Contents

Preface

This course provides students of information systems technology with the background knowledge and skills necessary to begin using the basic facilities of a mainframe computer.

For optimal learning, students are assumed to have successfully completed an introductory course in computer system concepts, such as computer organization and architecture, operating systems, data management, or data communications. They should also have successfully completed courses in one or more programming languages, and be PC literate.

Note that this course can also be used as a prerequisite for courses in advanced topics such as compiler algorithms, or for internships and special studies.

Others who will benefit from this course include data processing professionals who have experience with non-mainframe platforms, or who are familiar with some aspects of the mainframe but want to become knowledgeable with other facilities and benefits of the mainframe environment.

When moving through this course, instructors are encouraged to alternate between course, lecture, discussions, and hands-on exercises. The instructor-led discussions and hands-on exercises are an integral part of the learning experience, and can include topics not covered in this course.

After reading this course, students will have received:

- A general introduction to mainframe concepts, usage, and architecture
- A comprehensive overview of z/OS, a widely used mainframe operating system
- An understanding of mainframe workloads and an overview of the major middleware applications in use on mainframes today
- The basis for subsequent course work in more advanced, specialized areas of z/OS, such as system administration or application programming

How this course is organized

This course is organized in four parts, as follows:

- **Part 1. "Introduction to z/OS and the mainframe environment"** provides an overview of the types of workloads commonly processed on the mainframe, such as batch jobs and online transactions. This part of the course helps students explore the

user interfaces of z/OS, a widely used mainframe operating system. Discussion topics include TSO/E and ISPF, UNIX interfaces, job control language, file structures, and job entry subsystems. Special attention is paid to the users of mainframes and to the evolving role of mainframes in today's business world.

- **Part 2. "Application programming on z/OS"** introduces the tools and utilities for developing a simple program to run on z/OS. This part of the course guides the student through the process of application design, choosing a programming language, and using a runtime environment.
- **Part 3. "Online workloads for z/OS"** examines the major categories of interactive workloads processed by z/OS, such as transaction processing, database management, and Web-serving. This part of the course includes discussions of several popular middleware products, including DB2®, CICS®, and WebSphere® Application Server.
- **Part 4. "System programming on z/OS"** provides topics to help the student become familiar with the role of the z/OS system programmer. This part of the course includes discussions of system libraries, starting and stopping the system, security, network communications and the clustering of multiple systems. Also provided is an overview of mainframe hardware systems, including processors and I/O devices.

In this course, we use simplified examples and focus mainly on basic system functions. Hands-on exercises are provided throughout the course to help students explore the mainframe style of computing. Exercises include entering work into the system, checking its status, and examining the output of submitted jobs.

How each topic is organized

Each topic follows a common format:

- Objectives for the student
- Topics that teach a central theme related to mainframe computing
- Summary of the main ideas of the topic
- A list of key terms introduced in the topic
- Questions for review to help students verify their understanding of the material
- Topics for further discussion to encourage students to explore issues that extend beyond the topic objectives
- Hands-on exercises intended to help students reinforce their understanding of the material

Application programming on z/OS

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

1

Designing and developing applications for z/OS

Objective: One of the problems faced by distributed 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 z/OS 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, secure way according to workload or priority.

After completing this topic, you will be able to:

- List the major considerations for designing an application for z/OS
- Describe the advantages and disadvantages of using batch versus online for an application.
- Briefly describe the process for testing a new application on z/OS
- List three advantages for using z/OS as the host for a new application.

1.1 Application designers and programmers

An application is a collection of programs that provides one or more functions. An application program can be thought of as a group of 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.

The tasks of *designing* an application and *developing* one are distinct enough to treat each in a separate course. In larger z/OS sites, separate departments might be used to carry out each task. This topic 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 (sometimes called the application architect) is responsible for determining the best programming solution for an important business requirement. The success of any design will depend 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 existing I/T infrastructure. 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 the business systems analyst and from 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 follow.

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 systems.

1.2 Designing an application for z/OS

In designing an application for z/OS, one should first evaluate whether an existing product would provide the necessary framework. For example, if you need to create a transaction processing application to run on z/OS, you should first evaluate whether an existing product like CICS, IMS, or WebSphere Application Server would provide the solution. Such an evaluation should consider both functionality and cost. These products incur an ongoing license fee that does not exist with a home-grown application. This topic assumes that you have decided to create own application from scratch.

