any of the parallel callable services, or was compiled with the PARALLEL compile-time option.

parallel subroutine
In the context of MVS multitasking and the Fortran Multitasking Facitity, those portions of a program that can run independently of the main task program and each other. The parallel subroutines run in MVS subtasks.

parameter
1) Data items that are received by a routine. 2) The term used in certain other languages for the Fortran term *dummy argument*. See *argument*, *actual argument*, and *dummy argument*.

parent enclave
The enclave that issues a call to system services or language constructs to create a nested (child) enclave. See also *child enclave* and *nested enclave*.

partition
A fixed-size division of storage.

pass by content
A COBOL argument passing style synonymous with passing an argument by value (indirect). In this style, R1 contains a pointer to a copy of the argument.

pass by reference
In programming languages, one of the basic argument passing semantics where the address of the object is passed. Any

changes made by the callee to the argument value will be reflected in the calling routine at the time the change is made.

pass by value
In programming languages, one of the basic argument passing semantics where the value of the object is passed. Any changes made by the callee to the argument value will not be reflected in the calling routine.

percolate
The action taken by the condition manager when the returned value from a condition handler indicates that the handler could not handle the condition, and the condition will be transferred to the next handler.

picture string
Character strings used to specify date and time formats.

PID Process ID.

PL/I A general purpose scientific/business high-level language. PL/I is a high-powered procedure-oriented language especially well suited for solving complex scientific problems or running lengthy and complicated business transactions and record-keeping applications.

pointer
A data element that indicates the location of another data element.

portability
The ability to transfer an application from one platform to another with relatively few changes to the source code.

Portable Operating System Interface (POSIX)
Portable Operating System Interface for computing environments, an interface standard governed by the IEEE and based on UNIX. POSIX is not a product. Rather, it is an evolving family of standards describing a wide spectrum of operating system components ranging from C language and shell interfaces to system administration.

POSIX
Portable Operating System Interface.

POSIX process
An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the `fork()` function. The process that issues `fork()` is known as the parent process, and the new process created by the `fork()` is known as the child process. [POSIX.1]

POSIX signal
A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. [POSIX.1]

PPA1 entry point block
Program Prolog Area. This block contains information about the compiled module.

PPA2 entry point block
An extension of the *PPA1 entry point block*.

PPT Processing program table.

precedence
In programming languages, an order relation defining the sequence of the application of operations or options.

precision
A measure of the ability to distinguish between nearly equal values, usually with data of different lengths. See also *single-precision* and *double-precision*.

preinitialization
A facility that allows a routine to initialize the runtime environment once, perform multiple executions within the environment, then explicitly terminate the environment.

Preinitialized Environments for Authorized Programs
A facility that allows an authorized AMODE 64 application to run z/OS XL C/C++ and Language-Environment conforming Assembler routines through the use of preinitialized environments.

pre-Language Environment-conforming
Any HLL program that does not adhere to Language Environment's common interface. For example, VS COBOL II, OS/VS COBOL, OS PL/I, C/370 Version 1 and Version 2, VS FORTRAN Version 1, VS FORTRAN Version 2, FORTRAN IV

G1, and FORTRAN IV H Extended are all pre-Language Environment-conforming HLLs.

prelinker
A utility that collects compile-time initialization information from one or more object modules into a single initialization unit. In the process, the static external data part is mapped.

preprocessor
A routine that examines application source code for preprocessor statements that are then executed, resulting in the alteration of the source.

preventive service planning (PSP)
The online repository of program temporary fixes (PTFs) and other service information. This information could affect installation.

procedure
In COBOL, a procedure is a paragraph or section that can only be performed from within the program. In PL/I, a named block of code that can be invoked externally, usually via a call.

procedure library (PROCLIB)
A program library in direct access storage with job definitions. The reader/interpreter can be directed to read and interpret a particular job definition by an execute statement in the input stream.

process
The highest level of the Language Environment program management model. A process is a collection of resources, both program code and data, and consists of at least one enclave. See also *POSIX process*.

process ID (PID)
The unique identifier representing a process. A process ID is a positive integer that can be contained in the data type *pid_t*. A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process groups whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1. [POSIX.1]

processing program table (PPT)
Contains information about CICS load modules (whether the module is in storage or not, its language, use count and entry point address, etc.) needed to complete a transaction.

