z/OS Basic Skills Information Center

Application Programming on z/OS

z/OS Basic Skills Information Center

Application Programming on z/OS

Note
Before using this information and the product it supports, read the information in "Notices" on page 77.

This edition applies to z/OS (product number 5694-A01).

We appreciate your comments about this publication. Comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Send your comments through this Web site:

http://publib.boulder.ibm.com/infocenter/zoslnctr/v1r7/index.jsp?topic=/com.ibm.zcontact.doc/webqs.html

Contents

About application programming on z/OS

In this topic, we introduce the tools and utilities for developing a simple program to run on z/OS®. The sections that follow guide you through the process of application design, choosing a program language, and using a runtime environment.

Chapter 1. Designing and developing applications for z/OS

The life cycle of an application to run on z/OS begins with the requirement gathering phase, continues through design and code development, and ends with testing before the application enters into a steady state in which improvements or maintenance changes are applied.

This section provides a brief overview of a typical design, code, and test cycle for a new application on z/OS. Much of this information is applicable to all computing platforms in general, not just mainframes. This section also describes the roles of the application designer and application programmer. The discussion is intended to highlight the types of decisions that are involved in designing and developing an application to run in the mainframe environment. This is not to say that the process is much different on other platforms, but some of the questions and conclusions can be different.

This section describes the life cycle of designing and developing an application to run on z/OS. The process begins with the requirement gathering phase, in which the application designer analyzes user requirements to see how best to satisfy them. There might be many ways to arrive at a given solution; the object of the analysis and design phases is to ensure that the optimal solution is chosen. Here, "optimal" does not mean "quickest," although time is an issue in any project. Instead, optimal refers to the best overall solution, with regard to user requirements and problem analysis.

The EBCDIC character set is different from the ASCII character set. On a character-by-character basis, translation between these two character sets is trivial. When collating sequences are considered, the differences are more significant and converting programs from one character set to the other can be trivial or it can be quite complex. The EBCDIC character set became an established standard before the current 8-bit ASCII character set had significant use.

At the end of the design phase, the programmer's role takes over. The programmer must now translate the application design into error-free program code. Throughout the development phase, the programmer tests the code as each module is added to the whole. The programmer must correct any logic problems that are detected and add the updated modules to the completed suite of tested programs.

An application rarely exists in isolation. Rather, an application is usually part of a larger set of applications, where the output from one application is the input to the next application. To verify that a new application does not cause problems when incorporated into the larger set of applications, the application programmer conducts a system test or integration test. These tests are themselves designed, and many test results are verified by the actual application users. If any problems are found during system test, they must be resolved and the test repeated before the process can proceed to the next step.

Following a successful system test, the application is ready to go into production. This phase is sometimes referred to as promoting an application. Once promoted, the application code is now more closely controlled. A business would not want to introduce a change into a working system without being sure of its reliability. At

most z/OS sites, strict rules govern the promotion of applications (or modules within an application) to prevent untested code from contaminating a "pure" system.

At this point in the life cycle of an application, it has reached a steady state. The changes that will be made to a production application are enhancements, functional changes (for example, tax laws change, so payroll programs need to change), or corrections.

Application designers and programmers

The application designer gathers requirements from business systems analysts and end users. The application programmer is responsible for developing and maintaining application programs.

The tasks of **designing** an application and **developing** one are distinct enough to treat each in a separate learning module. In larger z/OS sites, separate departments might be used to carry out each task. This section provides an overview of these job roles and shows how each skill fits into the overall view of a typical application development life cycle on z/OS.

The application designer is responsible for determining the best programming solution for an important business requirement. The success of any design depends in part on the designer's knowledge of the business itself, awareness of other roles in the mainframe organization such as programming and database design, and understanding of the business's hardware and software. In short, the designer must have a global view of the entire project.

Another role involved in this process is the business systems analyst. This person is responsible for working with users in a particular department (accounting, sales, production control, manufacturing, and so on) to identify business needs for the application. Like the application designer, the business systems analyst requires a broad understanding of the organization's business goals, and the capabilities of the information system.

The application designer gathers requirements from business systems analysts and end users. The designer also determines which IT resources will be available to support the application. The application designer then writes the design specifications for the application programmers to implement.

The application programmer is responsible for developing and maintaining application programs. That is, the programmer builds, tests, and delivers the application programs that run on the mainframe for the end users. Based on the application designer's specifications, the programmer constructs an application program using a variety of tools. The build process includes many iterations of code changes and compiles, application builds, and unit testing.

During the development process, the designer and programmer must interact with other roles in the enterprise. The programmer, for example, often works on a team of other programmers who are building code for related application modules.

When the application modules are completed, they are passed through a testing process that can include functional, integration, and system tests. Following this testing process, the application programs must be acceptance-tested by the user community to determine whether the code actually accomplishes what the users desire.

Besides creating new application code, the programmer is responsible for maintaining and enhancing the company's existing mainframe applications. In fact, this is frequently the primary job for many application programmers on the mainframe today. While many mainframe installations still create new programs with COBOL or PL/I, languages such as Java™ have become popular for building new applications on the mainframe, just as on distributed platforms.

Designing an application for z/OS

During the early design phases, the application designer makes decisions regarding the characteristics of the application. These decisions are based on many criteria, which must be gathered and examined in detail to arrive at a solution that is acceptable to the user. The decisions are not independent of each other, in that one decision will have an impact on others and all decisions must be made taking into account the scope of the project and its constraints.

Designing an application to run on z/OS shares many of the steps followed for designing an application to run on other platforms, including the distributed environment. z/OS, however, introduces some special considerations. This section provides some examples of the decisions that the z/OS application designer makes during the design process for a given application. The list is not meant to be exhaustive, but rather to give you an idea of the process involved:

- Designing for z/OS: Batch or online?
- Designing for z/OS: Data sources and access methods
- Designing for z/OS: Availability and workload requirements
- Designing for z/OS: Exception handling.

Beyond these decisions, other factors that might influence the design of a z/OS application might include the choice of one or more programming languages and development environments. Other considerations discussed in this section include the following:

- Using mainframe character sets
- Using an interactive development environment (IDE)
- Differences between the various programming languages.

Keep in mind that the best designs are those that start with the end result in mind. We must know what it is that we are striving for before we start to design.

Designing for z/OS: Batch or online?

When designing an application for z/OS and the mainframe, a key consideration is whether the application will run as a batch program or an online program. In some cases, the decision is obvious, but most applications can be designed to fit either paradigm. How, then, does the designer decide which approach to use?

Reasons for using batch or online:

- Reasons for using batch:
  - Data is stored on tape.
  - Transactions are submitted for overnight processing.
  - User does not require online access to data.
- Reasons for using online:
  - User requires online access to data.

– High response time requirements.

Designing for z/OS: Data sources and access methods

Here, the designer's considerations typically include the following:

- What data must be stored?
- How will the data be accessed? This includes a choice of access method.
- Are the requests ad hoc or predictable?
- Will we choose PDS, VSAM, or a database management system (DBMS) such as DB2®?

Designing for z/OS: Availability and workload requirements

For an application that will run on z/OS, the designer must be able to answer the following questions:

- What is the quantity of data to store and access?
- Is there a need to share the data?
- What are the response time requirements?
- What are the cost constraints of the project?
- How many users will access the application at once?

What is the availability requirement of the application (24 hours a day 7 days a week or 8:00 AM to 5:00 PM weekdays, and so on)?

Designing for z/OS: Exception handling

Are there any unusual conditions that might occur? If so, we need to incorporate these in our design in order to prevent failures in the final application. We cannot always assume, for example, that input will always be entered as expected.

Application development life cycle: An overview

An application is a collection of programs that satisfies certain specific requirements (resolves certain problems). The solution could reside on any platform or combination of platforms, from a hardware or operating system point of view.

As with other operating systems, application development on z/OS is usually composed of the following phases:

- Design phase
- Gather requirements.
  - User, hardware and software requirements
  - Perform analysis.
  - Develop the design in its various iterations:
    - High-level design
    - Detailed design
  - Hand over the design to application programmers.
- Code and test application.
- Perform user tests.

User tests application for functionality and usability.

- Perform system tests.
  - Perform integration test (test application with other programs to verify that all programs continue to function as expected).
  - Perform performance (volume) test using production data.
- Go into production–hand off to operations.
- Ensure that all documentation is in place (user training, operation procedures).
- Maintenance phase--ongoing day-to-day changes and enhancements to application.

Figure 1 shows the process flow during the various phases of the application development life cycle.

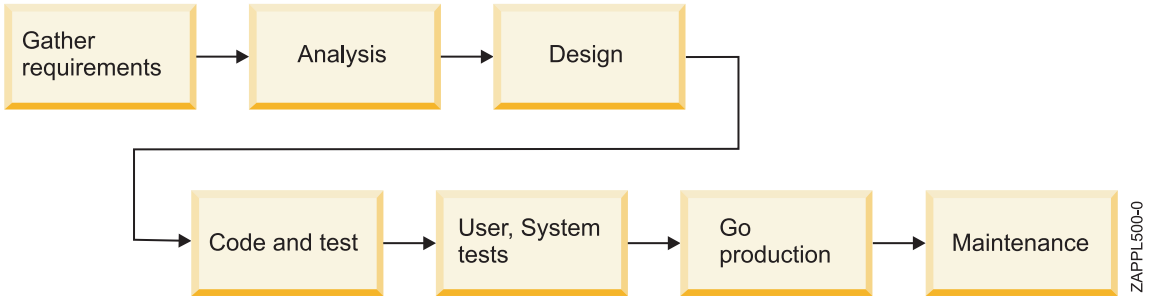

Figure 1. Application development life cycle

Figure 2 depicts the design phase up to the point of starting development. Once all of the requirements have been gathered, analyzed, verified, and a design has been produced, we are ready to pass on the programming requirements to the application programmers.

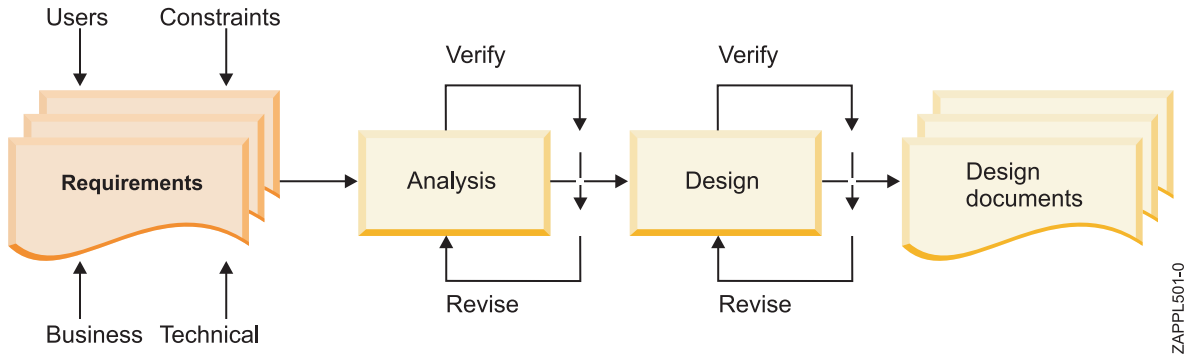

Figure 2. Design phase

The programmers take the design documents (programming requirements) and then proceed with the iterative process of coding, testing, revising, and testing again, as we see in Figure 3 on page 6.

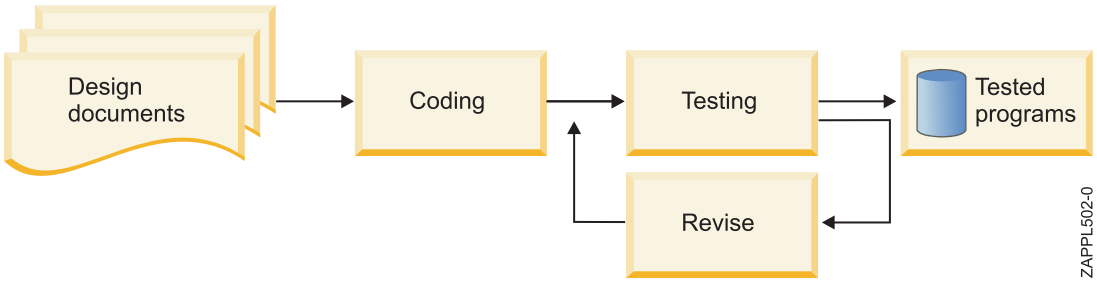

Figure 3. Development phase

After the programs have been tested by the programmers, they will be part of a series of formal user and system tests. These are used to verify usability and functionality from a user point of view, as well as to verify the functions of the application within a larger framework (Figure 4).

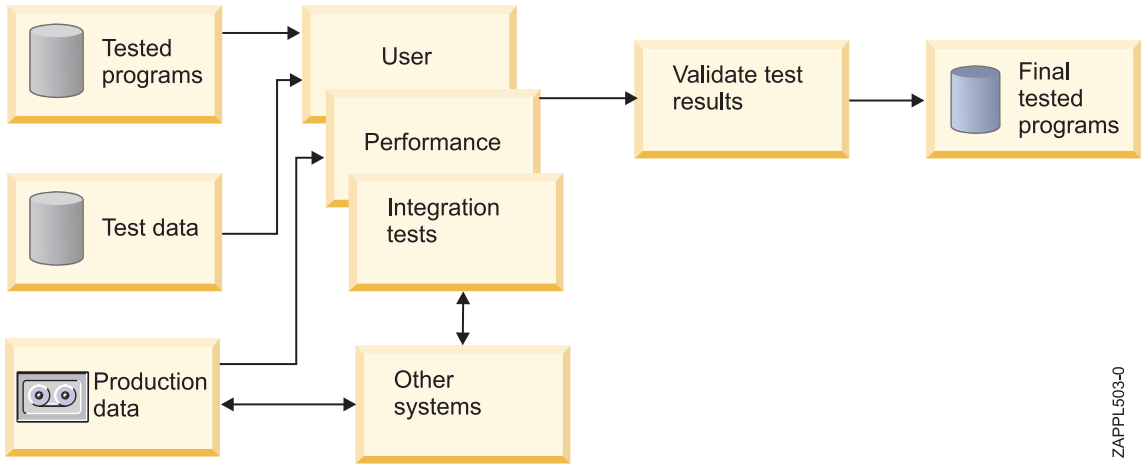

Figure 4. Testing

The final phase in the development life cycle is to go to production and become steady state. As a prerequisite to going to production, the development team needs to provide documentation. This usually consists of user training and operational procedures. The user training familiarizes the users with the new application. The operational procedures documentation enables Operations to take over responsibility for running the application on an ongoing basis.

In production, the changes and enhancements are handled by a group (possibly the same programming group) that performs the maintenance. At this point in the life cycle of the application, changes are tightly controlled and must be rigorously tested before being implemented into production (Figure 5).

Figure 5. Production

As mentioned before, to meet user requirements or solve problems, an application solution might be designed to reside on any platform or a combination of platforms. As shown in Figure 6, our specific application can be located in any of the three environments: Internet, enterprise network, or central site. The operating system must provide access to any of these environments.

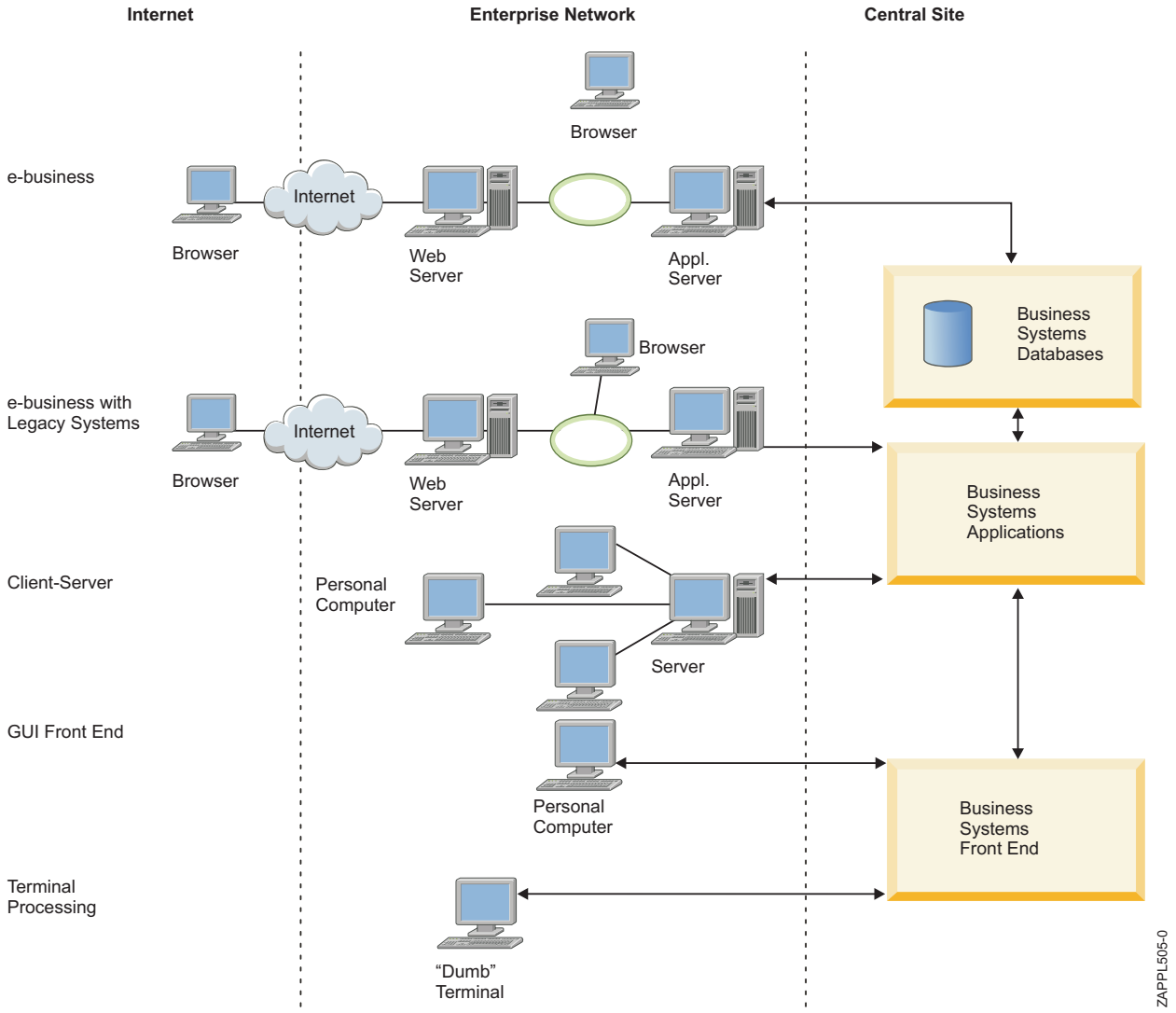

Figure 6. Growing infrastructure complexity

To begin the design process, we must first assess what we need to accomplish. Based on the constraints of the project, we determine how and with what we will accomplish the goals of the project. To do so, we conduct interviews with the users (those requesting the solution to a problem) as well as the other stakeholders.

The results of these interviews should inform every subsequent stage of the life cycle of the application project. At certain stages of the project, we again call upon the users to verify that we have understood their requirements and that our solution meets their requirements. At these milestones of the project, we also ask the users to sign off on what we have done, so that we can proceed to the next step of the project.

Gathering requirements for the design

When designing applications, there are many ways to classify the requirements: Functional requirements, non-functional requirements, emerging requirements, system requirements, process requirements, constraints on the development and on the operation--to name a few.