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 topic 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?" on page 6
- "Designing for z/OS: Data sources and access methods" on page 6
- "Designing for z/OS: Integrating a multi-tiered application" on page 7
- "Designing for z/OS: Availability and workload requirements" on page 8
- "Designing for z/OS: Exception Handling" on page 8

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. We discuss the use of programming languages on a mainframe in Chapter 2, "Using programming languages on z/OS" on page 23, but the choice of programming language is frequently dictated by the issue of portability. That is, will this mainframe application also run on other platforms? If so, languages like C, C++ and Java are usually preferred.

Other considerations discussed in this topic include the following:

- Using mainframe character sets in “Using the EBCDIC character set” on page 14
- Use of an interactive development environment (IDE) in “Using application development tools” on page 17

Keep in mind that the best systems 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.

1.2.1 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 on-line 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?

The most obvious reason for using batch is the need to process large amounts of data in a manner that does not require interaction with humans.

As discussed earlier in this book, batch jobs can be submitted in a number of ways, including:

- JCL submitted from TSO
- JCL submitted by a job scheduler product
- Started task - started by the operator
- Shell command submitted to run in the background.

Starting the desired program can be done in a variety of ways:

- Standard JCL with EXEC PGM=program_name
- JCL running BPXBATCH to run programs in a UNIX environment.
- JCL running BPXBATSL to run programs in a UNIX environment that access z/OS data sets

The main consideration in determining which of the above environments to run in is the language in which the application is coded. Java and C programs frequently require a UNIX environment, which is provided by a shell or BPXBATCH environment.

For long running batch jobs, you need to consider whether you want to Checkpoint/Restart to snapshot the job during processing. This will allow the job to be restarted from the last checkpoint if a failure prevents the job from completing. Checkpoint/Restart does not support the UNIX environment, so most C and Java applications cannot take advantage of Checkpoint/Restart.

1.2.2 Designing for z/OS: Data sources and access methods

- 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, UNIX directories and files, or a database management system (DBMS) such as DB2?

1.2.3 Designing for z/OS: Integrating a multi-tiered application

How is an application integrated into the larger mainframe scheme?

In general, when you talk about designing a multi-tiered application, you are talking about writing a server to run on z/OS. The server can be handling transactions from other programs or directly from users. There are all sorts of issues that can be discussed when writing a server application, for example:

- How many users do you need to concurrently support?
- Will you be able to support that number of users in a single address space (process) or will multiple processes be needed?
- How many tasks (threads) are reasonable to run in a single process. Too many threads can cause contention for resources (e.g. heap)
- How will the server authenticate users? Userid/password or certificate, for example?
- Will the work be executed under the userid of the requestor or under the userid of the server?
- Are all transactions equal or will there be some very long running transactions? How will the priority of each transaction be managed? Having long running transactions run with the same high priority as short user requests can adversely affect response time.
- How will applications connect to the mainframe? New applications tend to use TCP/IP. Assuming sockets, there are encryption options that must be considered. SSL is available or users can do their own encryption/decryption.

  - Design considerations for integrating a z/OS application with a z/OS transaction manager
  - Design considerations for integrating a z/OS application with a z/OS database manager

Example of a typical application topology on the mainframe

As an example, a typical server in a multi-tiered application would be as follows:

- Server started with a STARTED PROC or possibly INITTAB for UNIX style applications
- Listen on a well known port. Use asynch I/O to manage a large number of sockets
- Accept connections creating a socket to the end user. Set up SLL encryption if necessary
- Direct work to a separate thread to handle each active user request

- Authenticate user
- Each thread processes a transaction (possibly establishing the user identity on the thread) and then waits for more work
- For more complex servers, logic is needed to start multiple server address spaces (processes). WebSphere Application Server is often used for this purpose.

1.2.4 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? In particular, does the data need to be shared between online systems and batch systems, and if so, will there be a conflict and possible deadlocks?
- What are the response time requirements?
- What are the cost constraints of the project? Basing an application on a mainframe can provide some compelling cost benefits over a distributed system, which spreads its components across different machines and platforms. With z/OS, 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.
- 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, 8:00a.m - 5:00 p.m., and so on)?

1.2.5 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. This is an area which many application developers ignore; however, we cannot always assume, for example, that input will always be entered as expected.

In a z/OS system, approximately 25% of the system code is devoted to dealing with errors and handling bad input. Your application should be designed to:

- Validate input before using it
- Issue detailed error messages to simplify debugging
- Consider adding diagnostic tracing to the application which can be turned on and off
- Where possible, terminate a single transaction without taking down the entire server

The language you choose to code in has a large bearing on the types of recovery that can be exploited. C and Java support signal handlers. COBOL and PL1 have some exception processing. Assembler supports ESTAE and ESPIE.