program
See *enclave*.

program control data
In PL/I, data used to affect how a program runs; that is, any data that is not string or arithmetic data.

program interruption
The interruption of the execution of a program due to some event such as an operation exception, an exponent-overflow exception, or an addressing exception.

program level
The modification level, release, version, and fix level.

program management
The functions within the system that provide for establishing the necessary activation and invocation for a program to run in the applicable runtime environment when it is called.

program mask
In bits 20 through 23 of the program status word (PSW), a 4-bit structure that controls whether each of the fixed-point overflow, decimal overflow, exponent-overflow, and significance exceptions should cause a program interruption. The bits of the program mask can be manipulated to enable or disable the occurrence of a program interruption.

program number
The seven-digit code (in the format xxxx-xxx) used by IBM to identify each program product.

program specification block (PSB)
In IMS/VS, a control block that contains all database program communication blocks (DB PCB) that exist for a single application program. DB PCBs define which segments in a database an application can access.

program status word (PSW)
A 64-bit structure that includes the

instruction address, program mask, and other information used to control instruction sequencing and to determine the state of the CPU. See also *program mask*.

program temporary fix (PTF)
A temporary solution or bypass of a problem diagnosed by IBM as resulting from a defect in a current unaltered release of the program.

program unit
Synonym for *compilation unit*.

programmable workstation (PWS)
A workstation that has some degree of processing capability and that allows a user to change its functions.

prolog The code sequence when a routine is entered.

promote
To change a condition to a different one by a condition handler. A condition handler routine promotes a condition because the error needs to be handled in a way other than that suggested by the original condition.

PSB Program specification block.

PSP Preventive service planning.

PSW Program status word.

PWS Programmable workstation.

q_data
Qualifying data. Information that a user-written condition handler can use to identify and react to a given instance of a condition.

q_data descriptor
A qualifying datum that contains the data type and length of the immediately following qualifying datum associated with a condition token.

q_data_token
An optional 32-bit data object that is placed in the ISI. It is used to access the qualifying data associated with a given instance of a condition.

qualifier
A modifier that makes a name unique.

qualifying data
q_data. Unique information associated through a condition token with a given instance of a condition. A user-written condition handler uses qualifying data to identify and react to the condition.

qualifying datum
A single element of qualifying data associated with a condition. See *qualifying data*.

reason code
1) Return code to CICS only. 2) A value returned to the invoker of an enclave that indicates how the enclave terminated. The value reflects whether the enclave terminated successfully, or unsuccessfully, to an unhandled condition.

recursive routine
A routine that can call itself or be called by another routine that it has called.

reenterable
reentrant

reentrant
The attribute of a routine or application that allows more than one user to share a single copy of a load module.

register
Special processing areas that hold a specific amount of data and can process, load, and store this data quickly. To specify formally. In Language Environment, to register a condition handler means to add a user-written condition handler onto a routine's stack frame.

register save area (RSA)
Area of main storage in which contents of registers are saved.

regular file
A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1]

relative pathname
A pathname that does not begin with a slash. The predecessor of the first filename in the pathname is taken to be the current working directory of the process. [POSIX.1]

reserved word
In programming languages, a keyword that may not be used as an identifier.

resident modules
A module that remains in a particular area of storage.

resident routines
The Language Environment library routines linked with your application. They include such things as initialization routines and *callable service stubs*.

resume
To continue execution in an application at the point immediately after which a condition occurred. This occurs when a condition handler determines that a condition has been handled and normal application execution should continue.

resume cursor
The point in an application at which execution should continue if a condition handler requests the resume action for a condition it is processing. When a condition is signaled, the resume cursor is at the location at which the error occurred or at which the condition was first reported to the condition manager. The resume cursor can be moved with the CEEMRCE or CEEMRCR callable service.

return code
A code produced by a routine to indicate its success or failure. It may be used to influence the execution of succeeding instructions or programs.

return_code_modifier
A value set by Language Environment routines to indicate the severity of an unhandled condition. The return_code_modifier is a component of the return code that indicates the status of the execution of an enclave.