Computer applications operate on data, which resides somewhere and which needs to be accessed from either a local or remote location. The applications manipulate the data, performing some kind of processing on it, and then present the results to whomever was asking for in the first place.

This simple description involves many processes and many operations that have many different requirements, from computers to software products.

Although each application design is a separate case and can have many unique requirements, some of these are common to all applications that are part of the same system. Not only because they are part of the same set of applications that comprise a given information system, but also because they are part of the same installation, which is connected to the same external systems.

One of the problems faced by systems as a whole is that components are spread across different machines, different platforms, and so forth, each one performing its work in a **server farm** environment.

An important advantage to the zSeries® approach is that applications can be maintained using tools that reside on the mainframe. Some of these mainframe tools make it possible to have different platforms sharing resources and data in a coordinated and secure way according to workload or priority.

The following is a list of the various types of requirements for an application. The list is not exclusive; some items already include others.

- Accessibility
- Recoverability
- Serviceability
- Availability
- Security
- Connectivity
- Performance objectives
- Resource management
- Usability
- Frequency of data backup
- Portability
- Web services
- Changeability
- Inter-communicable
- Failure prevention and fault analysis

Developing an application on the mainframe

After the analysis has been completed and the decisions have been made, the process passes on to the application programmer. The programmer is not given free reign, but rather must adhere to the specifications of the designer. However, given that the designer is probably not a programmer, there may be changes required because of programming limitations. But at this point in the project, we are not talking about design changes, merely changes in the way the program does what the designer specified it should do.

The development process is iterative, usually working at the module level. A programmer will usually follow this process:

1. Code a module.
2. Test a module for functionality.
3. Make corrections to the module.
4. Repeat from step 2 until successful.

After testing has been completed on a module, it is signed off and effectively frozen to ensure that if changes are made to it later, it will be tested again. When sufficient modules have been coded and tested, they can be tested together in tests of ever-increasing complexity.

This process is repeated until all of the modules have been coded and tested. Although the process diagram shows testing only after development has been completed, testing is continuously occurring during the development phase.

The EBCDIC character set

z/OS data sets are encoded in the Extended Binary Coded Decimal Interchange (EBCDIC) character set. This is a character set that was developed before ASCII (American Standard Code for Information Interchange) became commonly used.

Most systems that you are familiar with use ASCII. In addition, z/OS UNIX® files are encoded in ASCII.

You need to be aware of the differences in encoding schemes when moving data from ASCII-based systems to EBCDIC-encoded systems. Generally the conversion is handled internally, for example when text is sent from a 3270 emulator running on a PC to a TSO session. However, when transferring programs, these must not normally be translated and a binary transfer must be specified. Occasionally, even when transferring text, there are problems with certain characters such as the OR sign (|) or the logical NOT sign, and the programmer must look at the actual value of the translated character.

ASCII and EBCDIC are both 8-bit character sets. The difference is the way they assign bits for specific characters. The following are a few examples:

Character	EBCDIC	ASCII
A	11000001 (x'C1')	01000001 (x'41')
B	11000010 (x'C2')	01000010 (x'42')
a	10000001 (x'81')	01100001 (x'61')
1	11110001 (x'F1')	00110001 (x'31')
space	01000000 (x'40')	00100000 (x'20')

Although the ASCII arrangement might seem more logical, the huge amount of existing data in EBCDIC and the large number of programs that are sensitive to the character set make it impractical to convert all existing data and programs to ASCII.

A character set has a collating sequence, corresponding to the binary value of the character bits. For example, A has a lower value than B in both ASCII and EBCDIC. The collating sequence is important for sorting and for almost any program that scans and manipulates character strings. The general collating sequence for common characters in the two character sets is as follows:

	EBCDIC	**ASCII**
Lowest value:	space	space
	punctuation	punctuation
	lower case	numbers
	upper case	upper case
Highest value:	numbers	lower case

For example, “a” is less than “A” in EBCDIC, but “a” is greater than “A” in ASCII. Numeric characters are less than any alphabetic letter in ASCII but are greater than any letter in EBCDIC. A-Z and a-z are two contiguous sequences in ASCII. In EBCDIC there are gaps between some letters. If we subtract A from Z in ASCII we have 25. If we subtract A from Z in EBCDIC we have 40 (due to the gaps in binary values between some letters).

Converting simple character strings between ASCII and EBCDIC is trivial. The situation is more difficult if the character being converted is not present in the standard character set of the target code. A good example is a logical not symbol that is used in a major mainframe programming language (PL/I); there is no corresponding character in the ASCII set. Likewise, some ASCII characters used for C programming were not present in the original EBCDIC character set, although these were later added to EBCDIC. There is still some confusion about the cent sign (¢) and the hat symbol (^), and a few more obscure symbols.

Mainframes also use several versions of double-byte character sets (DBCS), mostly for Asian languages. The same character sets are used by some PC programs.

Traditional mainframe programming does not use special characters to terminate fields. In particular, nulls and new line characters (or CL/LF character pairs) are not used. There is no concept of a **binary** versus a **text** file. Bytes can be interpreted as EBCDIC or ASCII or something else if programmed properly. If such files are sent to a mainframe printer, it will attempt to interpret them as EBCDIC characters because the printer is sensitive to the character set. The z/OS Web server routinely stores ASCII files because the data will be interpreted by a PC browser program that expects ASCII data. Providing that no one attempts to print the ASCII files on a mainframe printer (or display them on a 3270), the system does not care what character set is being used.

Unicode on the mainframe

Unicode, an industry standard, is a 16-bit character set intended to represent text and symbols in all modern languages and I/T protocols. Mainframes (using EBCDIC for single-byte characters), PCs, and various RISC systems use the same Unicode assignments.

Unicode is maintained by the Unicode Consortium.

There is increasing use of Unicode in mainframe applications. The latest zSeries mainframes include a number of unique hardware instructions for Unicode. At the time of writing, Unicode usage on mainframes is primarily in Java. However, z/OS middleware products are also beginning to use Unicode, and this is certainly an area of change for the near future.

Related information

http://www.unicode.org/

Interfaces for z/OS application programmers

When operating systems are developed to meet the needs of the computing marketplace, applications are written to run on those operating systems. Over the years, many applications have been developed that run on z/OS and, more recently, UNIX. To accommodate customers with UNIX applications, z/OS contains a full UNIX operating system in addition to its traditional z/OS interfaces. The z/OS implementation of UNIX interfaces is known collectively as z/OS UNIX System Services, or z/OS UNIX for short.

The most common interface for z/OS developers is TSO/E and its panel-driven interface, ISPF, using a 3270 terminal. Generally, developers use 3270 terminal emulators running on personal computers, rather than actual 3270 terminals. Emulators can provide developers with auxiliary functions, such as multiple sessions, and uploading and downloading code and data from the PC.

Program development on z/OS typically involves the use of a line editor to manipulate source code files, the use of batch jobs for compilation, and a variety of mechanisms for testing the code. Interactive debuggers, based on 3270 terminal functions, are available for common languages. This section introduces the tools and utilities for developing a simple program to run on z/OS.

Development using only the z/OS UNIX portion of z/OS can be through Telnet sessions (from which the vi editor is available) through 3270 and TSO/E using other editors, or through X Window System sessions from personal computers running X servers. The X server interfaces are less commonly used.

Alternate methods are available in conjunction with various middleware products. For example, the WebSphere® products provide GUI development facilities for personal computers. These facilities integrate TCP/IP links with z/OS to automatically invoke mainframe elements needed during development and testing phases for a new application.

Application development tools

Producing well-tested code requires the use of tools on the mainframe.

The primary tool for the programmer is the ISPF editor. When developing traditional, procedural programs in languages such as COBOL and PL/I, the programmer often logs on to the mainframe and uses an Interactive Development Environment (IDE) or the ISPF editor to modify the code, compile it, and run it. The programmer uses a common repository (such as the IBM® Software Configuration Library Manager or SCLM) to store code that is under development. The repository allows the programmer check code in or out, and ensures that programmers do not interfere with each others' work. SCLM is included with ISPF as an option from the main menu.

For purposes of simplicity, the source code could be stored and maintained in a partitioned data set (PDS). However, using a PDS would neither provide change control nor prevent multiple updates to the same version of code in the way that SCLM would. So, wherever we have written “checking out” or “saving” to SCLM, assume that you could substitute this with “edit a PDS member” or “save a PDS member.”

When the source code changes are complete, the programmer submits a JCL file to compile the source code, bind the application modules, and create an executable for testing. The programmer conducts "unit tests" of the functionality of the program. The programmer uses job monitoring and viewing tools to track the running programs, view the output, and make appropriate corrections to source code or other objects. Sometimes, a program will create a "dump" of memory when a failure occurs. The programmer can also use tools to interrogate the dump output and to trace through executing code to identify the failure points.

Some mainframe application programmers have now switched to the use of IDE tools to accelerate the edit/compile/test process. IDEs allow application programmers to edit, test, and debug source code on a workstation instead of directly on the mainframe system. The use of an IDE is particularly useful for building “hybrid” applications that employ host-based programs or transactional systems, but also contain a Web browser-like user interface.

After the components are developed and tested, the application programmer packages them into the appropriate deployment format and passes them to the team that coordinates production code deployments.

Application enablement services available on z/OS include:

- Language Environment®
- C/C++ IBM Open Class® Library
- DCE Application Support1
- Encina Toolkit Executive2
- C/C++ with Debug Tool
- DFSORT
- GDDM-PGF
- GDDM-REXX
- HLASM Toolkit
- Traditional languages such as COBOL, PL/I, and Fortran

Conducting a debugging session

The application programmer conducts a “unit test” to test the functionality of a particular module being developed. The programmer uses job monitoring and viewing software such as SDSF to track the running compile jobs, view the compiler output, and verify the results of the unit tests. If necessary, the programmer makes the appropriate corrections to source code or other objects.

Sometimes, a program will create a “dump” of memory when a failure occurs. When this happens, a z/OS application programmer might use tools such as IBM Debug Tool and IBM Fault Analyzer to interrogate the dump output and to trace through executing code to find the failure or misbehaving code.

A typical development session follows these steps:

1. Log on to z/OS.

2. Enter ISPF and open/check out source code from the SCLM repository (or PDS).
3. Edit the source code to make necessary modifications.
4. Submit JCL to build the application and do a test run.
5. Switch to SDSF to view the running job status.
6. View the job output in SDSF to check for errors.
7. View the dump output to find bugs.[1]
8. Re-run the compile/link/go job and view the status.
9. Check the validity of the job output.
10. Save the source code in SCLM (or PDS).

Some mainframe application programmers have now switched to the use of Interactive Development Environment (IDE) tools to accelerate the edit/compile/test process. IDE tools such as the WebSphere Studio Enterprise Developer are used to edit source code on a workstation instead of directly on the host system, to run compiles "off-platform," and to perform remote debugging.

The use of the IDE is particularly useful if hybrid applications are being built that employ host-based programs in COBOL or transaction systems such as CICS® and IMS™, but also contain a Web browser-like user interface. The IDE provides a unified development environment to build both the online transaction processing (OLTP) components in a high-level language and the HTML front-end user interface components. Once the components are developed and tested, they are packaged into the appropriate deployment format and passed to the team that coordinates production code deployments.

Besides new application code, the application programmer is responsible for the maintenance and enhancement of existing mainframe applications. In fact, this is the primary job for many high-level language programmers on the mainframe today. And, while most z/OS customers are still creating new programs with COBOL or PL/I, languages such as Java have become popular for building new applications on the mainframe, just as on distributed platforms.

However, for those of us interested in the traditional languages, there is still widespread development of programs on the mainframe in high-level languages such as COBOL and PL/I. There are many thousands of programs in production on mainframe systems around the world, and these programs are critical to the day-to-day business of the corporations that use them. COBOL and other high-level language programmers are needed to maintain existing code and make updates and modifications to those programs.

Also, many corporations continue to build new application logic in COBOL and other traditional languages, and IBM continues to enhance the high-level language compilers to include new functions and features that allow these languages to continue to exploit newer technologies and data formats.

1. The origin of the term “programming bug” is often attributed to US Navy Lieutenant Grace Murray Hopper in 1945. As the story goes, Lt. Hopper was testing the Mark II Aiken Relay Calculator at Harvard University. One day, a program that worked previously mysteriously failed. Upon inspection, the operator found that a moth was trapped between the circuit relay points and had created a short circuit (early calculators occupied many square feet, and consisted of tens of thousands of vacuum tubes). The September 9, 1945 log included both the moth and the entry: “First actual case of a bug being found,” and that they had “debugged the machine.”

Performing a system test

The difference between the testing done at this stage and the testing done during the development phase is that we are now testing the application as a whole, as well as in conjunction with other applications. We also carry out tests that can only be done once the application coding has been completed because we need to know how the whole application performs, and not just a portion of it.

The tests performed during this phase are:

- User testing--Testing the application for functionality and usability.
- Integration testing--The new application is tested together with other applications to see if they interface as expected.
- Performance or stress testing--The application is tested using real production data or at least real production data volume to see how well the application performs when there is high demand.

The results of the user and integration tests need to be verified to ensure that they are satisfactory. In addition, the performance of the application must match the requirements. Any issues coming out of these tests need to be addressed before going into production. The number of issues encountered during the testing phase are a good indication of how well we did our design work.

Going into production on the mainframe

The act of "going into production" is not simply turning on a switch that says now the application is production-ready. It is much more complicated than that. And from one project to the next, the way in which a program goes into production can change.

In some cases, where we have an existing system that we are replacing, we might decide to run in parallel for a period of time prior to switching over to the new application. In this case, we run both the old and the new systems against the same data and then compare the results. If after a certain period of time we are satisfied with the results, we switch to the new application. If we discover problems, we can correct them and continue the parallel run until there aren't any new problems.

In other cases, we are dealing with a new system, and we might just have a cut-over day when we start using it. Even in the case of a new system, we are usually replacing some form of system, even if it's a manual system, so we could still do a parallel test if we wanted to.

Whichever method is used to go into production, there are still all of the loose ends that need to be taken care of before we hand the system over to Operations. One of the tasks is to provide documentation for the system, as well as procedures for running and using it. We need to train everyone who interacts with the system.

When all of the documentation and training has been done, we can hand over responsibility for the running of the application to Operations and responsibility for maintaining the application to the Maintenance group. In some cases, the Development group also maintains applications.

At this point, the application development life cycle reaches a steady state and we enter the maintenance phase of the application. From this point onward, we only apply enhancements and day-to-day changes to the application. Because the application now falls under a change control process, all changes require testing

according to the process for change control, before they are accepted into production. In this way, a stable, running application is ensured for end users.

Chapter 2. Programming languages on z/OS

A critical factor in choosing a language is determining which one is most used at a given installation.

This section outlines the many decisions you might need to make when you design and develop an application for z/OS. Selecting a programming language to use is one important step in the design phase of an application. The application designer must be aware of the strengths as well as the weaknesses of each language to make the best choice, based on the particular requirements of the application.

A critical factor in choosing a language is determining which one is most used at a given installation. If COBOL is used for most of the applications in an installation, it will likely be the language of choice for the installation's new applications as well.

Understand that even when a choice for the primary language is made, however, it does not mean that you are locked into that choice for all programs within the application. There might be a case for using multiple languages, to take advantage of the strengths of a particular language for only certain parts of the application. Here, it might be best to write frequently invoked subroutines in Assembler language to make the application as efficient as possible, even when the rest of the application is written in COBOL or another high-level language.

Many z/OS sites maintain a library of subroutines that are shared across the business. The library might include, for example, date conversion routines. As long as these subroutines are written using standard linkage conventions, they can be called from other languages, regardless of the language in which the subroutines are written.

Each language has its inherent strengths, and designers should exploit these strengths. If a given application merits the added complication of writing it in multiple languages, the designer should take advantage of the particular features of each language. Keep in mind, however, that when it is time to update the application, other people must be able to program these languages as well. This is a cardinal rule of programming. The original programmer might be long gone, but the application will live on and on.

Thus, complexity in design must always be weighed against ease of maintenance.

Programming languages on the mainframe

A computer language is the way that a human communicates with a computer. It is needed because a computer works only with its machine language (bits and bytes). This is slow and cumbersome for humans to use. Therefore, we write programs in a computer language, which then gets converted into machine language for the computer to process.

There are many computer languages, and they have been evolving from machine language into a more natural way of writing. Some languages have been adapted to the kind of application that they intended to solve and to the kind of approach used in the design. The word **generation** has been used to indicate this evolution.

A classification of computer languages follows.

1. Machine language, the 1st generation, direct machine code.
2. Assembler, 2nd generation, using mnemonics to present the instructions to be translated later into machine language by an assembly program, such as Assembler language.
3. Procedural languages, 3rd generation, also known as high-level languages (HLL), such as Pascal, FORTRAN, Algol, COBOL, PL/I, Basic, and C. The coded program, called a source program, has to be translated through a compilation step.
4. Non-procedural languages, 4th generation, also known as 4GL, used for predefined functions in applications for databases, report generators, queries, such as RPG, CSP, QMF™.
5. Visual Programming languages that use a mouse and icons, such as VisualBasic and VisualC++.
6. HyperText Markup Language, used for writing of World Wide Web documents.
7. Object-oriented language, OO technology, such as Smalltalk, Java, and C++.
8. Other languages, for example 3D applications.

Each computer language evolved separately, driven by the creation of and adaptation to new standards. In the following sections we describe several of the most widely used computer languages supported by z/OS:

- Assembler
- COBOL
- PL/I
- C/C++
- Java
- CLIST
- REXX.

To this list, we can add the use of shell script and PERL in the z/OS UNIX System Services environment.

For the computer languages under discussion, we have listed their evolution and classified them. There are procedural and non-procedural, compiled and interpreted, and machine-dependent and non-machine-dependent languages.

Assembler language programs are **machine-dependent**, because the language is a symbolic version of the machine's language on which the program is running. Assembler language instructions can differ from one machine to another, so an Assembler language program written for one machine might not be portable to another. Rather, it would most likely need to be rewritten to use the instruction set of the other machine. A program written in a high-level language (HLL) would run on other platforms, but it would need to be recompiled into the machine language of the target platform.

Most of the HLLs that we touch upon in this section are **procedural languages**. This type is well-suited to writing structured programs. The **non-procedural languages**, such as SQL and RPG, are more suited for special purposes, such as report generation.

Most HLLs are compiled into machine language, but some are interpreted. Those that are compiled result in machine code which is very efficient for repeated

executions. Interpreted languages must be parsed, interpreted, and executed each time that the program is run. The trade-off for using interpreted languages is a decrease in programmer time, but an increase in machine resources.

The advantages of compiled and interpreted languages are further explored in this section.

Choosing a programming language for z/OS

Choosing a programming language depends on several things.

In developing a program to run on z/OS, your choice of a programming language might be determined by the following considerations:

- What type of application?
- What are the response time requirements?
- What are the budget constraints for development and ongoing support?
- What are the time constraints of the project?
- Do we need to write some of the subroutines in different languages because of the strengths of a particular language versus the overall language of choice?
- Do we use a compiled or an interpreted language?

The sections that follow look at considerations for several languages commonly supported on the mainframe.

The Assembler language on z/OS

Assembler language is a symbolic programming language that can be used to code instructions instead of coding in machine language.

The Assembler language is the symbolic programming language that is closest to the machine language in form and content, and therefore is an excellent candidate for writing programs in which:

- You need control of your program, down to the byte or bit level.
- You must write subroutines[2] for functions that are not provided by other symbolic programming languages, such as COBOL, FORTRAN, or PL/I.