1.3 Application development life cycle: An overview

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

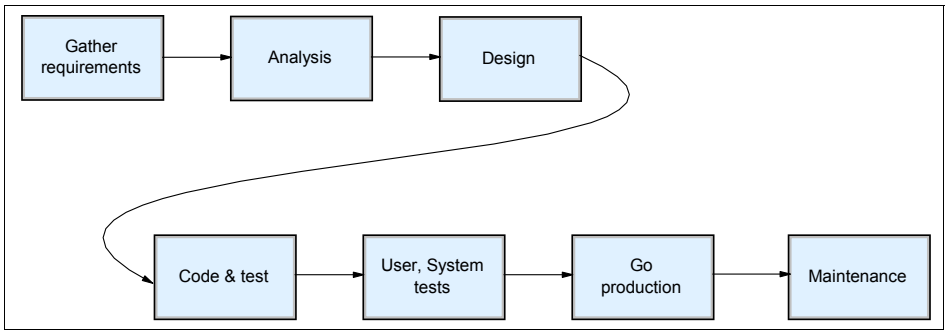

Figure 1-1 Application development life cycle

As with other operating systems, application development on z/OS includes the following phases:

1. **Design the application:**
   - Gather requirements: User, hardware and software requirements
   - Analyze requirements.
   - Create a High-level design, then a low-level (detailed) design
   - Hand over design to the Application Programming staff
2. **Develop the application:**
   - Code the application
   - Perform unit testing of the application
3. **Test the application:**
   - Perform *user testing* to ensure that the application functions properly and is easy to use
   - Perform *integration testing* to ensure that the changed application and the other existing programs continue to function as expected
   - Perform *performance testing*—with realistic production data—to ensure that the execution of the changed application has not been elongated, nor its performance been degraded
4. **Implement the application:**
   - "Go production"—turn over the application to the Operations staff
   - Ensure that all documentation is in place, such as user training materials and operation procedures
5. **Maintain the application:**

- Ongoing day-to-day changes and enhancements to application
- Diagnosing problems; these come in many flavors, such as ABENDs, incorrect output, hangs, excessive CPU, or slow response time.

Figure 1-2 depicts the design phase up to the point of starting development. After 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.

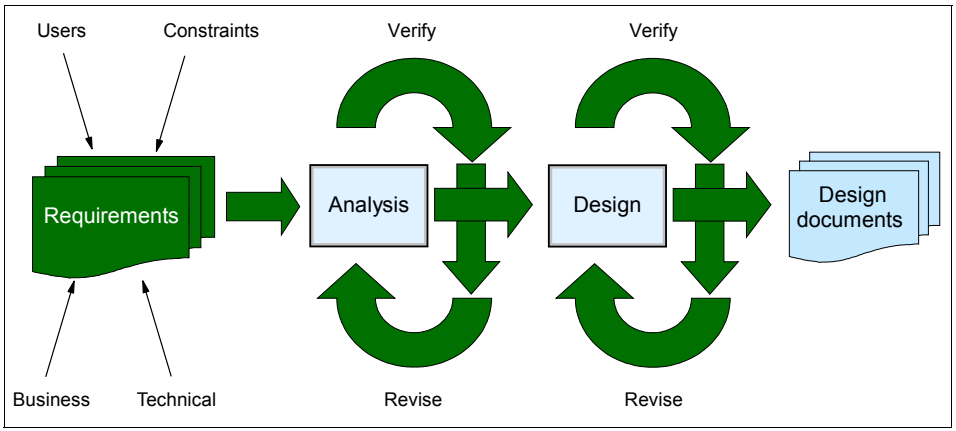

Figure 1-2 Design phase

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 1-3.

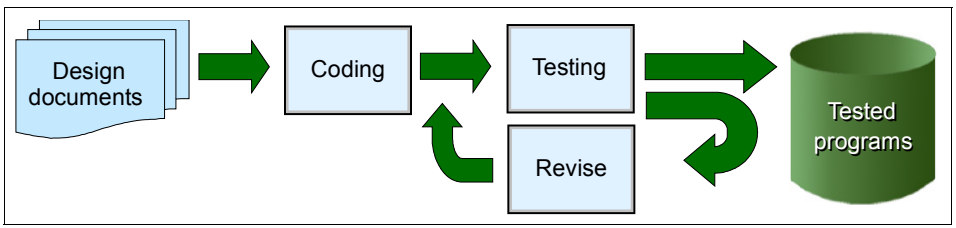

Figure 1-3 Development phase

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 1-4).