RMODE
Residence mode. Provided by the linkage editor, the attribute of a load module that specifies whether the module, when loaded, must reside below the 16MB virtual storage line or may reside anywhere in virtual storage.

rollback
The process of restoring data changed by an application to the state at its last commit point.

root load module
The load module containing a main routine and the first to be executed in an application.

routine
In Language Environment, refers to a PL/I procedure, a C function, a Fortran main program or subprogram, or a COBOL program or a separate subroutine.

RSA Register save area.

run To cause a program, utility, or other machine function to be performed.

RUNCOM
COBOL block containing the ID and address of the main program.

run time
Any instant at which a program is being executed. Synonymous with *execution time*.

runtime environment
A set of resources that are used to support the execution of a program. Synonymous with *execution environment*.

run unit
One or more object programs that are executed together. In Language Environment, a run unit is the equivalent of an *enclave*.

safe condition
Any condition having a severity of 0 or 1. Such conditions are ignored if no condition handler handles the condition.

save area
Area of main storage in which contents of registers are saved.

SBCS Single-byte character set.

scalar A quantity characterized by a single value. Contrast with *aggregate*.

scalar instruction
An instruction, such as a load, store, arithmetic, or logical instruction, that operates on a scalar. Contrast with *vector instruction*.

scope A term used to describe the effective range of the enablement of a condition and/or the establishment of a user-generated routine to handle a condition. Scope can be both statically

and dynamically defined. The portion of an application within which the definition of a variable remains unchanged.

scope terminator
Variable at the end of a statement.

segment
See *stack segment*.

severity code
A part of runtime messages that indicates the severity of the error condition (1, 2, 3, or 4).

shared segment
In VM, a feature of a saved system that allows one or more segments of reentrant code in real storage to be shared among many virtual machines.

shared storage
An area of storage that is the same for each virtual address space. Because it is the same space for all users, information stored there can be shared and does not have to be loaded in the user region.

shared virtual area (SVA)
In VSE, a high address area of virtual storage that contains a system directory list (SDL) of frequently used phases, resident programs that can be shared between partitions, and an area for system support.

signal In C, signals are conditions that may or may not be reported during program execution, depending upon how they are defined to the condition handler. A condition is registered in C using the `signal()` function; a condition is raised using the `raise()` function. See also *POSIX signal* and *synchronous signal*. To make the condition manager aware of a condition for processing.

signal catching function
In POSIX, analogous to *signal handler*. The signal catching function is specified through the `sigaction()` function. [POSIX.1]

signal handler
In C, a function to be called when a *signal* is reported.

signature CSECT
The resident routine that indicates that the load module in which it is present contains a routine written in a particular language.

significance exception
The program interruption that occurs when the resulting fraction in a floating-point addition or subtraction instruction is zero. This program interruption can be disabled through a program mask bit setting.

single-byte character set (SBCS)
A collection of characters represented by a 1-byte code.

single-precision
Pertaining to the use of one computer word to represent a number in accordance with the required precision. See also *precision* and *double-precision*.

S-name
In C, this is a single-case external identifier that is at most eight characters long. See also *L-name*.

softcopy
One or more files that can be electronically distributed, manipulated, and printed by a user. Contrasts with hardcopy.

sort/merge program
A processing program that can be used to sort or merge records in a prescribed sequence.

source code
The input to a compiler or assembler, written in a source language.

source program
A set of instructions written in a programming language that must be translated to machine language before the program can be run.

stack An area of storage used for suballocation of stack frames. Such suballocations are allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack frame
The physical representation of the activation of a routine. The stack frame is allocated on a LIFO stack and contains various pieces of information including a

save area, condition handling routines, fields to assist the acquisition of a stack frame from the stack, and the local, automatic variables for the routine. In Language Environment, a stack frame is synonymous with *DSA*.

stack frame collapse
An action that occurs when the condition manager skips over one or more active routines and execution resumes in an earlier routine on the stack. A stack frame collapse happens is an explicit GOTO is coded in a C or PL/I routine or if the resume cursor is moved with the CEEMRCR.

stack increment
See *increment*.

stack segment
A contiguous area of storage obtained directly from the operating system. The Language Environment storage management scheme subdivides stack segments into individual DSAs. If the initial stack segment becomes full, a second segment or increment is obtained from the operating system.

stack storage
See *stack* and *automatic storage*.

standard system action
The name given to the language-defined default action taken when a condition occurs and it is not handled by a condition handler.

static call
A call that results in the resolution of the called program during the link-edit of the application. Contrast with *dynamic call*.

static data
Data that retains its last-used state across calls.

static storage
Storage that persists and retains its value across calls. Contrast with *dynamic storage*.

storage heap
An unordered group of program stack areas that may be associated with programs running within a process.