Assembler language is made up of statements that represent either instructions or comments. The instruction statements are the working part of the language, and they are divided into the following three groups:

- A **machine instruction** is the symbolic representation of a machine language instruction of instruction sets, such as:
  - IBM Enterprise Systems Architecture/390 (ESA/390)
  - IBM z/Architecture®

  It is called a machine instruction because the assembler translates it into the machine language code that the computer can execute.
- An **assembler instruction** is a request to the assembler to do certain operations during the assembly of a source module; for example, defining data constants, reserving storage areas, and defining the end of the source module.

2. Subroutines are programs that are invoked frequently by other programs and by definition should be written with performance in mind. Assembler language is a good choice for a subroutine.

- A **macro instruction** or macro is a request to the assembler program to process a predefined sequence of instructions called a **macro definition**. From this definition, the assembler generates machine and assembler instructions, which it then processes as if they were part of the original input in the source module.

The assembler produces a program listing containing information that was generated during the various phases of the assembly process.[3] It is really a compiler for Assembler language programs.

The assembler also produces information for other processors, such as a **binder** (or **linker**, for earlier releases of the operating system). Before the computer can execute your program, the object code (called an object deck or simply OBJ) has to be run through another process to resolve the addresses where instructions and data will be located. This process is called **linkage-editing** (or **link-editing**, for short) and is performed by the binder.

The binder or linkage editor uses information in the object decks to combine them into load modules. At program fetch time, the load module produced by the binder is loaded into virtual storage. After the program is loaded, it can be run.

Figure 7 shows these steps.

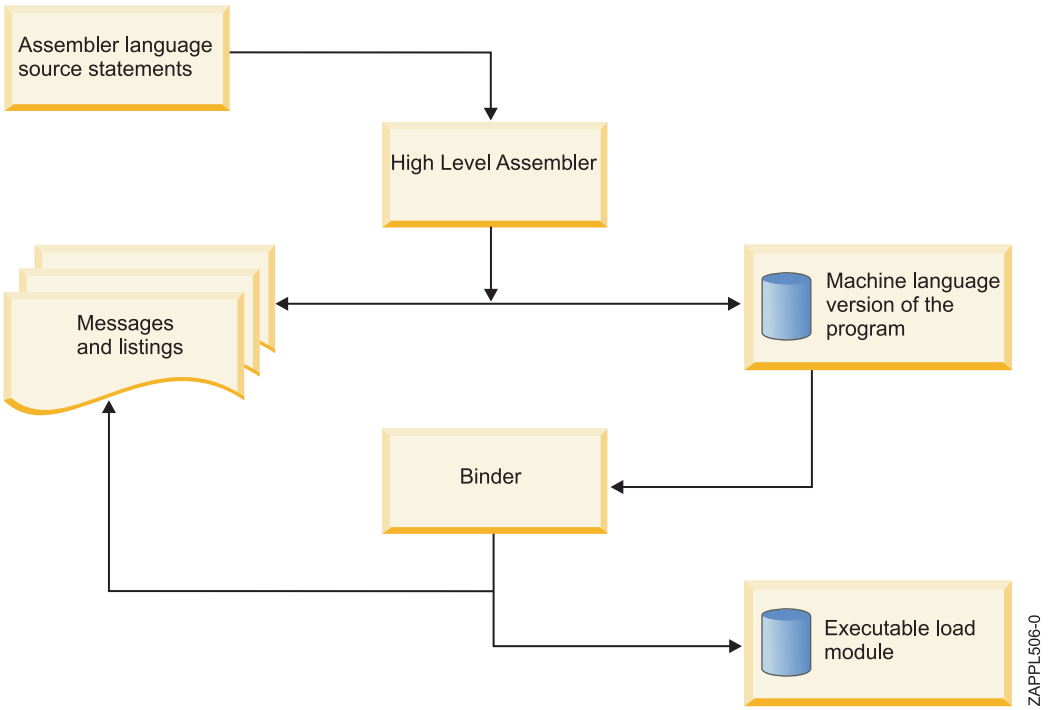

Figure 7. Assembler source to executable module

3. A program listing does not contain all of the information that is generated during the assembly process. To capture all of the information that could possibly be in the listing (and more), the z/OS programmer can specify an assembler option called ADATA to have the assembler produce a SYSADATA file as output. The SYSADATA file is not human-readable—its contents are in a form that is designed for a tool to process. The use of a SYSADATA file is simpler for tools to process than the older custom of extracting similar data through "listing scrapers."

Related reading: You can find more information about using Assembler language on z/OS in the IBM publications, *HLASM General Information*, GC26-4943, and *HLASM Language Reference*, SC26-4940.

More information about Assembler language

The assembler processes the machine and Assembler language instructions at different times during its processing sequence. The programmer should be aware of the assembler's processing sequence to code the program correctly.

Processing sequence

Processing involves the translation of source statements into machine language, assignment of storage locations to instructions and other elements of the program, and performance of auxiliary assembler functions you have designated.

The output of the assembler program is the **object deck**, a machine language translation of the source program. The assembler produces a printed listing of the source statements and object deck statements, as well as additional information such as error messages that are useful in analyzing the program. The object deck is in the format required by the binder.

The assembler processes most instructions twice, first during conditional assembly and later, at assembly time. However, as described in the following section, it does some processing only during conditional assembly.

Conditional assembly and macro instructions

The assembler processes conditional assembly instructions and macro processing instructions during conditional assembly. During this processing, the assembler evaluates arithmetic, logical, and character conditional assembly expressions. Conditional assembly takes place before assembly time.

The assembler processes the machine and ordinary assembler instructions generated from a macro definition called by a macro instruction at assembly time.

Machine instructions

The assembler processes all machine instructions, and translates them into object code at assembly time.

Assembler instructions

The assembler processes ordinary assembler instructions at assembly time. During this processing:

- The assembler evaluates absolute and relocatable expressions (sometimes called assembly-time expressions).
- Some instructions, such as ADATA, ALIAS, CATTR and XATTR, DC, DS, ENTRY, EXTRN, PUNCH, and REPRO, produce output for later processing by programs such as the binder.

Input/Output (I/O)

The terms input (I) and output (O) are used to describe the transfer of data between I/O devices and main storage. An operation involving this kind of

transfer is called an **I/O operation**. The facilities used to control I/O operations are collectively called the **channel subsystem** (I/O devices and their control units attached to the channel subsystem).

The channel subsystem directs the flow of information between I/O devices and main storage. It relieves CPUs of the task of communicating directly with I/O devices, and permits data processing to proceed concurrently with I/O processing.

I/O devices

An input/output (I/O) device provides external storage, a means of communication between data-processing systems, or a means of communication between a system and its environment. I/O devices include such equipment as magnetic tape units, direct access storage devices (for example, disks), display units, typewriter-keyboard devices, printers, teleprocessing devices, and sensor-based equipment. An I/O device may be physically distinct equipment, or it may share equipment with other I/O devices.

The term I/O device as used in this section refers to an entity with which the channel subsystem can directly communicate.

COBOL on z/OS

Common Business-Oriented Language (COBOL) is a programming language similar to English that is widely used to develop business-oriented applications in the area of commercial data processing.

COBOL has been almost a generic term for computer programming in this kind of computer language. However, as used in this section, COBOL refers to the product IBM Enterprise COBOL for z/OS and OS/390®.

In addition to the traditional characteristics provided by the COBOL language, this version of COBOL is capable, through COBOL functions, of integrating COBOL applications into Web-oriented business processes. With the capabilities of this release, application developers can do the following:

- Utilize new debugging functions in Debug Tool
- Enable interoperability with Java when an application runs in an IMS Java-dependent region
- Simplify the componentization of COBOL programs and enable interoperability with Java components across distributed applications
- Promote the exchange and usage of data in standardized formats including XML and Unicode.

With Enterprise COBOL for z/OS and OS/390, COBOL and Java applications can interoperate in the e-business world.

The COBOL compiler produces a program listing containing all the information that it generated during the compilation. The compiler also produces information for other processors, such as the binder.

Before the computer can execute your program, the object deck has to be run through another process to resolve the addresses where instructions and data will be located. This process is called **linkage edition** and is performed by the binder.

The binder uses information in the object decks to combine them into load modules. At program fetch time, the load module produced by the binder is loaded into virtual storage. When the program is loaded, it can then be run. Figure 8 illustrates the process of translating the COBOL source language statements into an executable load module.

This process is similar to that of Assembler language programs. In fact, this same process is used for all of the HLLs that are compiled.

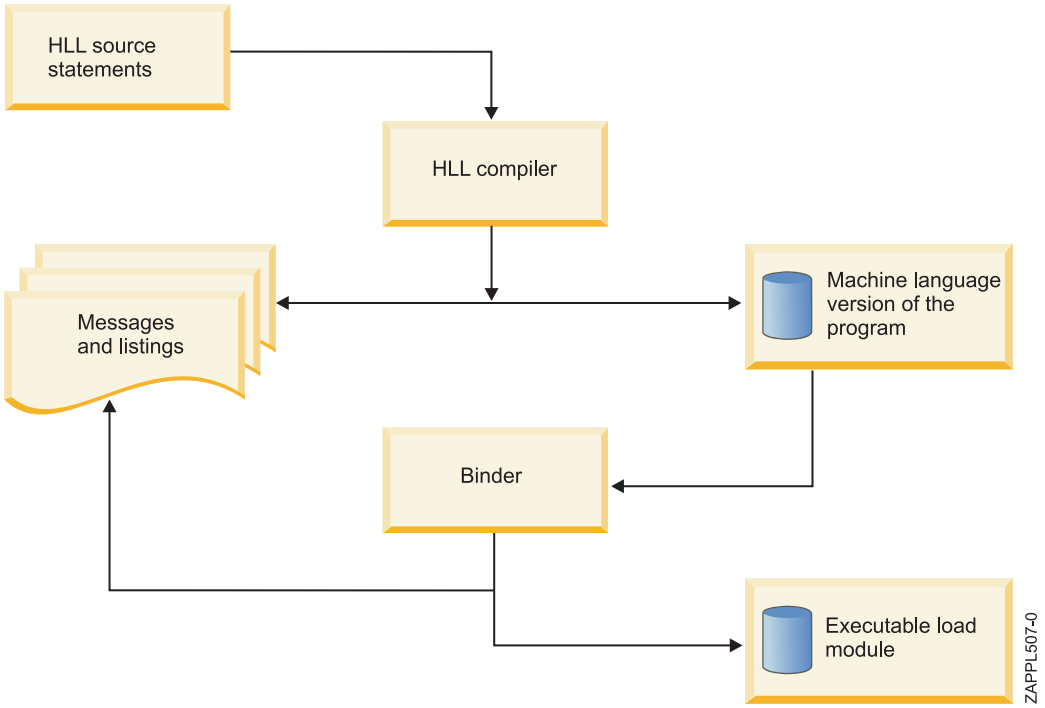

Figure 8. HLL source to executable module

COBOL program format

With the exception of the COPY and REPLACE statements and the end program marker, the statements, entries, paragraphs, and sections of a COBOL source program are grouped into four divisions.

- IDENTIFICATION DIVISION, which identifies the program with a name and, if you want, gives other identifying information.
- ENVIRONMENT DIVISION, where you describe the aspects of your program that depend on the computing environment.
- DATA DIVISION, where the characteristics of your data are defined in one of the following sections in the DATA DIVISION:
  - FILE SECTION, to define data used in input-output operations
  - LINKAGE SECTION, to describe data from another program.

  When defining data developed for internal processing:
  - WORKING-STORAGE SECTION, to have storage statically allocated and remain for the life of the run unit.
  - LOCAL-STORAGE SECTION, to have storage allocated each time a program is called and de-allocated when the program ends.
  - LINKAGE SECTION, to describe data from another program.

- PROCEDURE DIVISION, where the instructions related to the manipulation of data and interfaces with other procedures are specified.

  The PROCEDURE DIVISION of a program is divided into sections and paragraphs, which contain sentences and statements, as described here:

  – Section - a logical subdivision of your processing logic. A section has a section header and is optionally followed by one or more paragraphs. A section can be the subject of a PERFORM statement. One type of section is for declaratives.

    Declaratives are a set of one or more special purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word. DECLARATIVES and the last of which is followed by the key word END DECLARATIVES.
  – Paragraph - a subdivision of a section, procedure, or program. A paragraph can be the subject of a statement.
  – Sentence - a series of one or more COBOL statements ending with a period.
  – Statement - a defined step of COBOL processing, such as adding two numbers.
  – Phrase - a subdivision of a statement.

Example of COBOL divisions

```
 IDENTIFICATION DIVISION.
Program-ID. Helloprog.
Author. A. Programmer.
Installation.  Computing Laboratories.
Date-Written.  08/21/2002.
```

Figure 9. IDENTIFICATION DIVISION

Example of input-output coding

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. computer-name.
OBJECT-COMPUTER. computer-name.
SPECIAL-NAMES.
  special-names-entries.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT [OPTIONAL] file-name-1
        ASSIGN TO system-name [FOR MULTIPLE {REEL | UNIT}]
        [.... .
I-O-CONTROL.
    SAME [RECORD] AREA FOR file-name-1 ... file-name-n.
```

Figure 10. ENVIRONMENT DIVISION

Explanations of the user-supplied information follow Figure 11 on page 25.

```
 IDENTIFICATION DIVISION.
. . .
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name
    ORGANIZATION IS org ACCESS MODE IS access
    FILE STATUS IS file-status
. . .
DATA DIVISION.
FILE SECTION.
FD  filename
01  recordname
    nn . . . fieldlength & type
    nn . . . fieldlength & type
. . .
WORKING-STORAGE SECTION
01 file-status PICTURE 99.
. . .
PROCEDURE DIVISION.
    . . .
    OPEN iomode filename
    . . .
    READ filename
    . . .
    WRITE recordname
    . . .
    CLOSE filename
    . . .
    STOP RUN.
```

Figure 11. Input and output files in FILE-CONTROL

- *org* indicates the organization, which can be SEQUENTIAL, LINE SEQUENTIAL, INDEXED, or RELATIVE.
- *access* indicates the access mode, which can be SEQUENTIAL, RANDOM, or DYNAMIC.
- *iomode* is for INPUT or OUTPUT mode. If you are only reading from a file, code INPUT. If you are only writing to it, code OUTPUT or EXTEND. If you are both reading and writing, code I-O, except for organization LINE SEQUENTIAL.
- Other values like *filename*, *recordname*, *fieldname* (nn in the example), *fieldlength* and type are also specified.

COBOL relationship between JCL and program files

We can achieve device independence through a combination of JCL statements and the COBOL program.

Figure 12 on page 26 depicts the relationship between JCL statements and the files in a COBOL program. By not referring to physical locations of data files in a program, we achieve device independence. That is, we can change where the data resides and what it is called without having to change the program. We would only need to change the JCL.

```
//MYJOB    JOB
//STEP1    EXEC IGYWCLG
...
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT INPUT ASSIGN TO INPUT1 .....
     SELECT DISKOUT ASSIGN TO OUTPUT1 ...
   FILE SECTION.
     FD INPUT1
        BLOCK CONTAINS...
        DATA RECORD IS RECORD-IN
     01 INPUT-RECORD
...
     FD OUTPUT1
        DATA RECORD IS RECOUT
     01 OUTPUT-RECORD
...
/*
//GO.INPUT1  DD DSN=MY.INPUT,DISP=SHR
//GO.OUTPUT1 DD DSN=MY.OUTPUT,DISP=OLD
```

Figure 12. COBOL relationship between JCL and program files

Figure 12 shows a COBOL compile, link, and go job stream, listing the file program statements and the JCL statements to which they refer.

The COBOL SELECT statements create the links between the DDNAMEs INPUT1 and OUTPUT1, and the COBOL FDs INPUT1 and OUTPUT1, respectively. The COBOL FDs are associated with group items INPUT-RECORD and OUTPUT-RECORD.

The DD cards INPUT1 and OUTPUT1 are related to the data sets MY.INPUT and MY.OUTPUT, respectively. The end result of this linkage in our example is that records read from the file INPUT1 will be read from the physical data set MY.INPUT and records written to the file OUTPUT1 will be written to the physical data set MY.OUTPUT. The program is completely independent of the location of the data and the name of the data sets.

Figure 13 shows the relationship between the physical data set, the JCL, and the program for Figure 12.

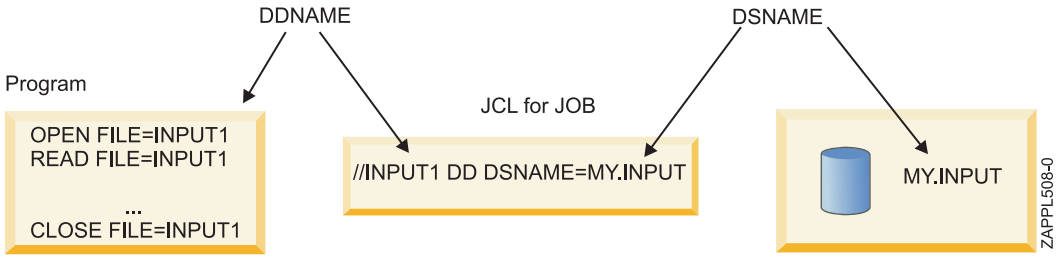

Figure 13. Relationship between JCL, program, and data set

Again, because the program does not make any reference to the physical data set, we would not need to recompile the program if the name of the data set or its location were to change.

Running COBOL programs under UNIX

To run COBOL programs in the UNIX environment, you must compile them with the Enterprise COBOL or the COBOL for OS/390 and VM compiler. They must be reentrant, so use the compiler and binder option RENT.

Communicating with Java methods

A COBOL program can interoperate with JAVA.

To achieve inter-language interoperability with Java, you must follow certain rules and guidelines for:

- Using services in the Java Native Interface (JNI)
- Coding data types
- Compiling your COBOL programs

You can invoke methods that are written in Java from COBOL programs, and you can invoke methods that are written in COBOL from Java programs. For basic Java object capabilities, you can use COBOL object-oriented language. For additional Java capabilities, you can call JNI services.

Because Java programs might be multi-threaded and use asynchronous signals, compile your COBOL programs with the THREAD option.

Creating a DLL or a DLL application

A dynamic link library or DLL is a file that contains executable code and data that is bound to a program at run-time. The code and data in a DLL can be shared by several applications simultaneously. Creating a DLL or a DLL application is similar to creating a regular COBOL application. It involves writing, compiling, and linking your source code.

Special considerations when writing a DLL or a DLL application include:

- Determining how the parts of the load module or the application relate to each other or to other DLLs
- Deciding what linking or calling mechanisms to use

Depending on whether you want a DLL load module or a load module that references a separate DLL, you need to use slightly different compiler and binder options.

Structuring OO applications

You can structure applications that use object-oriented COBOL syntax in one of three ways.

An OO application can begin with:

- A COBOL program, which can have any name.
- A Java class definition that contains a method called main. You can run the application with the Java command, specifying the name of the class that contains main and zero or more strings as command-line arguments.
- A COBOL class definition that contains a factory method called main. You can run the application with the Java command, specifying the name of the class that contains main and zero or more strings as command-line arguments.

Related reading: For more information about using COBOL on z/OS, see the IBM publications *Enterprise COBOL for z/OS and OS/390 V3R2 Language Reference*, SC27-1408, and *Enterprise COBOL for z/OS and OS/390 V3R2 Programming Guide*, SC27-1412.

More information about the COBOL language

You should know about COBOL standards, input and output, data structures, and program types.

Standards

The COBOL language adheres to ISO standards.

Input and output files

Input and output files are defined in the INPUT-OUTPUT section of the ENVIRONMENT DIVISION.

Data structure: data items and group items

Related data items can be parts of a hierarchical data structure. A data item that does not have any subordinate items is called an **elementary data item**. A data item that is composed of subordinated data items is called a **group item**. A record can be either an elementary data item or a group of data items.

COBOL program types

A COBOL source program is a syntactically correct set of COBOL statements. COBOL programs can be of different types:

- Nested program, a program that is contained in another program.
- Object deck, a set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. An object deck is generally the machine language result of the operation of a COBOL compiler on a source program.
- Run unit, one or more object decks that interact with one another and that function at run time as an entity to provide problem solutions.
- Sibling program, a nested program that is at the same nesting level as another nested program in the same containing program.

Targeting COBOL programs for certain environments

COBOL programs can run in CICS, DB2, and IMS environments.

COBOL programs for CICS

COBOL programs that are written for CICS can run under CICS Transaction Server. CICS COBOL application programs that use CICS services must use the CICS command-level interface.

COBOL programs for a DB2 environment

In general, the structure of your COBOL program is the same whether or not you want it to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data, you must:

- Connect to a data server.

- Execute SQL statements. SQL statements must begin with EXEC SQL and end with END-EXEC.
- Disconnect from the data server.

To prepare the program for execution, follow one of these procedures:

- Precompile, compile, link, and bind the program the program.
- Compile the program with the SQL option, and then link and bind the program.

COBOL programs for IMS

With Enterprise COBOL, you can invoke IMS facilities using the following interfaces:

- AIBTDLI
- AERTDLI
- CBLTDLI call
- Language Environment callable service CEETDLI

You can also run object-oriented COBOL programs with IMS dependent regions that support Java. You can mix the object-oriented COBOL and Java languages in a single application.

HLL relationship between JCL and program files

Symbolic file names can help isolate your program from changes.

By using symbolic names in JCL, we learned how to isolate a COBOL program from changes in data set name and data set location. The technique of referring to physical files by a symbolic file name is not restricted to COBOL; it is used by all HLLs and even in Assembler language. See Figure 14 on page 30 for a generic HLL example of a program that references data sets through symbolic file names.

Isolating your program from changes to data set name and location is the normal objective. However, there could be cases when a program needs to access a specific data set at a specific location on a direct access storage device (DASD). This can be accomplished in Assembler language and even in some HLLs.

The practice of "hard-coding" data set names or other such information in a program is not usually considered a good programming practice. Values that are hard-coded in a program are subject to change and would therefore require that the program be recompiled each time a value changed. Externalizing these values from programs, as with the case of referring to data sets within a program by a symbolic name, is a more effective practice that allows the program to continue working even if the data set name changes.