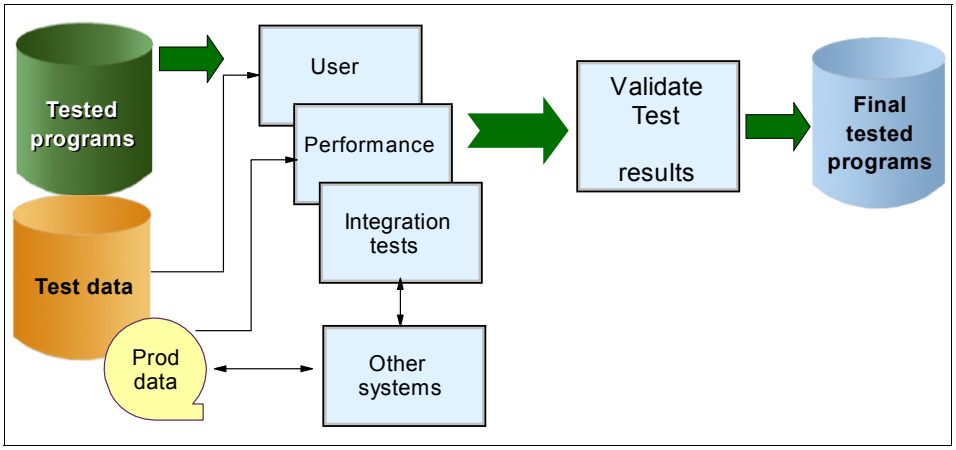

Figure 1-4 Testing

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.

Once in production, the changes and enhancements are handled by a group (possibly the same programming group) that performs this 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 1-5).

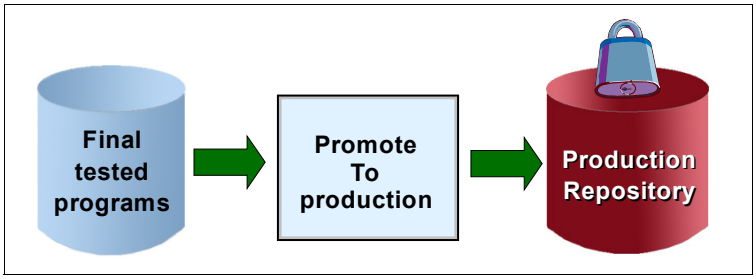

Figure 1-5 Production

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 1-6, 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.