SUBCOM
Control block containing information about multiple COBOL programs.

suboption
A value that can be provided as part of a compile-time or runtime option to further specify the meaning of the option.

subpool storage
All of the storage blocks allocated under a subpool number for a particular task.

subprogram
A program unit that is invoked or used by another program unit. In Fortran, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. Contrast with *main program*.

SUBSET
The value that specifies the FMID for a product level.

subsystem
A secondary or subordinate system, or programming support, usually capable of operating independently of or asynchronously with a controlling system. Examples are CICS and IMS.

subtask
In the context of MVS multitasking and the C Multitasking Facility (MTF), a task that is initiated and terminated by a higher order task (the *main task*). Subtasks run the *parallel functions*, those portions of the program that can run independently of the *main task program* and each other.

SVC Supervisor call. A request that serves as the interface to certain functions, such as the allocation of storage.

symbolic feedback code
The symbolic representation of the first 8 bytes of the 12-byte condition token. In a condition-handling routine, a symbolic feedback code is substituted for the hexadecimal coding of the condition-handling routine.

synchronous signal
A signal attributable to a specific thread. Signals that can be generated synchronously are SIGABRT, SIGILL, SIGFPE, SIGPIPE, and SIGSEGV.

syntax The rules governing the structure of a programming language and the construction of a statement in a programming language.

system abend
An abend caused by the operating system's inability to process a routine; may be caused by errors in the logic of the source routine.

systems programming facility
runtime facilities provided by C that allow programs to be developed that do not require the Language Environment common library.

target libraries
In SMP/E, a collection of data sets in which the various parts of an operating system are stored. These data sets are sometimes called system libraries.

target zone
In SMP/E, a collection of VSAM records describing the target system macros, modules, assemblies, load modules, source modules, and libraries copied from DLIBs during system generation, and the SYSMODs applied to the target system.

task In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer.

task control block (TCB)
An MVS related control block which contains information and pointers associated with the task in process.

task global table (TGT)
Table with information about addresses and length of working storage and the program start address.

TCB Task control block.

termination imminent step
The final step of the 3-step Language Environment condition handling model. In the termination imminent step, user-written condition handlers and PL/I ON-units are given one last chance to handle a condition or perform cleanup before the thread is terminated. See also *condition step* and *enablement step*.

THDCOM
Control block with COBOL thread information.

thread The basic runtime path within the Language Environment program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

thread safe
A locking mechanism (mutex) that allows a thread to work with critical data or structures while preventing other threads from gaining access to the same data or structures. When the thread has finished processing the critical data or structures, it must release the lock to allow other threads to gain access to the data or structures. [POSIX.1]

time sharing option (TSO/E)
An option on the operating system; for System/370, the option provides interactive time sharing from remote terminals.

token See *condition token*.

trace A record of the execution of a computer program. It exhibits the sequence in which the instructions were executed. To record a series of events as they occur.

traceback
A section of a dump that provides information about the stack frame (DSA), the program unit address, the entry point of the routine, the statement number, and status of the routines on the call-chain at the time the traceback was produced.

translator
See *CICS translator*.

transient data queue
A file to which runtime messages are written under CICS. Under Language Environment, the name of this file is CESE. Also a sequential data set used by the Folder Application Facility in CICS/MVS to log system messages.

transient routines
The Language Environment library routines that are loaded at run time. Contrast with *resident routines*.

translator
See *CICS translator*.

TSO TSO/E.

TSO/E Time Sharing Option Extensions. An MVS component that permits interactive

compiling, link-editing, executing, and debugging of programs.

UCLIN
In SMP/E, the command used to initiate changes to SMP/E data sets. Actual changes are made by subsequent UCL statements.

underflow
See *exponent-underflow exception*.

unhandled condition
A condition that isn't handled by any condition handler for any stack frame in the call chain. Contrast with *handled condition*.