```
//MYJOB    JOB
//STEP1    EXEC CLG
...
   OPEN FILE=INPUT1
   OPEN FILE=OUTPUT1
   READ FILE=INPUT1
...
   WRITE FILE=OUTPUT1
...
   CLOSE FILE=INPUT1
   CLOSE FILE=OUTPUT1
/*
//GO.INPUT1  DD DSN=MY.INPUT,DISP=SHR
//GO.OUTPUT1 DD DSN=MY.OUTPUT,DISP=OLD
```

Figure 14. HLL Relationship between JCL and program files

PL/I on z/OS

Programming Language/I (PL/I, pronounced “P-L one”), is a full-function, general-purpose, high-level programming language.

PL/I is suitable for the development of:

- Commercial applications
- Engineering/scientific applications
- Many other applications

The process of compiling a PL/I source program and then link-editing the object deck into a load module is basically the same as it is for COBOL.

The relationship between JCL and program files is the same for PL/I as it is for COBOL and other HLLs.

PL/I program structure

PL/I is a block-structured language, consisting of packages, procedures, statements, expressions, and built-in functions, as shown in Figure 15 on page 31.

PL/I programs are made up of blocks. A **block** can be either a subroutine, or just a group of statements. A PL/I block allows you to produce highly modular applications, because blocks can contain declarations that define variable names and storage classes. Thus, you can restrict the scope of a variable to a single block or a group of blocks, or you can make it known throughout the compilation unit or a load module.

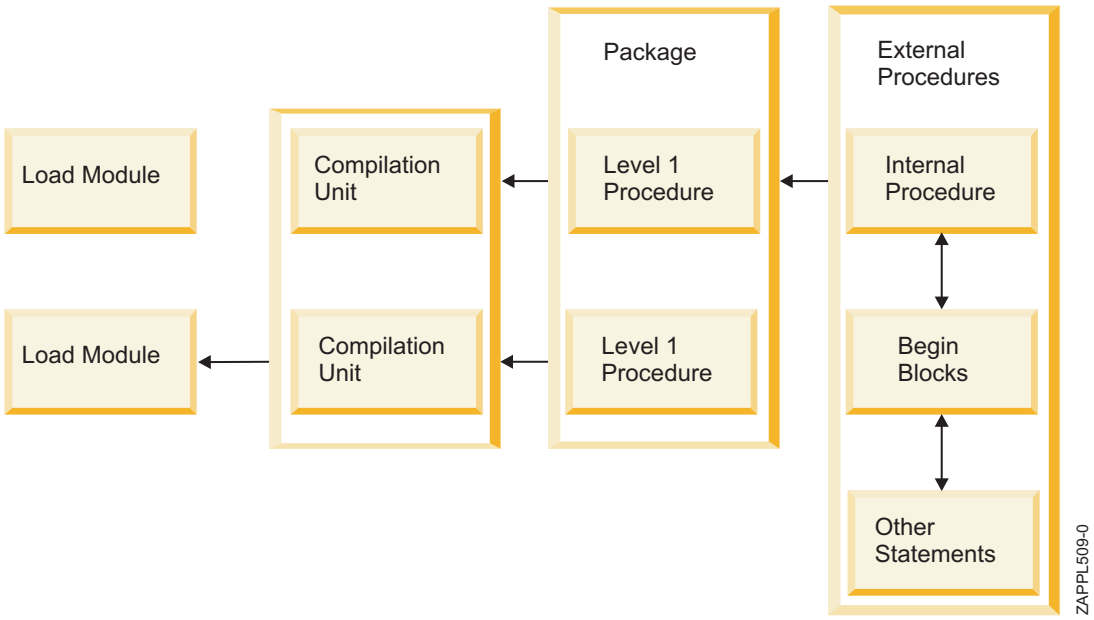

Figure 15. PL/I application structure

A PL/I application consists of one or more separately loadable entities, known as a **load modules**. Each load module can consist of one or more separately compiled entities, known as **compilation units**. Unless otherwise stated, a program refers to a PL/I application or a compilation unit.

A compilation unit is a PL/I package or an external procedure. Each package can contain zero or more procedures, some or all of which can be exported. A PL/I external or internal procedure contains zero or more blocks.

A PL/I block is either a PROCEDURE or a begin block, any of which contains zero or more statements and/or zero or more blocks. A procedure is a sequence of statements delimited by a procedure statement and a corresponding end statement, as shown in Figure 16. A procedure can be a main procedure, a subroutine, or a function.

```
A:  procedure;
            statement-1
            statement-2

     .

     .

     .

          statement-n
          end Name;
```

Figure 16. A PROCEDURE block

A begin **block** is a sequence of statements delimited by a begin statement and a corresponding end statement, as shown in Figure 17 on page 32. A program is

terminated when the main procedure is terminated.

```
B:  begin;
     statement-1
     statement-2

     .

     .