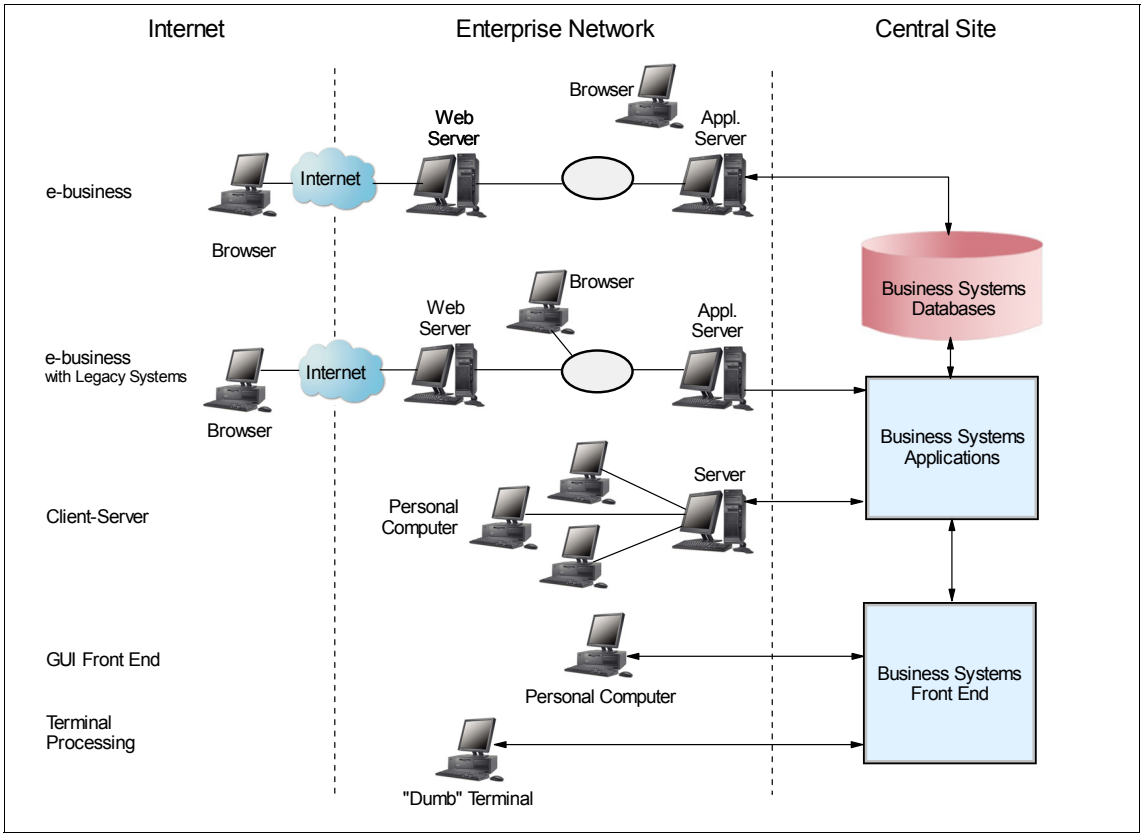

Figure 1-6 Growing infrastructure complexity

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 many 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 each phase of the project, we also ask the stakeholders to sign off on what we have done, so that we can proceed on to the next phase of the project.

1.3.1 Gathering requirements for the design

Gathering requirements is arguably the most important phase of application design. If the requirements of a project are not understood and met, the project will likely be considered a failure. 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 in 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 z/OS 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 that runs on z/OS. The list is not exclusive and some items already include others.

Accessibility	Client	Interoperability
Recoverability	Serviceability	Availability
Connectivity	Performance	Resource can be monitored, controlled, managed, and administered
Usability	Frequency of data backup	Distributed
Portability	Secure centralized controllable capacity	Web services
Changeability	Inter-communicable	Preventing failure and fault analysis

1.4 Developing an application for 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 rein, 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 whole process repeats itself 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.

1.4.1 Using the EBCDIC character set

z/OS data sets are encoded in the Extended Binary Coded Decimal Interchange (EBCDIC) character set. This is an 8-bit character set that was developed before 8-bit ASCII (American Standard Code for Information Interchange) became commonly used. Even z/OS UNIX files are encoded in EBCDIC.

In contrast, most other systems that you are familiar with use the ASCII system of character encoding. You need to be aware of the difference in encoding 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*, and the programmer must look at the actual value of the translated character.

A listing of EBCDIC and ASCII bit assignments might be useful for this discussion. 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. Both mainframes (using EBCDIC for single-byte characters), PCs, and various RISC systems use the same Unicode assignments. Unicode provides 16-bit characters, intending to include all modern languages in use today.

Traditional mainframe programming does not make much use of special characters to terminate fields[1]. 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

[1] Although x'15' for New Line is used extensively for variable length records.

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

The use of Unicode is slowly growing. The latest IBM 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.

1.4.2 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 supports a full range of UNIX services and is branded XPG/4. The z/OS implementation of UNIX application interfaces is known collectively as z/OS UNIX System Services, or z/OS UNIX for short.

One common interface for z/OS developers is through 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 topic 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.

1.4.3 Using 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 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 to 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, build the application modules, and create a program ready to run (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.

Some mainframe application programmers have now switched to the use of interactive development environment (IDE) tools to accelerate the edit/compile/test process. IDEs such as the WebSphere Studio Enterprise Developer allow application programmers to edit, test, and debug source code on a workstation ("off-platform") instead of directly on the mainframe system.

The use of the IDE is particularly useful for building "hybrid" applications that employ host-based programs in COBOL or transaction systems, such as CICS or IMS, but also contain a Web browser-like user interface. The IDE provides a unified development environment to build both the on-line transaction processing (OLTP) components in a high-level language and the HTML front-end user interface components.

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.

Some of the application enablement services that are available on z/OS include the following:

IBM WebSphere Studio Enterprise Developer	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	DBX

As we learned earlier in this course, a data set contains the information that you want to sort, copy, or merge. DFSORT (or Data Facility sort) is an example of an optional, application enablement product that is available for working with data sets. Programmers use DFSORT to enable their z/OS applications to sort, merge, or copy the records in data sets (*sorting* means arranging records in either ascending or descending order within a file). You can use DFSORT for simple tasks such as alphabetizing a list of names, or you can use it to for complex tasks such as taking inventory or running a billing system. DFSORT provides versatile data handling capabilities at the record, field and bit level.

Related Reading: You can find more information about DFSORT in the IBM publication *z/OS DFSORT Application Programming Guide*, which is available at the z/OS Internet Library Web site:

http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

1.4.4 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[2]
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)

1.4.5 Maintaining existing code

Besides developing 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.

1.4.6 Performing a system test

The difference between a unit test and a system test 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 - The user tests the application for functionality and usability.
- Integration testing - The new application is tested together with other applications to see if they work together as expected.

[2] 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”.

- Performance testing - The application is tested using realistic production data to ensure that the application completes in a reasonable amount of time and performance has not been degraded.