UNIX See *z/OS UNIX System Services*.

unpacked decimal format
A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1s (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. Synonymous with *zoned decimal format*.

upward-growing stack
With Extra Performance Linkage (XPLINK), a stack that grows from low addresses to high addresses in memory.

upwardly compatible
The ability for applications that have been linked with Language Environment to continue to run on later releases of OS/390 Language Environment, without the need to recompile or relink. Language Environment is guaranteed to be upwardly compatible.

user abend
A request made by user code to the operating system to abnormally terminate a routine. Contrast with *system abend*.

user-written condition handler
A routine that analyzes and possibly takes action on conditions presented to it by the condition manager. The condition handler is registered either by calling the CEEHDLR callable service or by specifying the USRHDLR runtime option.

user exit
A routine that takes control at a specific point in an application. Two assembler user exits and one HLL user exit are provided by Language Environment. They are invoked to perform initialization functions and both normal and abnormal termination functions.

user heap
See *initial heap*.

usermod
User modification.

user stack
An independent area of stack storage that may be located above or below 16M, designed to be used by both library routines and compiled code. See also *stack* and *stack frame*.

vector A linearly ordered collection of scalars of the same type. Each scalar is said to be an *element* of the vector. See also *array*. Contrast with *scalar*.

vector instruction
An instruction, such as a load, store, arithmetic, or logical instruction, that operates on vectors residing in storage or in a vector register in the vector facility. Contrast with *scalar instruction*.

vendor
A person or company that provides a service or product to another person or company.

virtual origin
The address of an element in an array whose subscripts are all zero.

VO Virtual origin.

void function
The C representation of a procedure invocation. A void function is a function that does not return a value.

VOLSER
Volume serial number.

volume
A certain portion of data, together with its data carrier, that can be handled conveniently as a unit. A data carrier mounted and demounted as a unit; for example, a reel of magnetic tape, a disk pack.

volume label
An area on a standard label tape used to identify the tape volume and its owner. This area is the first 80 bytes and contains VOL 1 in the first four positions.

volume serial number
A number in a volume label assigned when a volume is prepared for use in a system.

VSAM
Virtual storage access method. A high-performance mass storage access method. Three types of data organization are available: entry sequenced data sets (ESDS), key sequenced data sets (KSDS), and relative record data sets (RRDS).

VSTRING
The VSTRING data type is used for the character string parameters in many of the Language Environment callable services. In z/OS Language Environment, VSTRING is a halfword length-prefixed character string for input, or a fixed-length 80-character string for output.

weak external reference
A special type of external reference that is not to be resolved by automatic library calls unless an ordinary external reference to the same symbol is found. The external symbol dictionary entry specifies the symbol; the location is unknown.

work registers
Registers used by the PL/I compiler as required.

WORKING-STORAGE
In COBOL, the storage required for data items in the WORKING-STORAGE section. WORKING-STORAGE is a portion of main storage that is used by a computer program to hold data temporarily.

workstation
One or more programmable or nonprogrammable devices that allow a user to do work on a computer. See also *programmable workstation*.

writable static
In C, writable static may be any of the following:

- Program variables with the `extern` storage class
- Program variables with the `static` storage class
- Writable strings

The Language Environment term for writable static is *external data*.

XPG4 This term refers to the XPG4 interface standard. The XPG4 standard is described in detail in *X/Open Specification Issue 4*.

XPLINK (Extra Performance Linkage)
See *Extra Performance Linkage*.

zoned decimal format
Synonym for *unpacked decimal format*.

z/OS Language Environment
An element of z/OS that provides a common runtime environment and common runtime services for C/C++, COBOL, PL/I, and Fortran applications.

z/OS UNIX System Services (z/OS UNIX)
The set of functions provided by the Shell and Utilities, kernel, debugger, file system, C/C++ Runtime Library, Language Environment, and other elements of the z/OS operating system that allow users to write and run application programs that conform to UNIX standards.

31–bit mode
See AMODE 31.

64–bit virtual mode
See AMODE 64.

Index

Product Number: 5650-ZOS

Printed in USA

SA38-0687-00