     statement-n
     end B;
```

Figure 17. BEGIN block

Preprocessors

The PL/I compiler allows you to select one or more of the integrated preprocessors as required for use in your program. You can select the include preprocessor, the macro preprocessor, the SQL preprocessor, or the CICS preprocessor—and you can select the order in which you would like them to be called.

Each preprocessor supports a number of options to allow you to tailor the processing to your needs.

- Include preprocessor

  This allows you to incorporate external source files into your programs by using include directives other than the PL/I directive %INCLUDE (the %INCLUDE directive is used to incorporate external text into the source program).
- Macro preprocessor

  Macros allow you to write commonly used PL/I code in a way that hides implementation details and the data that is manipulated, and exposes only the operations. In contrast to a generalized subroutine, macros allow generation of only the code that is needed for each individual use.
- DB2 precompiler or DB2 coprocessor

  In general, the structure of your Pl/I program is the same whether or not you want it to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data, you must:
  – Connect to a data server.
  – Execute SQL statements. SQL statements must begin with EXEC SQL.
  – Disconnect from the data server.

  To prepare the program for execution, you follow one of these procedures:
  – Preprocess, precompile, compile, link, and bind the program the program.
  – Preprocess, then compile the program with the SQL option, and then link and bind the program.

  Before you can take advantage of EXEC SQL support, you must have authority to access a DB2 system.

  Note that the PL/I SQL Preprocessor currently does not support DBCS.
- CICS preprocessor

  You can use EXEC CICS statements in PL/I applications that run as transactions under CICS.

Related reading: For more information about using PL/1 on z/OS, see the IBM publications *Enterprise PL/I for z/OS V3R3 Language Reference*, SC27-1460, and *Enterprise PL/I for z/OS V3R3 Programming Guide*, SC27-1457.

The SAX parser

The PL/I compiler provides an interface called PLISAXx (x = A or B) that provides you with basic XML capability. The support includes a high-speed XML parser, which allows programs to accept inbound XML messages, check them for being well-formed, and transform their contents to PL/I data structures.

The XML support does not provide XML generation, which must instead be accomplished by PL/I program logic. The XML support has no special environmental requirements. It executes in all the principal runtime environments, including CICS, IMS, and MQ Series, as well as z/OS batch and TSO.

More information about the PL/I language

The PL/I compiler is designed according to the specifications of industry standards as understood and interpreted by IBM as of December 1987.

- American National Standard Code for Information Interchange (ASCII), X3.4 - 1977
- American National Standard Representation of Pocket Select Characters in Information Interchange, level 1, X3.77 - 1980 (proposed to ISO, March 1, 1979)
- The draft proposed American National Standard Representation of Vertical Carriage Positioning Characters in Information Interchange, level 1, dpANS X3.78 (also proposed to ISO, March 1, 1979)
- Selected features of the American National Standard PL/I General Purpose Subset (ANSI X3.74-1987).

Input and output

PL/I input and output statements (such as READ, WRITE, GET, PUT) let you transmit data between the main storage and auxiliary storage of a computer.

A collection of data external to a program is called a **data set**. Transmission of data from a data set to a program is called **input**. Transmission of data from a program to a data set is called **output**. (If you are using a terminal, “data set” can also mean your terminal.)

PL/I input and output statements are concerned with the **logical** organization of a data set and not with its physical characteristics. A program can be designed without specific knowledge of the input/output devices that are used when the program is executed. To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs models of data sets, called **files**. A file can be associated with different data sets at different times during the execution of a program.

Data transmission

PL/I uses two types of data transmission: stream and record.

- In stream-oriented data transmission, the organization of the data in the data set is ignored within the program, and the data is treated as though it were a continuous stream of individual data values in character form. Data is converted from character form to internal form on input, and from internal form to character form on output.
- In record-oriented data transmission, the data set is a collection of discrete records. The record on the external medium is generally an exact copy of the record as it exists in internal storage. No data conversion takes place during record-oriented data transmission. On input, the data is transmitted exactly as it is recorded in the data set, and on output, it is transmitted exactly as it is recorded internally.

Stream-oriented data transmission is more versatile than record-oriented data transmission in its data-formatting abilities, but is less efficient in terms of run time.

Record-oriented data transmission is more versatile than stream-oriented data transmission, in both the manner in which data can be processed and the types of data sets that it can process. Since data is recorded in a data set exactly as it appears in main storage, any data type is acceptable. No conversions occur, but you must have a greater awareness of the data structure.

Data sets

In addition to being used as input from and output to your terminal, data sets are stored on a variety of auxiliary storage media, including magnetic tape and direct access storage devices (DASDs). Despite their variety, these media have characteristics that allow common methods of collecting, storing, and transmitting data. The organization of a data set determines how data is recorded in a data set and how the data is subsequently retrieved so that it can be transmitted to the program. Records are stored in and retrieved from a data set either sequentially, on the basis of successive physical or logical positions, or directly, by the use of keys specified in data transmission statements.

PL/I supports the following types of data set organizations:

- Consecutive. In the consecutive data set organization, records are organized solely on the basis of their successive physical positions. When the data set is created, records are written consecutively in the order in which they are presented. The records can be retrieved only in the order in which they were written.
- Indexed. In the indexed data set organization, records are placed in a logical sequence based on the key of each record. An indexed data set must reside on a direct-access device. A character string key identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.
- Relative. In the relative data set organization, numbered records are placed in a position relative to each other. The records are numbered in succession, beginning with one. A relative data set must reside on a direct-access device. A key that specifies the record number identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.
- Regional. The regional data set organization is divided into numbered regions, each of which can contain one record. The regions are numbered in succession, beginning with zero. A region can be accessed by specifying its region number, and perhaps a key, in a data transmission statement. The key specifies the region number and identifies the region to allow optimized direct retrieval, replacement, addition, and deletion of records.

The data set organizations differ in the way they store data and in the means they use to access data.

Data declarations

When a PL/I program is executed, it can manipulate many different data items of particular data types. Each data item, except an unnamed arithmetic or string constant, is referred to in the program by a name. Each data name is given attributes and a meaning by a declaration (explicit or implicit).

A **data item** is either the value of a variable or a constant. (Note that these terms, as used here, are not exactly the same as in general mathematical usage.) Data items can be single items, called **scalars**, or they can be a collection of items called **data aggregates** that are groups of data items that can be referred to either collectively or individually. The types of data aggregates are arrays, structures, and unions.

A **variable** has a value or values that might change during execution of a program. A variable is introduced by a declaration, which declares the name and certain attributes of the variable. A name is explicitly declared if it appears in a DECLARE statement that explicitly declares attributes of names.

A **constant** has a value that cannot change. Constants for computational data are referred to by stating the value of the constant or naming the constant in a DECLARE statement.

PL/I interfaces to other products

PL/I can interface with sort programs, C, the Checkpoint/Restart facility, and user exits.

The Sort program

The compiler provides an interface called PLISRTx (x = A, B, C, or D) that allows you to make use of the IBM-supplied Sort programs.

When used from PL/I, the Sort program sorts records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a user-written PL/I procedure that the Sort program will call each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Inter-Language Communication with C

Inter-Language Communication (ILC) between PL/I and C allows you to use many of the data types common to both languages and should enable you to write PL/I code that either calls or is called by C.

The Checkpoint/Restart facility

The PL/I Checkpoint/Restart feature provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system.

User exits

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The PL/I products supply default exits and the associated source files.

With PL/I, you can write your own user exit or use the exit provided with the product, either “as is” or modified, depending on what you want to do with it.

C/C++ on z/OS

The C language contains a concise set of statements with functionality added through its library. This division enables C to be both flexible and efficient. An additional benefit is that the language is highly consistent across different systems.

C is a programming language designed for a wide variety of programming purposes, including:

- System-level code
- Text processing
- Graphics.

The process of compiling a C source program and then link-editing the object deck into a load module is basically the same as it is for COBOL. The relationship between JCL and program files is the same for C/C++ as it is for COBOL and other HLLs.

More information about the C++ language

C supports numerous data types, including characters, integers, floating-point numbers and pointers—each in a variety of forms. In addition, C supports arrays, structures (records), unions, and enumerations. The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as well as date and time manipulation. Use of this library helps to maintain program portability, because the underlying implementation details for the various operations need not concern the programmer.

In the following paragraphs we refer to the C and C++ languages collectively as C.

Source programs

A C source program is a collection of one or more directives, declarations, and statements contained in one or more source files.

- **Directives** instruct the C preprocessor to act on the text of the program.
- **Declarations** establish names and define characteristics such as scope, data type and linkage.
- **Statements** specify the action to be performed.
- **Definitions** are declarations that allocate storage for data objects or define a body for functions. An object definition allocates storage and may optionally initialize the object.

A function definition precedes the function body. The function body is a compound statement that can contain declarations and statements that define what the function does. The function definition declares the function name, its parameters, and the data type of the value it returns.

The order and scope of declarations affect how you can use variables and functions in statements. In particular, an identifier can be used only **after** it is declared.

A program must contain at least one function definition. If the program contains only one function definition, the function must be called **main**. If the program contains more than one function definition, only **one** of the functions can be called main. The main function is the first function called when a program is run.

C source files

A C source file is the collection of text files that contains all of a C source program. It can include any of the functions that the program needs. To create an executable object module, you compile the source files individually and then link them as one program. With the `#include` directive, you can combine source files into larger source files.

A source file contains any combination of directives, declarations, and definitions. You can split items such as function definitions and large data structures between text files, but you cannot split them between compiled files. Before the source file is compiled, the preprocessor filters out preprocessor directives that may change the files. As a result of the preprocessing stage, preprocessor directives are completed, macros are expanded, and a source file is created containing C statements, completed directives, declarations, and definitions.

It is sometimes useful to place variable definitions in one source file and declare references to those variables in any source files that use them. This procedure makes definitions easy to find and change, if necessary. You can also organize constants and macros into separate files and include them into source files as required.

Directives in a source file apply to that source file and its included files only. Each directive applies only to the part of the file following the directive.

Data types

The C data types are:

- Characters
- Floating-Point Numbers
- Integers
- Enumerations
- Structures
- Unions

From these types, you can derive the following:

- Arrays
- Pointers
- Functions

The void type

The void data type always represents an empty set of values. The keyword for this type is void. When a function does not return a value, you should use void as the type specifier in the function definition and declaration. The only object that can be declared with the type specifier void is a pointer.

Java on z/OS

Java is an object-oriented programming language developed by Sun Microsystems, Inc. Java can be used for developing traditional mainframe commercial applications as well as Internet and intranet applications that use standard interfaces.

Java is an increasingly popular programming language used for many applications across multiple operating systems. IBM is a major supporter and user of Java across all of the IBM computing platforms, including z/OS. The z/OS Java products provide the same, full function Java APIs as on all other IBM platforms. In addition, the z/OS Java licensed programs have been enhanced to allow Java access to z/OS unique file systems. Programming languages such as Enterprise COBOL and Enterprise PL/I in z/OS provide interfaces to programs written in Java. These languages provide a set of interfaces or facilities for interacting with programs written in Java.

The various Java Software Development Kit (SDK) licensed programs for z/OS help application developers use the Java APIs for z/OS, write or run applications across multiple platforms, or use Java to access data that resides on the mainframe. Some of these products allow Java applications to run in only a 31-bit addressing environment. However, with 64-bit SDKs for z/OS, pure Java applications that were previously storage-constrained by 31-bit addressing can execute in a 64-bit environment. Also, some mainframes support a special processor for running Java applications called the zSeries Application Assist Processor (zAAP). Programs can be run interactively through z/OS UNIX or in batch.

IBM SDK products for z/OS

As with Java SDKs for other IBM platforms, z/OS Java SDK licensed programs are supplied for industry standard APIs. The z/OS SDK products are independent of each other and can be ordered and serviced separately.

At the time of writing, the following Java SDKs are available for z/OS:

- The Java SDK 1.3.1 product called IBM Developer Kit for OS/390, Java 2 Technology Edition works on z/OS as well as the older OS/390. This is a 31-bit product. Many z/OS customers have moved (or migrated) their Java applications to the latest versions of Java.
- IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4 is IBM's 31-bit port of the Sun Microsystems Java Software Development Kit SDK) to the z/OS platform and is certified as a fully compliant Java product. IBM has successfully executed the Java Certification Kit (JCK) 1.4 provided by Sun Microsystems, Inc.
- IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4 runs on z/OS Version 1 Release 4 or later, or z/OS.e Version 1 Release 4 or later. It provides a Java execution environment equivalent to that available on any other server platform.
- IBM 64-bit SDK for z/OS, Java 2 Technology Edition, Version 1.4 allows Java applications to execute in a 64-bit environment. It runs on z/OS Version 1 Release 6 or later. As with the 31-bit product, this product allows usage of the Java SDK1.4 APIs.

IBM provides more information about its Java SDK products for z/OS on the Web at:

http://www.ibm.com/servers/eserver/zseries/software/java/

The Java Native Interface (JNI)

The Java Native Interface (JNI) is the Java interface to native programming languages and is part of the Java Development Kits. If the standard Java APIs do not have the function you need, the JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other languages, such as PL/I. In addition, the Invocation API allows you to embed a Java Virtual Machine into your native PL/I applications.

Java is a fairly complete programming language; however, there are situations in which you want to call a program written in another programming language. You would do this from Java with a method call to a native language, known as a **native method**. Programming through the JNI lets you use native methods to do many different operations. A native method can:

- Use Java objects in the same way that a Java method uses these objects.
- Create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks.
- Inspect and use objects created by Java application code.
- Update Java objects that it created or were passed to it; these updated objects can then be made available to the Java application.

Lastly, native methods can also easily call already-existing Java methods, capitalizing on the functionality already incorporated in the Java programming framework. In this way, both the native language side and the Java side of an application can create, update, and access Java objects, and then share these objects between them.

The CLIST language on z/OS

The CLIST and REXX languages are the two command languages available from TSO/E. The CLIST language enables you to work more efficiently with TSO/E.

The CLIST language is an interpreted language. Like programs in other high-level interpreted languages, CLISTs are easy to write and test. You do not compile or link-edit them. To test a CLIST, you simply run it and correct any errors that might occur until the program runs without error.

The term CLIST (pronounced “see list”) stands for **command list**; it is called this because the most basic CLISTs are lists of TSO/E commands. When you invoke such a CLIST, it issues the TSO/E commands in sequence.

The CLIST programming language is used for:

- Performing routine tasks (such as entering TSO/E commands)
- Invoking other CLISTs
- Invoking applications written in other languages
- ISPF applications (such as displaying panels and controlling application flow).

Types of CLISTs

A CLIST can perform a wide range of tasks, but most fall into one of three general categories.

- CLISTs that perform routine tasks
- CLISTs that are structured applications
- CLISTs that manage applications written in other languages.

These are described in this section.

CLISTs that perform routine tasks

As a user of TSO/E, you will probably perform certain tasks on a regular basis. These tasks might involve entering TSO/E commands to check on the status of data sets, to allocate data sets for particular programs, or to print files.

You can write CLISTs that significantly reduce the amount of time that you have to spend on these routine tasks. By grouping all the instructions required to perform a task in a CLIST, you reduce the time, number of keystrokes, and errors involved in performing the task and increase your productivity. A CLIST can consist of TSO/E commands only or a combination of TSO/E commands and CLIST statements.

CLISTs that are structured applications

The CLIST language includes the basic tools you need to write complete, structured applications. Any CLIST can invoke another CLIST, which is referred to as a **nested** CLIST. CLISTs can also contain separate routines called **sub-procedures**. Nested CLISTs and sub-procedures let you separate your CLISTs into logical units and put common functions in a single location. Specific CLIST statements let you:

- Define common data for sub-procedures and nested CLISTs
- Restrict data to certain sub-procedures and CLISTs
- Pass specific data to a sub-procedure or nested CLIST.

For interactive applications, CLISTs can issue ISPF commands to display full-screen panels. Conversely, ISPF panels can invoke CLISTs, based on input that a user types on the panel.

CLISTs that manage applications written in other languages

Suppose you have access to applications written in other programming languages, but the interfaces to these applications might not be easy to use or remember. Rather than write new applications, you can write CLISTs that provide easy-to-use interfaces between the user and such applications.

A CLIST can send messages to, and receive messages from, the terminal to determine what the user wants to do. Then, based on this information, the CLIST can set up the environment and issue the commands required to invoke the program that performs the requested tasks.

Executing CLISTs

To execute a CLIST, use the EXEC command.

From an ISPF command line, type **TSO** at the beginning of the command. In TSO/E EDIT or TEST mode, use the EXEC subcommand as you would use the EXEC command. (CLISTs executed under EDIT or TEST can issue only EDIT or TEST subcommands and CLIST statements, but you can use the END subcommand in a CLIST to end EDIT or TEST mode and allow the CLIST to issue TSO/E commands.)

Other uses for the CLIST language

Besides issuing TSO/E commands, CLISTs can perform more complex programming tasks. The CLIST language includes the programming tools you need to write extensive, structured applications. CLISTs can perform any number of complex tasks, from displaying a series of full-screen panels to managing programs written in other languages.

CLIST language features include:

- An extensive set of arithmetic and logical operators for processing numeric data

- String-handling functions for processing character data
- CLIST statements that let you structure your programs, perform I/O, define and modify variables, and handle errors and attention interrupts.

REXX on z/OS

The Restructured Extended Executor (REXX) language is a procedural language that allows programs and algorithms to be written in a clear and structural way. It is an interpreted and compiled language. An interpreted language is different from other programming languages, such as COBOL, because it is not necessary to compile a REXX command list before executing it. However, you can choose to compile a REXX command list before executing it to reduce processing time.

The REXX programming language is typically used for:

- Performing routine tasks, such as entering TSO/E commands
- Invoking other REXX execs
- Invoking applications written in other languages
- ISPF applications (displaying panels and controlling application flow)
- One-time quick solutions to problems
- System programming
- Wherever we can use another HLL compiled language.

REXX is also used in the Java environment; for example, a dialect of REXX called NetRexx works seamlessly with Java. NetRexx programs can use any Java classes directly, and can be used for writing any Java class. This brings Java security and performance to REXX programs, and REXX arithmetic and simplicity to Java. Thus, a single language, NetRexx, can be used for both scripting and application development.

The structure of a REXX program is simple. It provides a conventional selection of control constructs. For example, these include `IF... THEN... ELSE...` for simple conditional processing, `SELECT... WHEN... OTHERWISE... END` for selecting from a number of alternatives, and several varieties of `DO... END` for grouping and repetitions. No `GOTO` instruction is included, but a `SIGNAL` instruction is provided for abnormal transfer of control such as error exits and computed branching.

The relationship between JCL and program files is the same for REXX as it is for COBOL and other HLLs.

Compiling and executing REXX command lists

The process of compiling a REXX source program and then link-editing the object deck into a load module is basically the same as it is for COBOL.

A REXX program compiled under z/OS can run under z/VM®. Similarly, a REXX program compiled under z/VM can run under z/OS. A REXX program compiled under z/OS or z/VM can run under z/VSE™ if REXX/VSE is installed.

There are three main components of the REXX language when using a compiler:

- IBM Compiler for REXX on zSeries. The Compiler translates REXX source programs into compiled programs.
- IBM Library for REXX on zSeries. The Library contains routines that are called by compiled programs at run-time.

- Alternate Library. The Alternate Library contains a language processor that transforms the compiled programs and runs them with the interpreter. It can be used by z/OS and z/VM users who do not have the IBM Library for REXX on zSeries to run compiled programs.

The Compiler and Library run on z/OS systems with TSO/E, and under CMS on z/VM systems. The IBM Library for REXX in REXX/VSE runs under z/VSE.

The Compiler can produce output in the following forms:

- Compiled EXECs. These EXECs behave exactly like interpreted REXX programs. They are invoked the same way by the system's EXEC handler, and the search sequence is the same. The easiest way of replacing interpreted programs with compiled programs is by producing compiled EXECs. Users need not know whether the REXX programs they use are compiled EXECs or interpretable programs. Compiled EXECs can be sent to z/VSE to be run there.
- Object decks under z/OS or TEXT files under z/VM.
- A TEXT file is an object code file whose external references have not been resolved (this term is used on z/VM only). These must be transformed into executable form (load modules) before they can be used. Load modules and MODULE files are invoked the same way as load modules derived from other compilers, and the same search sequence applies. However, the search sequence is different from that of interpreted REXX programs and compiled EXECs. These load modules can be used as commands and as parts of REXX function packages. Object decks or MODULE files can be sent to z/VSE to build phases.
- IEXEC output. This output contains the expanded source of the REXX program being compiled. Expanded means that the main program and all the parts included at compilation time by means of the `%INCLUDE` directive are contained in the IEXEC output. Only the text within the specified margins is contained in the IEXEC output. Note, however, that the default setting of **MARGINS** includes the entire text in the input records.

Related reading: You can find more information about REXX in the following publications:

- *The REXX Language*, 2nd Ed., Cowlishaw, ZB35-5100
- *SAA CPI Procedures Language Reference (Level 1)*, SC26-4358
- *REXX on zSeries V1R4.0 User's Guide and Reference*, SH19-8160
- *Creating Java Applications Using NetRexx*, SG24-2216

Also, visit the following Web site:

http://www-306.ibm.com/software/awdtools/rexx/

More information about the REXX language

The System Product Interpreter (SPI), a component of the z/VM operating system, processes procedures, XEDIT macros, and programs written in the REXX language. The REXX interpreter operates directly on the program as it executes, line-by-line and word-by-word.

There is no mechanism for declaring variables. Variables may be documented and initialized at the start of a program and implicit declarations occur during execution. The only true declarations are the markers (labels) that identify points in the program that may be used as the targets of SIGNAL instructions or internal routine calls.

Input and output functions in REXX are defined only for simple character-based operations.

Instructions written in any high level language, such as REXX, must be prepared for execution. The two types of programs that can perform this task are:

- An interpreter, which parses and executes an instruction before it parses and executes the next instruction.
- A compiler, which translates all the instructions of a program into a machine code program. It can keep the machine code program for later execution. It does not execute the program.

The input to a compiler is the source program that you write. The output from a compiler is the compiled program and the listing. The process of translating a source program into a compiled program is known as **compilation**.

Compiled versus interpreted languages

During the design of an application, you might need to decide whether to use a compiled language or an interpreted language for the application source code.

Both types of languages have their strengths and weaknesses. Usually, the decision to use an interpreted language is based on time restrictions on development or for ease of future changes to the program. A trade-off is made when using an interpreted language. You trade speed of development for higher execution costs. Because each line of an interpreted program must be translated each time it is executed, there is a higher overhead. Thus, an interpreted language is generally more suited to ad hoc requests than predefined requests.

Advantages of compiled languages

Assembler, COBOL, PL/I, C/C++ are all translated by running the source code through a compiler. This results in very efficient code that can be executed any number of times. The overhead for the translation is incurred just once, when the source is compiled; thereafter, it need only be loaded and executed.

Interpreted languages, in contrast, must be parsed, interpreted, and executed each time the program is run, thereby greatly adding to the cost of running the program. For this reason, interpreted programs are usually less efficient than compiled programs.

Some programming languages, such as REXX and Java, can be either interpreted or compiled.

Advantages of interpreted languages

There are reasons for using languages that are compiled and reasons for using interpreted languages. There is no simple answer as to which language is "better"—it depends on the application. Even within an application we could end up using many different languages. For example, one of the strengths of a language like CLIST is that it is easy to code, test, and change. However, it is not very efficient. The trade-off is machine resources for programmer time.

Keeping this in mind, we can see that it would make sense to use a compiled language for the intensive parts of an application (heavy resource usage), whereas interfaces (invoking the application) and less-intensive parts could be written in an

interpreted language. An interpreted language might also be suited for ad hoc requests or even for prototyping an application.

One of the jobs of a designer is to weigh the strengths and weaknesses of each language and then decide which part of an application is best served by a particular language.

What is z/OS Language Environment?

On z/OS, the Language Environment product provides a common environment for all conforming high-level language (HLL) products. An HLL is a programming language above the level of assembler language and below that of program generators and query languages. z/OS Language Environment establishes a common language development and execution environment for application programmers on z/OS. Whereas functions were previously provided in individual language products, Language Environment eliminates the need to maintain separate language libraries.

An **application** is a collection of one or more programs cooperating to achieve particular objectives, such as inventory control or payroll. The goals of application development include modularizing and sharing code, and developing applications on a workstation-based front end.

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

With Language Environment, and its ability to call one language from another, z/OS application programmers can exploit the functions and features in each language.

How Language Environment is used

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. These services are available through a set of interfaces that are consistent across programming languages. The application program can either call these interfaces directly, 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.

Figure 18 on page 45 shows the components in the Language Environment, including:

- 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 or 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.

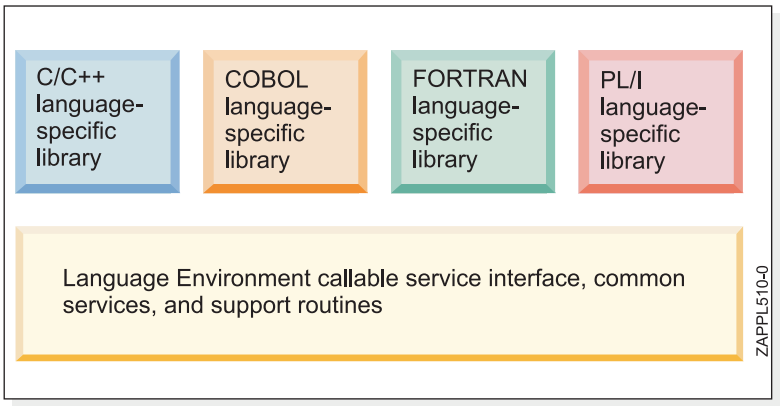

Figure 18. z/OS Language Environment components

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

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

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.

A closer look at Language Environment

The language-specific portions of Language Environment provide language interfaces and specific services that are supported for each individual language, and that can be called through a common callable interface.

In this section we discuss some of these interfaces and services in more detail.

Figure 19 on page 46 shows a common runtime environment established through Language Environment.

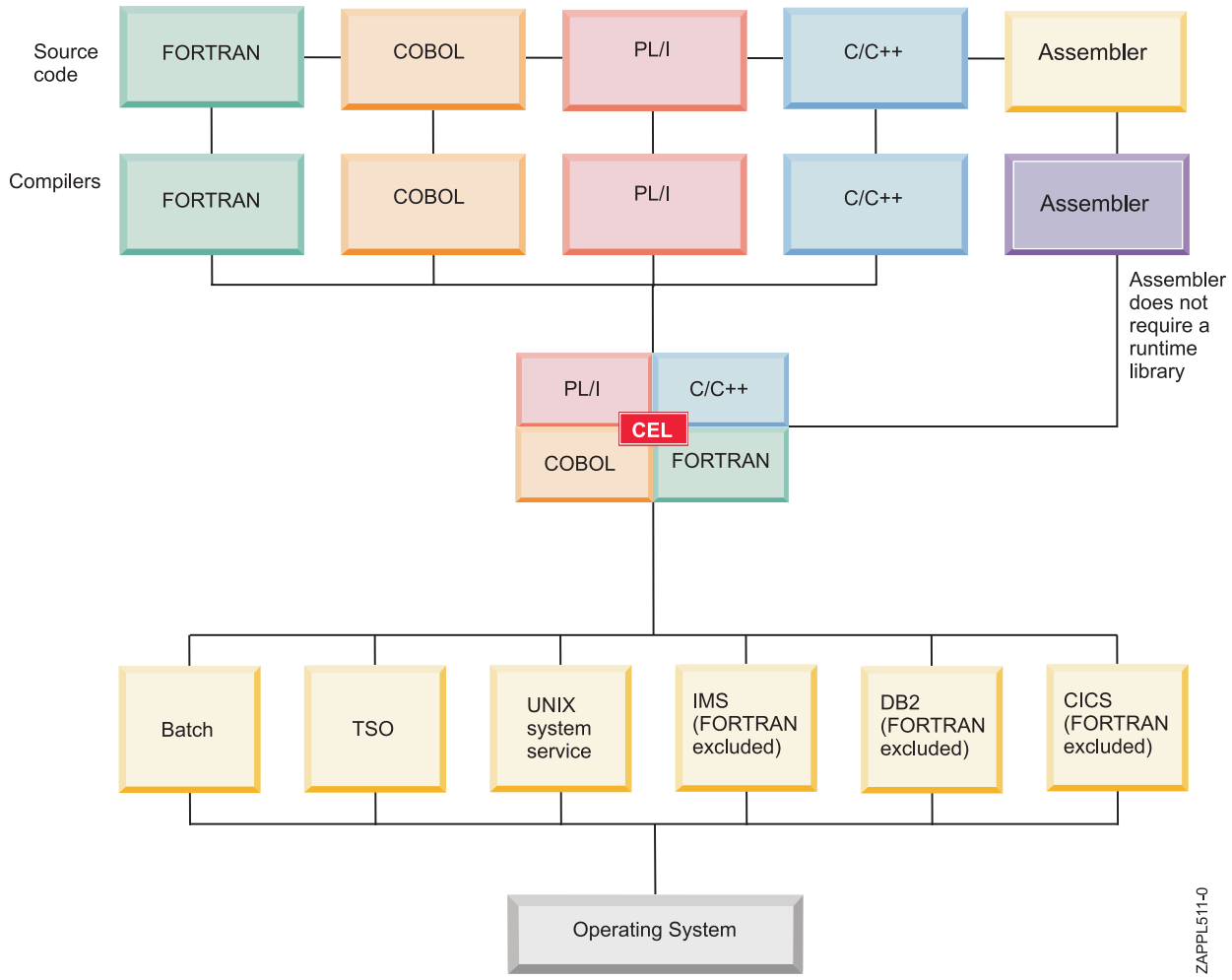

Figure 19. Language Environment's common runtime environment

The Language Environment architecture is built from models for the following:

- Program management
- Condition handling
- Message services
- Storage management.

Program management model in Language Environment

The Language Environment program management model provides a framework within which an application runs. It is the foundation for 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.

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.

Program management

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

Three entities, **process**, **enclave**, and **thread**, are at the core of the Language Environment program management model.

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

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 sub-functions, in C and C++
- A main procedure and all of its subroutines, in PL/I
- A program and its subroutines, in Fortran.

In Language Environment, environment is normally a reference to the runtime environment of HLLs at the enclave level. The enclave consists of one main routine and zero or more subroutines. The main routine is the first to execute in an enclave; all subsequent routines are named as subroutines.

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.

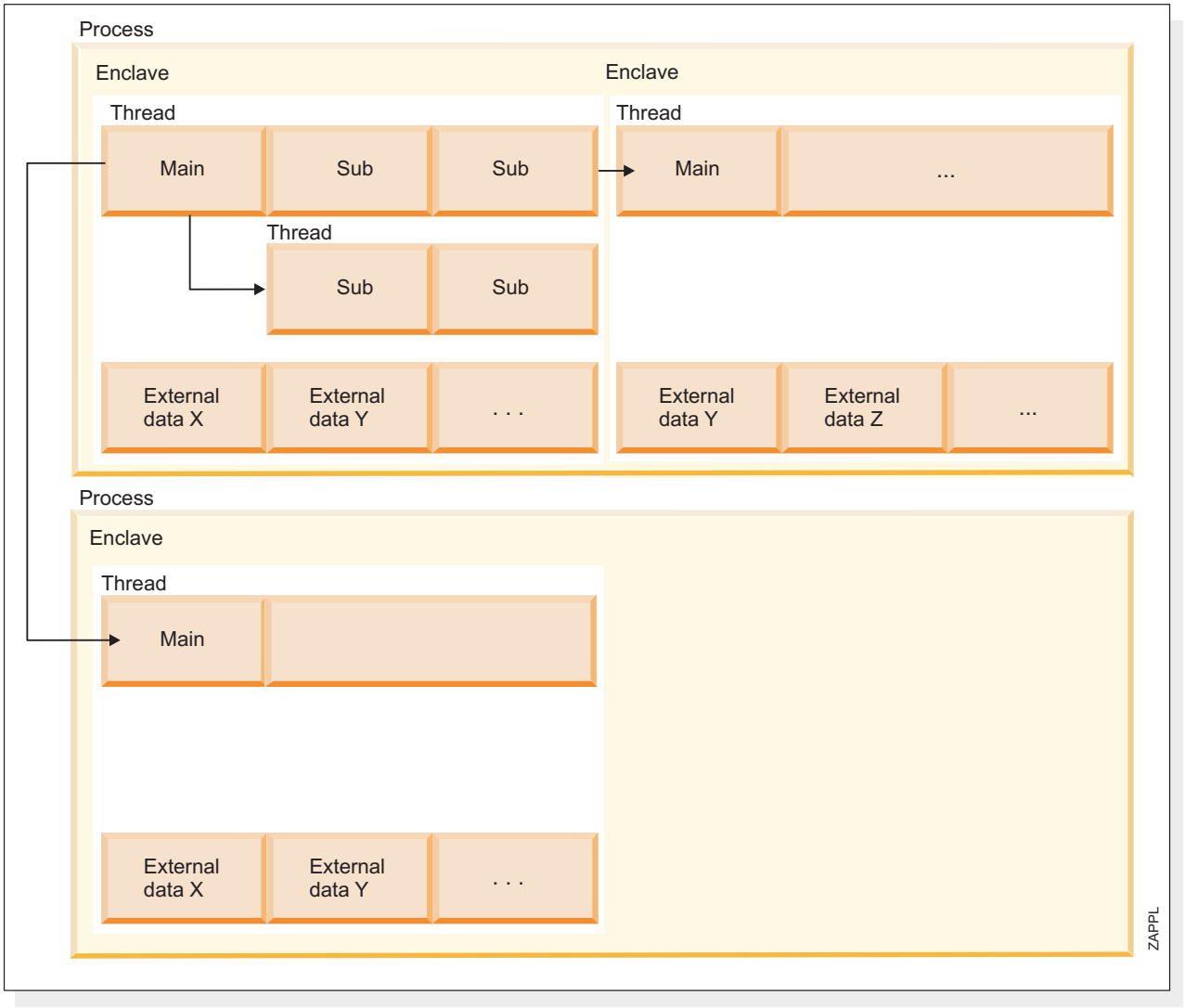

Figure 20. Full Language Environment program model

Figure 20 illustrates the full Language Environment program model, with its multiple processes, enclaves, and threads. As the figure shows, each process is within its own address space. An enclave consists of one main routine, with any number of subroutines.

The threads can create enclaves, which can create more threads, and so on.

Condition-handling model

For single-language 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.

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 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.

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-language 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.

How to run your program with Language Environment

Language Environment offers flexibility in how your program runs.

After compiling your program you can do the following:

- Link-edit and run an existing object deck and accept the default Language Environment runtime options
- Link-edit and run an existing object deck and specify new Language Environment runtime options
- Call a Language Environment service.

Accepting the default run-time options

To run an existing object deck under batch and accept all of the default Language Environment runtime options, you can use a Language Environment-provided link-edit and run cataloged procedure, CEEWLG, which identifies the Language Environment libraries that your object deck needs to link-edit and run.

Runtime library services

The Language Environment libraries are located in data sets identified with a high-level qualifier specific to the installation. For example, SCEERUN contains the runtime library routines needed during execution of applications written in C/C++, PL/I, COBOL, and FORTRAN. SCEERUN2 contains the runtime library routines needed during execution of applications written in C/C++ and COBOL.

Applications that require the runtime library provided by Language Environment can access the SCEERUN and SCEERUN2 data sets using one or both of these methods:

- LNKLST
- STEPLIB

Important: Language Environment library routines are divided into two categories: resident routines and dynamic routines. The resident routines are linked with the application and include such things as initialization and termination routines and pointers to callable services. The dynamic routines are not part of the application and are dynamically loaded during run time.

There are certain considerations that you must be aware of before link-editing and running applications under Language Environment.

Language Environment callable services

COBOL application developers will find Language Environment's consistent condition handling services especially useful. For all languages the same occurs with common math services, as well as the date and time services.

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 and termination services
- Locale callable services
- Math services
- Message handling services
- National language support services.

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. Following are examples of the invocation of a callable math service from three of the languages we have described in this section.

Figure 21 shows how a COBOL program invokes the math callable services CEESDLG1 for log base 10.