The results of the user and integration tests need to be verified to ensure that they are satisfactory. For some high volume applications (such as Web applications or CICS transactions) we might require additional testing known as stress testing. Here, the application is run with a realistic volume of activity (such as database updates) to determine how well the application performs when there is high demand.

Any issues resulting from these tests must be addressed before the application is placed in production. The number of issues encountered during the testing phase is one indication of how well the application was designed.

1.5 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 anyone who interacts with the system.

When all of the documentation and training has been done, then 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.

1.6 Summary

This topic 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 topic then 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 attempts to identify the relevant parts of the problem to be solved. The designer analyzes user requirements to see how best to satisfy them. There may be many ways to arrive at the same 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 there are problems with the system test, the problems will need to 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 only 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.

Key terms in this topic		
application	ASCII	design
DFSORT	EBCDIC	enablement
executable	user requirements	system testing

2

Using programming languages on z/OS

Objective: As your company's newest z/OS application programmer, you will need to know which programming languages are supported on z/OS, and how to determine which is best for a given set of requirements.

After completing this topic, you will be able to:

- List several common programming languages for the mainframe
- Explain the differences between a compiled language and an interpreted language
- Create a simple CLIST or REXX program
- Choose an appropriate data file organization for an online application
- Compare the advantages of a high level language to those of Assembler language
- Explain the relationship between a data set name, a DD name, and the file name within a program
- Explain how the use of z/OS Language Environment affects the decisions made by the application designer

2.1 Overview of programming languages

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 - “Using Assembler language on z/OS” on page 26
- COBOL - “Using COBOL on z/OS” on page 28
- PL/I - “Using PL/I on z/OS” on page 35
- C/C++ - “Using C/C++ on z/OS” on page 39
- Java - “Using Java on z/OS” on page 40
- CLIST - “Using CLIST language on z/OS” on page 42
- REXX - “Using REXX on z/OS” on page 44

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-dependen*t, 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 topic 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 2.11, "Compiled versus interpreted languages" on page 46.

2.2 Choosing a programming language for z/OS

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.

2.3 Using 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. It is the symbolic programming language that is closest to the machine language in form and content. Therefore, Assembler language 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[1] 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.
- 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[2]. 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

[1] 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.

[2] 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".

process is called *linkage-editing* (or *link-editing*, for short) and is performed by the binder.

The binder or linkage editor (for more details, see 3.3.7, “How is a linkage editor used?” on page 75) 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 2-1 on page 27 shows these steps.

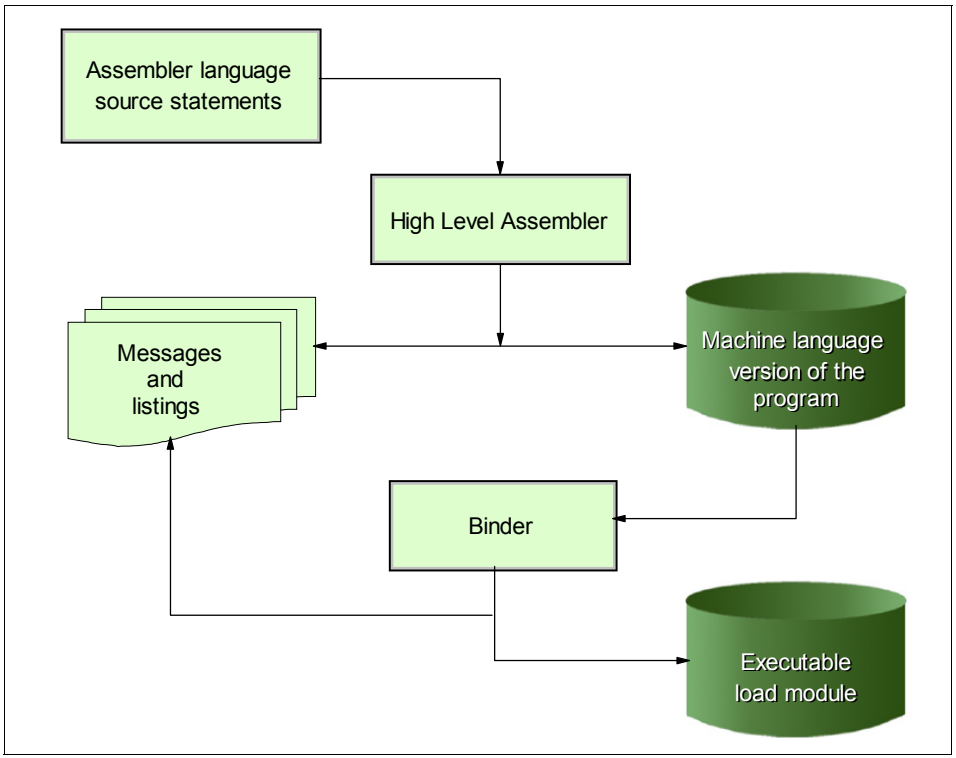

Figure 2-1 Assembler source to executable module

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. These books are available on the Web at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/find_shelves.html`

2.4 Using 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 topic, 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, applications 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 (these are further discussed in 3.3.7, “How is a linkage editor used?” on page 75). 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 2-2 on page 29 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.

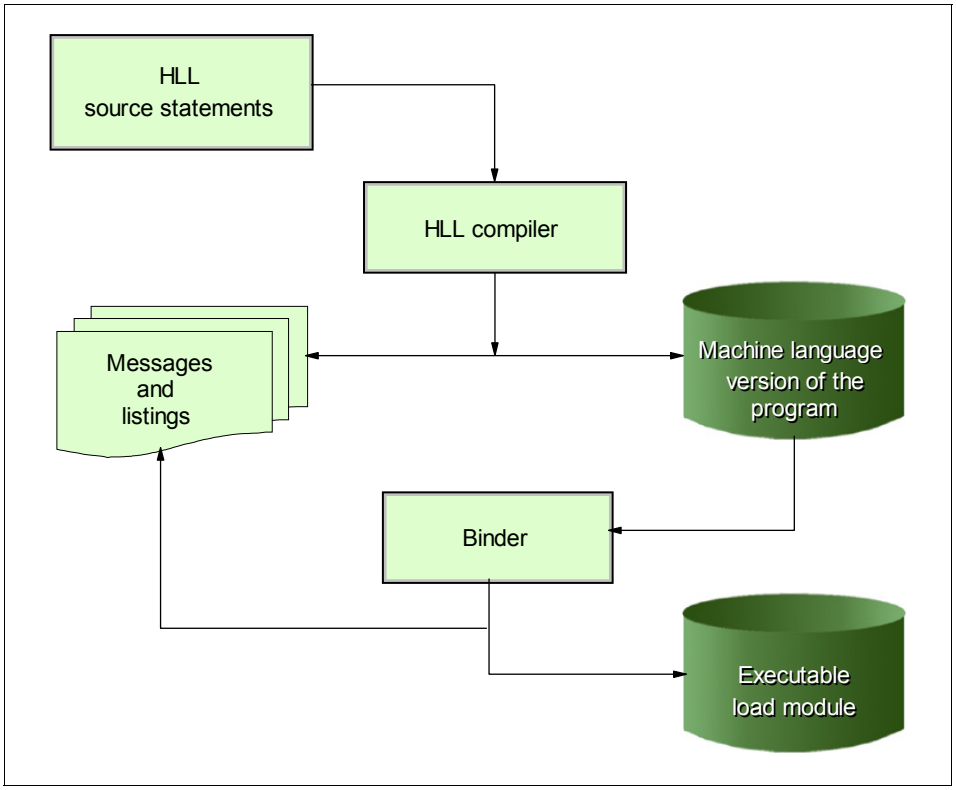

Figure 2-2 HLL source to executable module

2.4.1 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 the following 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. These 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 describe data from another program.
- 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** - is a series of one or more COBOL statements ending with a period.
  - **Statement** - performs a defined step of COBOL processing, such as adding two numbers.
  - **Phrase** - a subdivision of a statement.

Examples of COBOL divisions

Example 2-1 IDENTIFICATION DIVISION

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

Example 2-2 ENVIRONMENT 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.
```

Explanations of the user-supplied information follow Example 2-3.

Example 2-3 Input and output files in FILE-CONTROL

```
     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.
```

- `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.
- Others like `filename, recordname, fieldname` (nn in the example), `fieldlength` and `type` are also specified.

2.4.2 COBOL relationship between JCL and program files

Example 2-4 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.

Example 2-4 COBOL relationship between JCL and program files