```
77   ARG1RL  COMP-2.
77   FBCODE  PIC X(12).
77   RESLTRL COMP-2.
     CALL "CEESDLG1" USING ARG1RL , FBCODE ,
     RESLTRL.
```

Figure 21. Sample invocation of a math callable service from a COBOL program

Related reading: The callable services are more fully described in the IBM publication, *z/OS Language Environment Programming Reference*, SA22-7562.

More information about the z/OS Language Environment

Examples show how program languages use callable services.

Invocation of callable services from PL/I

Figure 22 shows a PL/I program invoking the math callable services CEESIMOD and CEESSLOG for Modular arithmetic and log base e.

```
PLIMATH: PROC OPTIONS(MAIN);
         DCL CEESSLOG ENTRY OPTIONS(ASM) EXTERNAL;
         DCL CEESIMOD ENTRY OPTIONS(ASM) EXTERNAL;
         DCL ARG1 RESULT REAL FLOAT DEC (6);
         DCL ARGM1 ARGM2  RES2 FLOAT BINARY(21)
         DCL FC   CHARACTER (12);
        /* Call log base e routine, which has    */
        /* only one input parameter              */
         CALL CEESSLOG (ARG1, FC, RESULT)
         IF ( FC = '000000000000000000000000'X )
           THEN DO;
             PUT SKIP LIST
                ('Error occurred in call to CEESSLOG.' );
           ELSE;
        /* Call modular arithmetic routine,      */
        /* which has two input parameters        */
         CALL CEESIMOD (ARGM1, ARGM2, FC, RES2);
         IF ( FC = '000000000000000000000000'X )
           THEN DO;
             PUT SKIP LIST
                ('Error occurred in call to CEESIMOD.' );
           ELSE;
         END;
```

Figure 22. Sample invocation of a math callable service from a PL/I program

Invocation of callable services from C

Figure 23 on page 52 shows a C program invoking the math callable services CEESDGL1 and CEESIMOD for Log base 10 and modular arithmetic.

```
      #include <leawi.h>
#include <string.h>
#include <stdio.h>
int main (void) {
  _FLOAT8 f1,result;
  _INT4 int1, int2, intr;
  _FEEDBACK fc;
  #define SUCCESS "\0\0\0\0"
  f1 = 1000.0;
  CEESDLG1(&f1,&fc,&result);
  if (memcmp(&fc,SUCCESS,4) != 0) {
     printf("CEESDLG1 failed with message number %d\n",
             fc.tok_msgno);
     exit(2999);
  }
  printf("%f log base 10 is %f\n",f1,result);
  int1 = 39;
  int2 = 7;
  CEESIMOD(&int1,&int2,&fc,&intr);
  if (memcmp(&fc,SUCCESS,4) != 0) {
     printf("CEESIMOD failed with message number %d\n",
           fc.tok_msgno);
     exit(2999);
  }
  printf("%d modulo %d is %d\n",int1,int2,intr);
}
```

Figure 23. Sample invocation of a math callable service from a C program

Invocation of callable services from Assembler

Figure 24 shows the invocation of a callable service from an Assembler language program.

```
        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 24. Sample invocation of a callable service from an Assembler program

Chapter 3. How programs are prepared to run on z/OS

The process steps for translating a source program into an executable load module, and executing the load module, include compiling and link-editing, although there might be a third step to pre-process the source prior to compiling it. The pre-processing step would be required if your source program issues CICS command language calls or SQL calls. The output of the pre-processing step is then fed into the compile step.

The purpose of the compile step is to validate and translate source code into relocatable machine language, in the form of object code. Although the object code is machine language, it is not yet executable. It must be processed by a linkage editor, binder, or loader before it can be executed.

The linkage editor, binder, and loader take as input object code and other load modules, and then produce an executable load module and, in the case of the loader, execute it. This process resolves any unresolved references within the object code and ensures that everything that is required for this program to execute is included within the final load module. The load module is now ready for execution.

To execute a load module, it must be loaded into central storage. The binder or program manager service loads the module into storage and then transfers control to it to begin execution. Part of transferring control to the module is to supply it with the address of the start of the program in storage. Because the program's instructions and data are addressed using a base address and a displacement from the base, this starting address gives addressability to the instructions and data within the limits of the range of displacement[4].

Source, object, and load modules

A program can be divided into logical units that perform specific functions. A logical unit of code that performs a function or several related functions is a **module**. Separate functions should be programmed into separate modules, a process called **modular programming**. Each module can be written in the symbolic language that best suits the function to be performed.

Each module is assembled or compiled by one of the language translators. The input to a language translator is a source module; the output from a language translator is an object deck. Before an object deck can be executed, it must be processed by the binder (or the linkage editor). The output of the binder is a **load module**; see Figure 25 on page 54.

4. The maximum displacement for each base register is 4096 (4K). Any program bigger than 4K must have more than one base register in order to have addressability to the entire program.

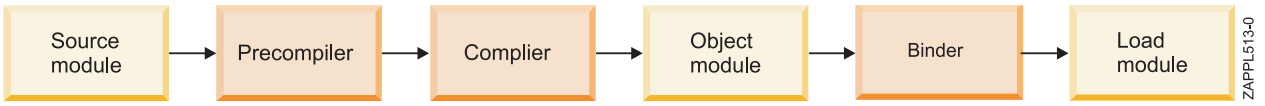

Figure 25. Source, object, and load modules

Depending on the status of the module, whatever it is–source, object or load–it can be stored in a library. A library is a partitioned data set (PDS) or a partitioned data set extended (PDSE) on direct access storage. PDSs and PDSEs are divided into partitions called **members**. In a library, each member contains a program or part of a program.

Source programs (or **source code**) is a set of statements written in a computer language. Source programs, once they are error-free, are stored in a partitioned data set known as a **source library**. Source libraries contain the source code to be submitted for a compilation process, or to be retrieved for modification by an application programmer.

A **copybook** is a source library containing pre-written text. It is used to copy text into a source program, at compile time, as a shortcut to avoid having to code the same set of statements over and over again. It is usually a shared library in which programmers store commonly used program segments to be later included into their source programs. It should not be confused with a subroutine or a program. A copybook member is just text; it might not be actual programming language statements.

A **subroutine** is a commonly-called routine that performs a predefined function. The purpose behind a copybook member and a subroutine are essentially the same, to avoid having to code something that has previously been done. However, a subroutine is a small program (compiled, link-edited and executable) that is called and returns a result, based on the information that it was passed. A copybook member is just text that will be included in a source program on its way to becoming an executable program. The term copybook is a COBOL term, but the same concept is used in most programming languages.

If you use copybooks in the program that you are compiling, you can retrieve them from the source library by supplying a DD statement for SYSLIB or other libraries that you specify in COPY statements. In Figure 26, we insert the text in member INPUTRCD from the library DEPT88.BOBS.COBLIB into the source program that is to be compiled.

```
//COBOL.SYSLIB  DD  DISP=SHR,DSN=DEPT88.BOBS.COBLIB
//SYSIN    DD  *
                IDENTIFICATION DIVISION.
      . . .
      COPY INPUTRCD
      . . .
```

Figure 26. Copybook in COBOL source code

Libraries must reside on direct access storage devices (DASDs). They cannot be in a hierarchical file system (HFS) when you compile using JCL or under TSO.

How programs are compiled on z/OS

The function of a compiler is to translate source code into an object deck, which must then be processed by a binder (or a linkage editor) before it is executed.

During the compilation of a source module, the compiler assigns relative addresses to all instructions, data elements, and labels, starting from zero. The addresses are in the form of a base address plus a displacement. This allows programs to be relocated, that is, they do not have to be loaded into the same location in storage each time that they are executed. Any references to external programs or subroutines are left as unresolved. These references will either be resolved when the object deck is linked, or dynamically resolved when the program is executed.

To compile programs on z/OS, you can use a batch job, or you can compile under TSO/E through commands, CLISTs, or ISPF panels. For C programs, you can compile in a z/OS UNIX shell with the c89 command. For COBOL programs, you can compile in a z/OS UNIX shell with the cob2 command.

For compiling through a batch job, z/OS includes a set of cataloged procedures that can help you avoid some of the JCL coding you would otherwise need to do. If none of the cataloged procedures meet your needs, you will need to write all of the JCL for the compilation.

As part of the compilation step, you need to define the data sets needed for the compilation and specify any compiler options necessary for your program and the desired output.

The data set (library) that contains your source code is specified on the SYSIN DD statement, as shown in Figure 27.

```
//SYSIN DD DSNAME=dsname,
// DISP=SHR
```

Figure 27. SYSIN DD statement for the source code

You can place your source code directly in the input stream. If you do so, use this SYSIN DD statement:

```
//SYSIN DD *
```

When you use the `DD *` convention, the source code must follow the statement. If another job step follows the compilation, the EXEC statement for that step follows the `/*` statement or the last source statement.

What is a precompiler?

Some compilers have a precompile or preprocessor to process statements that are not part of the computer programming language. If your source program contains EXEC CICS statements or EXEC SQL statements, then it must first be pre-processed to convert these statements into COBOL, PL/I or Assembler language statements, depending on the language in which your program is written.

Compiling with cataloged procedures

The simplest way to compile your program under z/OS is by using a batch job with a **cataloged procedure**. A cataloged procedure is a set of job control statements placed in a partitioned data set (PDS) called the procedure library (PROCLIB).

z/OS comes with a procedure library called SYS1.PROCLIB. A simple way to look at the use of cataloged procedures is to think of them as copybooks. Instead of source statements, however, cataloged procedures contain JCL statements. You do not need to code a JCL statement to tell the system where to find them because they are located in a system library which automatically gets searched when you execute JCL that references a procedure.

You need to include the following information in the JCL for compilation:

- Job description
- Execution statement to invoke the compiler
- Definitions for the data sets needed but not supplied by the procedure.

COBOL compile procedure

An example shows a single-step procedure for compiling a source program.

The JCL in Figure 28 executes the IGYWC procedure, which is a single-step procedure for compiling a source program. It produces an object deck that will be stored in the SYSLIN data set, as we can see in Figure 29 on page 57.

```
//COMP    JOB
//COMPILE EXEC IGYWC
//SYSIN    DD   *
    IDENTIFICATION DIVISION (source program)
.
.
.
/*
//
```

Figure 28. Basic JCL for compiling a COBOL source program inline

The SYSIN DD statement indicates the location of the source program. In this case, the asterisk (*) indicates that it is in the same input stream.

For PL/I programs, in addition to the replacement of the source program, the compile EXEC statement should be replaced by:

```
//compile EXEC IBMZC
```

The statements shown in Figure 29 on page 57 make up the IGYWC cataloged procedure used in Figure 28. As mentioned previously, the result of the compilation process, the compiled program, is placed in the data set identified on the SYSLIN DD statement.

```
//IGYWC  PROC LNGPRFX='IGY.V3R2M0',SYSLBLK=3200
//*
//*  COMPILE A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE
//*   SYSLBLK   3200
//*   LNGPRFX   IGY.V3R2M0
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
```

Figure 29. Procedure IGYWC - COBOL compile

COBOL pre-processor and compile and link procedure

An example shows a three-step procedure for pre-processing a COBOL source program, compiling the output from the pre-processing step, and then linking it into a load library.

The JCL in Figure 30 on page 58 executes the DFHEITVL procedure, which is a three-step procedure for pre-processing a COBOL source program, compiling the output from the pre-processing step, and then linking it into a load library. The first step produces pre-processed source code in the SYSPUNCH temporary data sets, with any CICS calls expanded into COBOL language statements. The second step takes this temporary data set as input and produces an object deck that is stored in the SYSLIN temporary data set, as shown in Figure 31 on page 58. The third step takes the SYSLIN temporary data set as input, as well as any other modules that might need to be included, and creates a load module in the data set referenced by the SYSLMOD DD statement.

In Figure 30 on page 58, you can see that the JCL is a bit more complicated than in the simple compile job (Figure 30 on page 58). Once we go from one step to multiple steps, we must tell the system which step we are referring to when we supply JCL overrides.

Looking at the JCL in Figure 31 on page 58, we see that the first step (each step is an EXEC statement, and the step name is the name on the same line as the EXEC statement) is named TRN, so we must qualify the SYSIN DD statement with TRN to ensure that it will be used in the TRN step.

Similarly, the fourth step is called LKED, so we must qualify the SYSIN DD statement with LKED in order for it to apply to the LKED step.

The end result of running the JCL in Figure 30 (assuming that there are no errors) should be to pre-process and compile our inline source program, link-edit the object deck, and then store the load module called PROG1 in the data set MY.LOADLIB.

```
//PPCOMLNK   JOB
//PPCL      EXEC DFHEITVL,PROGLIB='MY.LOADLIB'
//TRN.SYSIN    DD *
   IDENTIFICATION DIVISION (source program)
     EXEC CICS ...
...
     EXEC CICS ...
...
//LKED.SYSIN DD *
    NAME PROG1(R)
/*
```

Figure 30. Basic JCL for pre-processing, compiling, and linking a COBOL source program inline

The statements shown in Figure 31 make up the DFHEITVL cataloged procedure used in Figure 30. As with the other compile and link procedures, the result of the preprocessor, compile, and link steps, which is the load module, is placed in the data set identified on the SYSLMOD DD statement.

```
//DFHEITVL PROC SUFFIX=1$,           Suffix for translator module
//*
//* This procedure has been changed since CICS/ESA Version 3
//*
//* Parameter INDEX2 has been removed
//*
//       INDEX='CICSTS12.CICS', Qualifier(s) for CICS libraries
//       PROGLIB=&INDEX..SDFHLOAD,   Name of output library
//       DSCTLIB=&INDEX..SDFHCOB,    Name of private macro/DSECT lib
//       COMPHLQ='SYS1',             Qualifier(s) for COBOL compiler
//       OUTC=A,                     Class for print output
//       REG=2M,                     Region size for all steps
//       LNKPARM='LIST,XREF',        Link edit parameters
//       STUB='DFHEILIC',            Link edit INCLUDE for DFHECI
//       LIB='SDFHCOB',              Library
//       WORK=SYSDA                  Unit for work data sets
//*      This procedure contains 4 steps
//*      1.   Exec the COBOL translator
//*           (using the supplied suffix 1$)
//*      2.   Exec the vs COBOL II compiler
//*      3.   Reblock &LIB(&STUB) for use by the linkedit step
//*      4.   Linkedit the output into data set &PROGLIB
//*
```

Figure 31. Procedure DFHEITVL - COBOL preprocessor, compile, and link

```
//*      The following JCL should be used
//*      to execute this procedure
//*
//*      //APPLPROG EXEC DFHEITVL
//*      //TRN.SYSIN  DD *
//*         .
//*         . Application program
//*         .
//*      /*
//*      //LKED.SYSIN DD *
//*         NAME anyname(R)
//*      /*
//*
//* Where   anyname  is the name of your application program.
//*      (Refer to the system definition guide for full details,
//*      including what to do if your program contains calls to
//*      the common programming interface.)
//*
//TRN    EXEC PGM=DFHECP&SUFFIX,
//            PARM='COBOL2',
//            REGION=&REG
//STEPLIB  DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSPUNCH DD DSN=&&SYSCIN,
//            DISP=(,PASS),UNIT=&WORK,
//            DCB=BLKSIZE=400,
//            SPACE=(400,(400,100))
//*
//COB    EXEC PGM=IGYCRCTL,REGION=&REG,
//       PARM='NODYNAM,LIB,OBJECT,RENT,RES,APOST,MAP,XREF'
//STEPLIB  DD DSN=&COMPHLQ..COB2COMP,DISP=SHR
//SYSLIB   DD DSN=&DSCTLIB,DISP=SHR
//         DD DSN=&INDEX..SDFHCOB,DISP=SHR
//         DD DSN=&INDEX..SDFHMAC,DISP=SHR
//         DD DSN=&INDEX..SDFHSAMP,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSIN    DD DSN=&&SYSCIN,DISP=(OLD,DELETE)
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),
//            UNIT=&WORK,SPACE=(80,(250,100))
//SYSUT1   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT2   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT3   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT4   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT5   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT6   DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT7   DD UNIT=&WORK,SPACE=(460,(350,100))
//*
```

Figure 32. Procedure DFHEITVL - COBOL preprocessor, compile, and link (continued)

```
//COPYLINK EXEC PGM=IEBGENER,COND=(7,LT,COB)
//SYSUT1   DD DSN=&INDEX..&LIB(&STUB),DISP=SHR
//SYSUT2   DD DSN=&&COPYLINK,DISP=(NEW,PASS),
//            DCB=(LRECL=80,BLKSIZE=400,RECFM=FB),
//            UNIT=&WORK,SPACE=(400,(20,20))
//SYSPRINT DD SYSOUT=&OUTC
//SYSIN    DD DUMMY
//*
//LKED   EXEC PGM=IEWL,REGION=&REG,
//            PARM='&LNKPARM',COND=(5,LT,COB)
//SYSLIB   DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//         DD DSN=&COMPHLQ..COB2CICS,DISP=SHR
//         DD DSN=&COMPHLQ..COB2LIB,DISP=SHR
//SYSLMOD  DD DSN=&PROGLIB,DISP=SHR
//SYSUT1   DD UNIT=&WORK,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=&OUTC
//SYSLIN   DD DSN=&&COPYLINK,DISP=(OLD,DELETE)
//         DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
```

Figure 33. Procedure DFHEITVL - COBOL preprocessor, compile, and link (continued)

COBOL compile and link procedure

An example shows a two-step procedure for compiling a source program and linking it into a load library.

The JCL in Figure 34 executes the IGYWCL procedure, which is a two-step procedure for compiling a source program and linking it into a load library. The first step produces an object deck that is stored in the SYSLIN temporary data set, as shown in Figure 35 on page 61. The second step takes the SYSLIN temporary data set as input, as well as any other modules that might need to be included, and creates a load module in the data set referenced by the SYSLMOD DD statement.

The end result of running the JCL in Figure 34 (assuming that there are no errors) should be to compile our inline source program, link-edit the object deck, and then store the load module called PROG1 in the data set MY.LOADLIB.

```
//COMLNK   JOB
//CL       EXEC IGYWCL
//COBOL.SYSIN    DD *
   IDENTIFICATION DIVISION (source program)
.
.
.
/*
//LKED.SYSLMOD DD DSN=MY.LOADLIB(PROG1),DISP=OLD
```

Figure 34. Basic JCL for compiling and linking a COBOL source program inline

The statements shown in Figure 35 on page 61 make up the IGYWCL cataloged procedure used in Figure 34. As mentioned previously, the result of the compile and link steps, which is the load module, is placed in the data set identified on the SYSLMOD DD statement.

```
//IGYWCL PROC LNGPRFX='IGY.V2R1M0',SYSLBLK=3200,
//            LIBPRFX='CEE',
//            PGMLIB='&&GOSET',GOPGM=GO
//*
//*  COMPILE AND LINK EDIT A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE
//*   LNGPRFX   IGY.V2R1M0
//*   SYSLBLK   3200
//*   LIBPRFX   CEE
//*   PGMLIB    &&GOSET          DATA SET NAME FOR LOAD MODULE
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD ...
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD DSNAME=&LNGPRFX..SIGYCOMP,
//            DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,UNIT=VIO,
//            DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//            DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=VIO,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD DSNAME=&LIBPRFX..SCEELKED,
//            DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSNAME=&PGMLIB(&GOPGM),
//            SPACE=(TRK,(10,10,1)),
//            UNIT=VIO,DISP=(MOD,PASS)
//SYSUT1   DD UNIT=VIO,SPACE=(TRK,(10,10))
```

Figure 35. Procedure IGYWCL - COBOL compile and link

COBOL compile link and go procedure

An example shows a three-step procedure for compiling a source program, linking it into a load library, and then executing the load module.

The JCL in Figure 36 on page 62 executes the IGYWCLG procedure, which is a three-step procedure for compiling a source program, linking it into a load library, and then executing the load module. The first two steps are the same as those in the compile and link example (see related links). However, whereas in the other example we override the SYSLMOD DD statement in order to permanently save the load module, in Figure 36 on page 62, we do not need to save it in order to execute it. That is why the override to the SYSLMOD DD statement in Figure 36 on page 62 is enclosed in square brackets, to indicate that it is optional.

If it is coded, then the load module PROG1 will be permanently saved in MY.LOADLIB. If it is not coded, then the load module will be saved in a temporary data set and deleted after the GO step.

In Figure 36, you can see that the JCL is very similar to the JCL used in the simple compile job (see related links). Looking at the JCL in Figure 37 on page 63, the only difference is that we have added the GO step. The end result of running the JCL in Figure 36 (assuming that there are no errors) should be to compile our inline source program, link-edit the object deck, store the load module (either temporarily or permanently), and then execute the load module.

```
//CLGO    JOB
//CLG     EXEC IGYWCLG
//COBOL.SYSIN    DD *
  IDENTIFICATION DIVISION (source program)
.
.
.
/*
[//LKED.SYSLMOD DD DSN=MY.LOADLIB(PROG1),DISP=OLD]
```

Figure 36. Basic JCL for compiling, linking and executing a COBOL source program inline

The statements shown in Figure 37 on page 63 make up the IGYWCLG cataloged procedure used in Figure 36.

```
//IGYWCLG PROC LNGPRFX='IGY.V2R1M0',SYSLBLK=3200,
//            LIBPRFX='CEE',GOPGM=GO
//*
//*  COMPILE, LINK EDIT AND RUN A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IGY.V2R1M0
//*   SYSLBLK   3200
//*   LIBPRFX   CEE
//*   GOPGM     GO
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD ...
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD DSNAME=&LNGPRFX..SIGYCOMP,
//            DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,UNIT=VIO,
//            DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//            DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6   DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=VIO,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD DSNAME=&LIBPRFX..SCEELKED,
//            DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSLMOD  DD DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
//            UNIT=VIO,DISP=(MOD,PASS)
//SYSUT1   DD UNIT=VIO,SPACE=(TRK,(10,10))
//GO     EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
//            REGION=2048K
//STEPLIB  DD DSNAME=&LIBPRFX..SCEERUN,
//            DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP  DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
```

Figure 37. Procedure IGYWCLG - COBOL compile, link, and go

How object-oriented (OO) applications are compiled

An example uses the SYSJAVA ddname to write a generated Java source file to a file in the HFS.

If you use a batch job or TSO/E to compile an OO COBOL program or class definition, the generated object deck is written, as usual, to the data set that you identify with the SYSLIN or SYSPUNCH ddname.

If the COBOL program or class definition uses the JNI[5] environment structure to access JNI callable services, copy the file JNI.cpy from the HFS to a PDS or PDSE

5. The Java Native Interface (JNI) is the Java interface to native programming languages and is part of the Java Development Kits. By writing programs that use the JNI, you ensure that your code is portable across many platforms.

member called JNI, identify that library with a SYSLIB DD statement, and use a COPY statement of the form COPY JNI in the COBOL source program.

As shown in Figure 38, use the SYSJAVA ddname to write the generated Java source file to a file in the HFS. For example:

```
//SYSJAVA DD PATH='/u/userid/java/Classname.java',
//           PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//           PATHMODE=SIRWXU,
//           FILEDATA=TEXT
```

Figure 38. SYSJAVA ddname for a Java source file

What is an object deck?

An **object deck** is a collection of one or more compilation units produced by an assembler, compiler, or other language translator, and used as input to the binder (or linkage editor).

An object deck is in relocatable format with machine code that is not executable. A load module is also relocatable, but with executable machine code. A load module is in a format that can be loaded into virtual storage and relocated by program manager, a program that prepares load modules for execution by loading them at specific storage locations.

Object decks and load modules share the same logical structure consisting of:

- Control dictionaries, containing information to resolve symbolic cross-references between control sections of different modules, and to relocate address constants
- Text, containing the instructions and data of the program
- An end-of-module indication, which is an END statement in an object deck, or an end-of-module indicator in a load module.

Object decks are stored in a partitioned data set identified by the SYSLIN or SYSPUNCH DD statement, which is input to the next linkage edition process.

What is an object library?

You can use an object library to store object decks. The object decks to be link-edited are retrieved from the object library and transformed into an executable or loadable program.

When using the **OBJECT** compiler option, you can store the object deck on disk as a traditional data set, as an UNIX file, or on tape. The **DISP** parameter of the SYSLIN DD statement indicates whether the object deck is to be:

- Passed to the binder (or linkage editor) after compile (**DISP=PASS**)
- Cataloged in an existent object library (**DISP=OLD**)
- Kept (**DISP=KEEP**)
- Added to a new object library, which is cataloged at the end of the step (**DISP=CATLG**).

An object deck can be the primary input to the binder by specifying its data set name and member name on the SYSLIN DD statement. In the following example, the member named TAXCOMP in the object library USER.LIBROUT is the primary input. USER.LIBROUT is a cataloged partitioned data set:

```
//SYSLIN DD DSNAME=USER.LIBROUT(TAXCOMP),DISP=SHR
```

The library member is processed as if it were a sequential data set.

How does program management work?

Although program management components provide many services, they are used primarily to convert object decks into executable programs, store them in program libraries, and load them into virtual storage for execution.

You can use the program management binder and loader to perform these tasks. These components can also be used in conjunction with the linkage editor. A load module produced by the linkage editor can be accepted as input by the binder, or can be loaded into storage for execution by the program management loader. The linkage editor can also process load modules produced by the binder.

Figure 39 on page 66 shows how the program management components work together, and how each one is used to prepare an executable program. We have already discussed some of these components (source modules and object decks), so now we take a look at the rest of them.

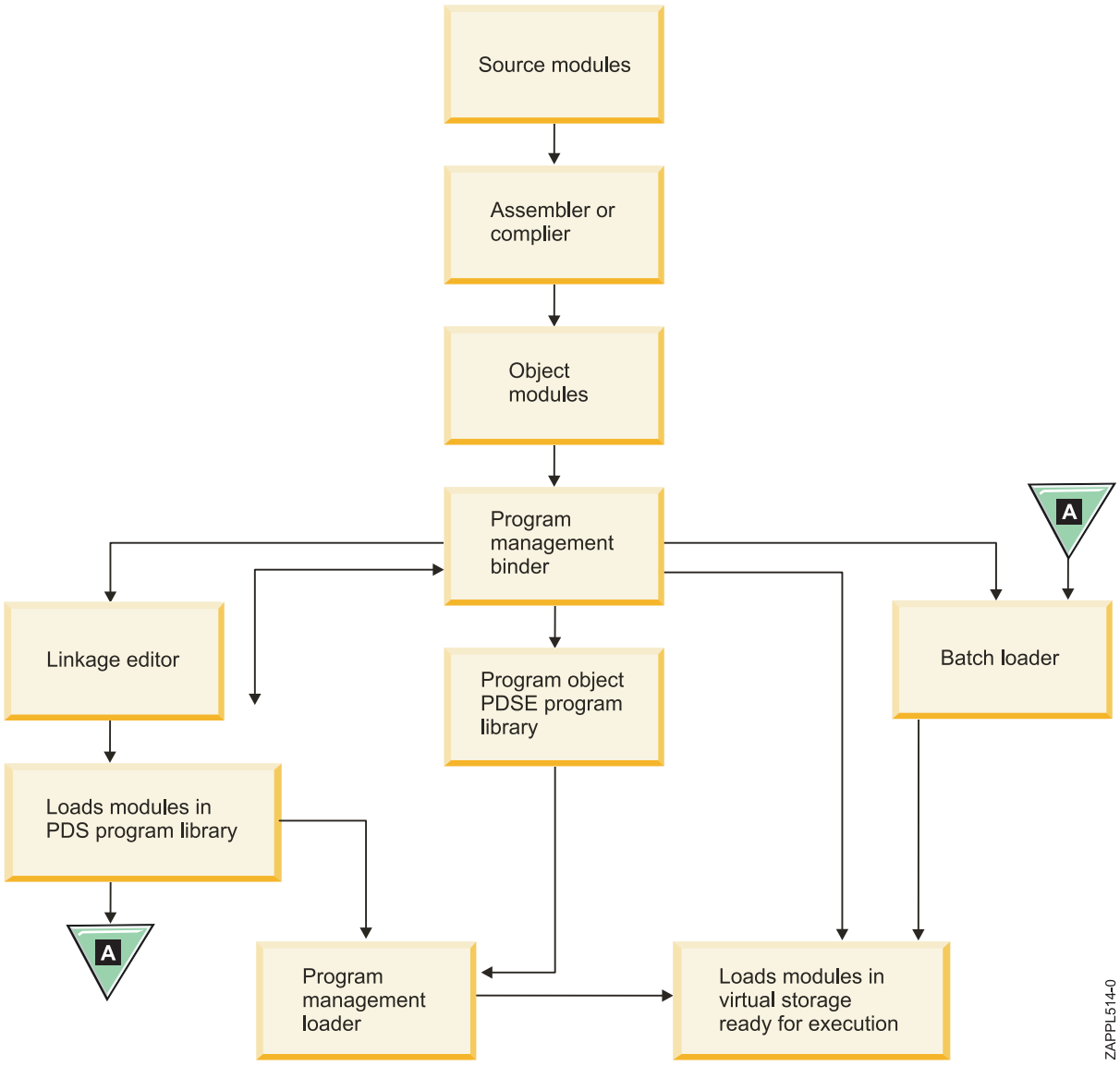

Figure 39. Using program management components to create and load programs

How is a linkage editor used?

Linkage editor processing follows the source program assembly or compilation of any problem program. The **linkage editor** is both a processing program and a service program used in association with the language translators.

Linkage editor and loader processing programs prepare the output of language translators for execution. The linkage editor prepares a load module that is to be brought into storage for execution by the program manager.

The linkage editor accepts two major types of input:

- Primary input, consisting of object decks and linkage editor control statements.

- Additional user-specified input, which can contain either object decks and control statements, or load modules. This input is either specified by you as input, or is incorporated automatically by the linkage editor from a call library.

Output of the linkage editor is of two types:

- A load module placed in a library (a partitioned data set) as a named member
- Diagnostic output produced as a sequential data set.

The loader prepares the executable program in storage and passes control to it directly.

How a load module is created

In processing object decks and load modules, the linkage editor assigns consecutive relative virtual storage addresses to control sections, and resolves references between control sections. Object decks produced by several different language translators can be used to form one load module.

An output load module is composed of all input object decks and input load modules processed by the linkage editor. The control dictionaries of an output module are, therefore, a composite of all the control dictionaries in the linkage editor input. The control dictionaries of a load module are called the composite external symbol dictionary (CESD) and the relocation dictionary (RLD). The load module also contains the text from each input module, and an end-of-module indicator.

Figure 40 shows the process of compiling two source programs: PROGA and PROGB. PROGA is a COBOL program and PROGB is an Assembler language program. PROGA calls PROGB. In this figure we see that after compilation, the reference to PROGB in PROGA is an unresolved reference. The process of link-editing the two object decks resolves the reference so that when PROGA is executed, the call to PROGB will work correctly. PROGB will be transferred to, it will execute, and control will return to PROGA, after the point where PROGB was called.

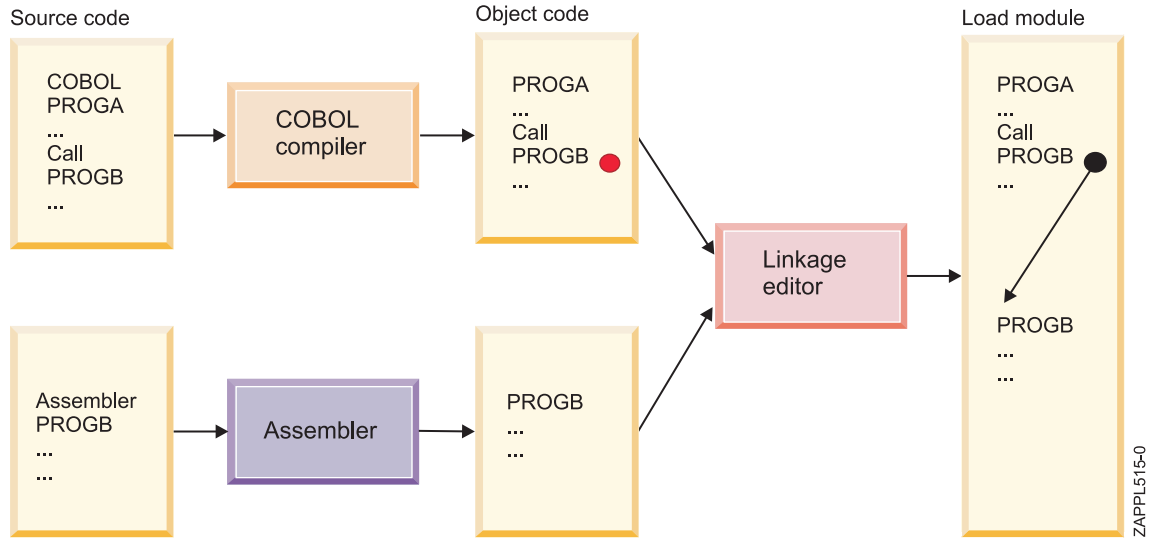

Figure 40. Resolving references during load module creation

The binder

The **binder** provided with z/OS performs all of the functions of the linkage editor. The binder link-edits (combines and edits) the individual object decks, load modules, and program objects that comprise an application and produces a single program object or load module that you can load for execution. When a member of a program library is needed, the loader brings it into virtual storage and prepares it for execution.

You can use the binder to:

- Convert an object deck or load module into a program object and store it in a partitioned data set extended (PDSE) program library, or in a z/OS UNIX file.
- Convert an object deck or program object into a load module and store it in a partitioned data set (PDS) program library. This is equivalent to what the linkage editor does with object decks and load modules.
- Convert object decks or load modules, or program objects, into an executable program in virtual storage and execute the program. This is equivalent to what the batch loader does with object decks and load modules.

The binder processes object decks, load modules and program objects, link-editing or binding multiple modules into a single load module or program object. Control statements specify how to combine the input into one or more load modules or program objects with contiguous virtual storage addresses. Each object deck can be processed separately by the binder, so that only the modules that have been modified need to be recompiled or reassembled. The binder can create programs in 24-bit, 31-bit and 64-bit addressing modes.

You assign an addressing mode (**AMODE**) to indicate which hardware addressing mode is active when the program executes. Addressing modes are:

- 24, which indicates that 24-bit addressing must be in effect
- 31, which indicates that 31-bit addressing must be in effect
- 64, which indicates that 64-bit addressing must be in effect
- ANY, which indicates that 24-bit, 31-bit, or 64-bit addressing can be in effect
- MIN, which requests that the binder assign an **AMODE** value to the program module.

The binder selects the most restrictive **AMODE** of all control sections in the input to the program module. An **AMODE** value of 24 is the **input to the program module. An AMODE value of 24 is the most** restrictive; **an AMODE value of ANY is the least** restrictive.

All of the services of the linkage editor can be performed by the binder.

Binder and linkage editor

The binder relaxes or eliminates many restrictions of the linkage editor. The binder removes the linkage editor's limit of 64 aliases, allowing a load module or program object to have as many aliases as desired. The binder accepts any system-supported block size for the primary (SYSLIN) input data set, eliminating the linkage editor's maximum block size limit of 3200 bytes. The binder also does not restrict the number of external names, whereas the linkage editor sets a limit of 32767 names.

By the way, the prelinker provided with z/OS Language Environment is another facility for combining object decks into a single object deck. Following a pre-link, you can link-edit the object deck into a load module (which is stored in a PDS), or bind it into a load module or a program object (which is stored in a PDS, PDSE, or zFS file). With the binder, however, z/OS application programmers no longer need to pre-link, because the binder handles all of the functionality of the pre-linker. Whether you use the binder or linkage editor is a matter of preference. The binder is the latest way to create your load module.

The primary input, required for every binder job step, is defined on a DD statement with the ddname SYSLIN. Primary input can be:

- A sequential data set
- A member of a partitioned data set (PDS)
- A member of a partitioned data set extended (PDSE)
- Concatenated sequential data sets, or members of partitioned data sets or PDSEs, or a combination
- A z/OS UNIX file.

The primary data set can contain object decks, control statements, load modules and program objects. All modules and control statements are processed sequentially, and their order determines the order of binder processing. The order of the sections after processing, however, might not match the input sequence. Figure 41 shows a job that can be used to link-edit an object deck. The output from the LKED step will be placed in a private library identified by the SYSLMOD DD. The input is passed from a previous job step to a binder job step in the same job (for example, the output from the compiler is direct input to the binder).

```
//LKED   EXEC PGM=IEWL,PARM='XREF,LIST', IEWL is IEWBLINK alias
//           REGION=2M,COND=(5,LT,prior-step)
//*
//*        Define secondary input
//*
//SYSLIB   DD  DSN=language.library,DISP=SHR         optional
//PRIVLIB  DD  DSN=private.include.library,DISP=SHR  optional
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))          ignored
//*
//*        Define output module library
//*
//SYSLMOD  DD  DSN=program.library,DISP=SHR          required
//SYSPRINT DD  SYSOUT=*                              required
//SYSTERM  DD  SYSOUT=*                              optional
//*
//*        Define primary input
//SYSLIN   DD  DSN=&&OBJECT,DISP=(MOD,PASS)          required
//*
//         DD  * inline control statements
                INCLUDE  PRIVLIB(membername)
                NAME     modname(R)
/*
```

Figure 41. Binder JCL example

An explanation of the JCL statements follows:

EXEC Binds a program module and stores it in a program library. Alternative names for IEWBLINK are IEWL, LINKEDIT, EWL, and HEWLH096. The

PARM field option requests a cross-reference table and a module map to be produced on the diagnostic output data set.

SYSUT1
Defines a temporary direct access data set to be used as the intermediate data set.

SYSLMOD
Defines a temporary data set to be used as the output module library.

SYSPRINT
Defines the diagnostic output data set, which is assigned to output class A.

SYSLIN
Defines the primary input data set, &&OBJECT, which contains the input object deck; this data set was passed from a previous job step and is to be passed at the end of this job step.

INCLUDE
Specifies sequential data sets, library members, or z/OS UNIX files that are to be sources of additional input for the binder (in this case, a member of the private library PRIVLIB).

NAME
Specifies the name of the program module created from the preceding input modules, and serves as a delimiter for input to the program module. (R) indicates that this program module replaces an identically named module in the output module library.

Load modules for executable programs

A **load module** is an executable program stored in a partitioned data set program library. Creating a load module to execute only, will require that you use a batch loader or program management loader. Creating a load module that can be stored in a program library requires that you use the binder or linkage editor. In all cases, the load module is relocatable, which means that it can be located at any address in virtual storage within the confines of the residency mode (**RMODE**).

Once a program is loaded, control is passed to it, with a value in the base register. This gives the program its starting address, where it was loaded, so that all addresses can be resolved as the sum of the base plus the offset. Relocatable programs allow an identical copy of a program to be loaded in many different address spaces, each being loaded at a different starting address.

Batch loader

The **batch loader** combines the basic editing and loading services (which can also be provided by the linkage editor and program manager) into one job step. The batch loader accepts object decks and load modules, and loads them into virtual storage for execution. Unlike the binder and linkage editor, the batch loader does not produce load modules that can be stored in program libraries. The batch loader prepares the executable program in storage and passes control to it directly.

Batch loader processing is performed in a load step, which is equivalent to the link-edit and go steps of the binder or linkage editor. The batch loader can be used for both compile-load and load jobs. It can include modules from a call library (SYSLIB), the link pack area (LPA), or both. Like the other program management components, the batch loader supports addressing and residence mode attributes

in the 24-bit, 31-bit, and 64-bit addressing modes. The batch loader program is reentrant and therefore can reside in the resident link pack area.

Note: In more recent releases of z/OS, the binder replaces the batch loader.

Program management loader

The program management loader increases the services of the program manager component by adding support for loading program objects. The loader reads both program objects and load modules into virtual storage and prepares them for execution. It resolves any address constants in the program to point to the appropriate areas in virtual storage and supports the 24-bit, 31-bit and 64-bit addressing modes.

In processing object and load modules, the linkage editor assigns consecutive relative virtual storage addresses to control sections and resolves references between control sections. Object decks produced by several different language translators can be used to form one load module.

In Figure 42 on page 72 we have a compile, link-edit, and execute job, in this case for an assembler program.

```
//USUAL     JOB  A2317P,'COMPLGO'
//ASM       EXEC PGM=IEV90,REGION=256K,                    EXECUTES ASSEMBLER
//          PARM=(OBJECT,NODECK,'LINECOUNT=50')
//SYSPRINT  DD   SYSOUT=*,DCB=BLKSIZE=3509                 PRINT THE ASSEMBLY LISTING
//SYSPUNCH  DD   SYSOUT=B                                  PUNCH THE ASSEMBLY LISTING
//SYSLIB    DD   DSNAME=SYS1.MACLIB,DISP=SHR THE MACRO LIBRARY
//SYSUT1    DD   DSNAME=&&SYSUT1,UNIT=SYSDA,               A WORK DATA SET
//          SPACE=(CYL,(10,1))
//SYSLIN    DD   DSNAME=&&OBJECT,UNIT=SYSDA,               THE OUTPUT OBJECT DECK
//          SPACE=(TRK,(10,2)),DCB=BLKSIZE=3120,DISP=(,PASS)
//SYSIN     DD    *                                        inline SOURCE CODE
                                       .
                                       .
                                       code
                                       .
/*
//LKED      EXEC PGM=HEWL,                                  EXECUTES LINKAGE EDITOR
//          PARM='XREF,LIST,LET',COND=(8,LE,ASM)
//SYSPRINT  DD   SYSOUT=*                                   LINKEDIT MAP PRINTOUT
//SYSLIN    DD   DSNAME=&&OBJECT,DISP=(OLD,DELETE)          INPUT OBJECT DECK
//SYSUT1    DD   DSNAME=&&SYSUT1,UNIT=SYSDA,                A WORK DATA SET
//          SPACE=(CYL,(10,1))
//SYSLMOD   DD   DSNAME=&&LOADMOD,UNIT=SYSDA,               THE OUTPUT LOAD MODULE
//          DISP=(MOD,PASS),SPACE=(1024,(50,20,1))
//GO        EXEC PGM=*.LKED.SYSLMOD,TIME=(,30),             EXECUTES THE PROGRAM
//          COND=((8,LE,ASM),(8,LE,LKED))
//SYSUDUMP  DD   SYSOUT=*                                   IF FAILS, DUMP LISTING
//SYSPRINT  DD   SYSOUT=*,                                  OUTPUT LISTING
//          DCB=(RECFM=FBA,LRECL=121)
//OUTPUT    DD   SYSOUT=A,                                  PROGRAM DATA OUTPUT
//          DCB=(LRECL=100,BLKSIZE=3000,RECFM=FBA)
//INPUT     DD    *                                         PROGRAM DATA INPUT
               .
               .
               data
               .
/*
//
```

Figure 42. Compile, link-edit, and execute JCL

Notes:

- In the step ASM (compile), SYSIN DD is for the inline source code and SYSLIN DD is for the output object deck.
- In the step LKED (linkage-edition), the SYSLIN DD is for the input object deck and the SYSLMOD DD is for the output load module.
- In the step GO (execute the program), the EXEC JCL statement states that it will execute a program identified in the SYSLMOD DD statement of the previous step.
- This example does not use a cataloged procedure, as the COBOL examples did; instead, all of the JCL has been coded inline. We could have used an existing JCL procedure, or coded one and then only supplied the overrides, such as the INPUT DD statement.

What is a load library?

A **load library** contains programs ready to be executed.

A load library can be any of the following:

- System library
- Private library

- Temporary library.

System library

Unless a job or step specifies a private library, the system searches for a program in the system libraries when you code:

```
//stepname  EXEC  PGM=program-name
```

The system looks in the libraries for a member with a name or alias that is the same as the specified program-name. The most-used system library is SYS1.LINKLIB, which contains executable programs that have been processed by the linkage editor.

Private library

Each executable, user-written program is a member of a private library. To tell the system that a program is in a private library, the DD statement defining that library can be coded in one of the following ways:

- With a DD statement with the ddname JOBLIB after the JOB statement, and before the first EXEC statement in the job.
- If the library is going to be used in only one step, with a DD statement with the ddname STEPLIB in the step.

To execute a program from a private library, code:

```
//stepname  EXEC  PGM=program-name
```

When you code JOBLIB or STEPLIB, the system searches for the program to be executed in the library defined by the JOBLIB or STEPLIB DD statement before searching in the system libraries.

If an earlier DD statement in the job defines the program as a member of a private library, refer to that DD statement to execute the program:

```
//stepname  EXEC  PGM=*.stepname.ddname
```

Private libraries are particularly useful for programs used too seldom to be needed in a system library. For example, programs that prepare quarterly sales tax reports are good candidates for a private library.

Temporary library

Temporary libraries are partitioned data sets created to store a program until it is used in a later step of the **same** job. A temporary library is created and deleted within a job.

When testing a newly written program, a temporary library is particularly useful for storing the load module from the linkage editor until it is executed by a later job step. Because the module will not be needed by other jobs until it is fully tested, it should not be stored in a private library or a system library.

In Figure 43 on page 74, the LKED step creates a temporary library called &&LOADMOD on the SYSLMOD DD statement. In the GO step, we refer back to the same temporary data set by coding:

```
//GO EXEC PGM=*.LKED.SYSLMOD,....
```

```
//USUAL     JOB  A2317P,'COMPLGO'
//ASM       EXEC PGM=IEV90,REGION=256K,                    EXECUTES ASSEMBLER
//          PARM=(OBJECT,NODECK,'LINECOUNT=50')
//SYSPRINT  DD   SYSOUT=*,DCB=BLKSIZE=3509                 PRINT THE ASSEMBLY LISTING
//SYSPUNCH  DD   SYSOUT=B                                  PUNCH THE ASSEMBLY LISTING
//SYSLIB    DD   DSNAME=SYS1.MACLIB,DISP=SHR THE MACRO LIBRARY
//SYSUT1    DD   DSNAME=&&SYSUT1,UNIT=SYSDA,               A WORK DATA SET
//          SPACE=(CYL,(10,1))
//SYSLIN    DD   DSNAME=&&OBJECT,UNIT=SYSDA,               THE OUTPUT OBJECT DECK
//          SPACE=(TRK,(10,2)),DCB=BLKSIZE=3120,DISP=(,PASS)
//SYSIN     DD    *                                        inline SOURCE CODE
                                       .
                                       .
                                       code
                                       .
/*
//LKED      EXEC PGM=HEWL,                                  EXECUTES LINKAGE EDITOR
//          PARM='XREF,LIST,LET',COND=(8,LE,ASM)
//SYSPRINT  DD   SYSOUT=*                                   LINKEDIT MAP PRINTOUT
//SYSLIN    DD   DSNAME=&&OBJECT,DISP=(OLD,DELETE)          INPUT OBJECT DECK
//SYSUT1    DD   DSNAME=&&SYSUT1,UNIT=SYSDA,                A WORK DATA SET
//          SPACE=(CYL,(10,1))
//SYSLMOD   DD   DSNAME=&&LOADMOD,UNIT=SYSDA,               THE OUTPUT LOAD MODULE
//          DISP=(MOD,PASS),SPACE=(1024,(50,20,1))
//GO        EXEC PGM=*.LKED.SYSLMOD,TIME=(,30),             EXECUTES THE PROGRAM
//          COND=((8,LE,ASM),(8,LE,LKED))
//SYSUDUMP  DD   SYSOUT=*                                   IF FAILS, DUMP LISTING
//SYSPRINT  DD   SYSOUT=*,                                  OUTPUT LISTING
//          DCB=(RECFM=FBA,LRECL=121)
//OUTPUT    DD   SYSOUT=A,                                  PROGRAM DATA OUTPUT
//          DCB=(LRECL=100,BLKSIZE=3000,RECFM=FBA)
//INPUT     DD    *                                         PROGRAM DATA INPUT
               .
               .
               data
               .
/*
//
```

Figure 43. Compile, link-edit, and execute JCL

Compilation to execution

A diagram shows the relationship between the object decks and the load module stored in a load library, and then loaded into central memory for execution.

In Figure 44 on page 75, we can see the relationship between the object decks and the load module stored in a load library, and then loaded into central memory for execution.

We start with two programs, A and B, which are compiled into two object decks. Then the two object decks are linked into one load module call MYPROG, which is stored in a load library on direct access storage. The load module MYPROG is then loaded into central storage by the program management loader, and control is transferred to it to for execution.

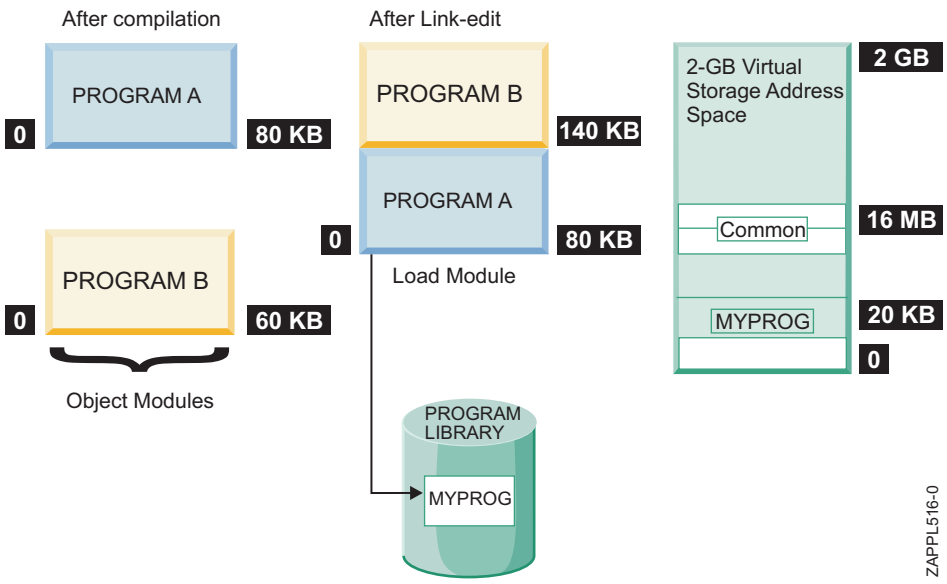

Figure 44. Program compile, link-edit, and execution

How procedures are used

To save time and prevent errors, you can prepare sets of job control statements and place them in a partitioned data set (PDS) or partitioned data set extended (PDSE), known as a **procedure library**. This can be used, for example, to compile, assemble, link-edit, and execute a program.

A procedure library is a library that contains procedures. A set of job control statements in the system procedure library, SYS1.PROCLIB or an installation-defined procedure library, is called a **cataloged procedure**.

To test a procedure before storing it in a procedure library, add the procedure to the input stream and execute it; a procedure in the input stream is called an **inline procedure**. The maximum number of inline procedures you can code in any job is 15. In order to test a procedure in the input stream, it must end with a procedure end (PEND) statement. The PEND statement signals the end of the PROC. This is only required when the procedure is coded inline. In a procedure library, you do not require a PEND statement.

An inline procedure must appear in the same job before the EXEC statement that calls it.

```
//DEF     PROC STATUS=OLD,LIBRARY=SYSLIB,NUMBER=777777
//NOTIFY  EXEC PGM=ACCUM
//DD1     DD   DSNAME=MGMT,DISP=(&STATUS,KEEP),UNIT=3400-6,
//             VOLUME=SER=888888
//DD2     DD   DSNAME=&LIBRARY,DISP=(OLD,KEEP),UNIT=3390,
//             VOLUME=SER=&NUMBER
```

Figure 45. Sample definition of a procedure

Three symbolic parameters are defined in the cataloged procedure shown in Figure 45 on page 75; they are &STATUS, &LIBRARY, and &NUMBER. Values are assigned to the symbolic parameters on the PROC statement. These values are used if the procedure is called but no values are assigned to the symbolic parameters on the calling EXEC statement.

In Figure 46 we are testing the procedure called DEF. Note that the procedure is delineated by the PROC and PEND statements. The EXEC statement that follows the procedure DEF references the procedure to be invoked. In this case, since the name DEF matches a procedure that was previously coded inline, the system will use the procedure inline and will not search any further.

```
//TESTJOB  JOB ....
//DEF      PROC STATUS=OLD,LIBRARY=SYSLIB,NUMBER=777777
//NOTIFY   EXEC PGM=ACCUM
//DD1      DD   DSNAME=MGMT,DISP=(&STATUS,KEEP),UNIT=3400-6,
//              VOLUME=SER=888888
//DD2      DD   DSNAME=&LIBRARY,DISP=(OLD,KEEP),UNIT=3390,
//              VOLUME=SER=&NUMBER
//         PEND
//*
//TESTPROC EXEC DEF
//
```

Figure 46. Testing a procedure inline

Notices

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

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 grant 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 (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A.

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 document 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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains 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.

Programming interface information

This publication documents information that is NOT intended to be used as Programming Interfaces of z/OS.

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 "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Celeron®, Intel® Inside®, Intel SpeedStep®, Intel, Itanium®, Pentium®, Xeon®, Intel logo, Intel Inside® logo, and Intel Centrino® logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

IPX, Java, Sun™, Sun Microsystems™, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft®, Windows® NT®, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

Other company, product, or service names may be trademarks or service marks of others.

Printed in USA