```
//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
```

Example 2-4 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 make 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 2-3 shows the relationship between the physical data set, the JCL, and the program for Example 2-4.

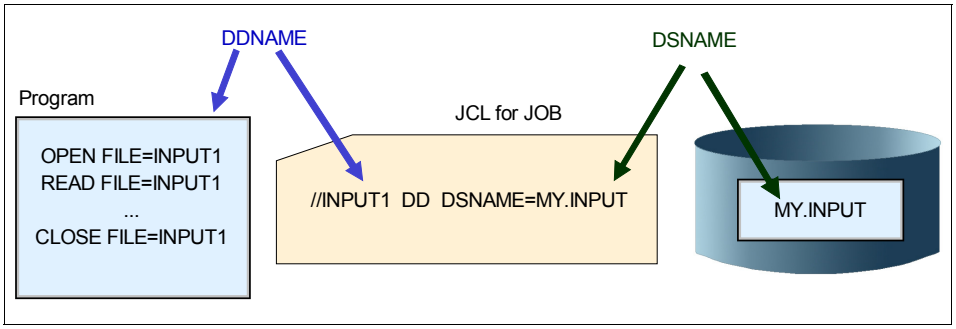

Figure 2-3 Relationship between JCL, program, and data set

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.

2.4.3 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.

2.4.4 Communicating with Java methods

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. You need to code COBOL object-oriented language for basic Java object capabilities. 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.

2.4.5 Creating a DLL or a DLL application

A dynamic link library or DLL is 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.

2.4.6 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: You can find more information about using COBOL on z/OS in 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. These books are available on the Web at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/find_shelves.html`

2.5 HLL relationship between JCL and program files

In 2.4.2, “COBOL relationship between JCL and program files” on page 32, 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 Example 2-5 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.

Example 2-5 HLL Relationship between JCL and program files

```
//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
```

2.6 Using PL/I on z/OS

Programming Language/I (PL/I, pronounced “P-L one”), is a full function, general-purpose, high-level programming language suitable for the development of:

- Commercial applications
- Engineering/scientific applications

- Many other applications

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.

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. See Example 2-2 on page 29, 3.3.7, “How is a linkage editor used?” on page 75and Figure 2-3 on page 33.

The relationship between JCL and program files is the same for PL/I as it is for COBOL and other HLLs. See Figure 2-3 on page 33 and to Example 2-5 on page 35.

2.6.1 PL/I program structure

PL/I is a block-structured language, consisting of packages, procedures, *begin blocks*, statements, expressions, and built-in functions as it is shown in Figure 2-4.

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 class. 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.

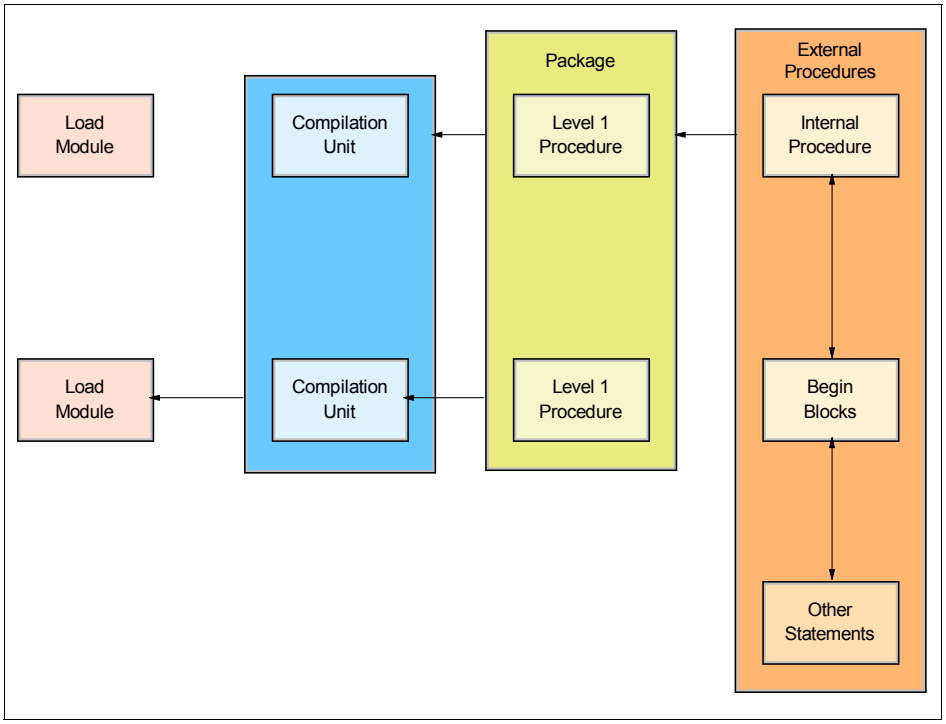

Figure 2-4 PL/I application structure

A PL/I application consists of one or more separately loadable entities, known as a *load module*. Each load module can consist of one or more separately compiled entities, known as a *compilation unit*. 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 Example 2-6. A procedure can be a main procedure, a subroutine, or a function. An application must have exactly one external procedure that has OPTIONS(MAIN).

Example 2-6 A PROCEDURE block

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

```
      statement-2

     .

     .

     .
      statement-n
      end Name;
```

A *BEGIN block* is a sequence of statements delimited by a BEGIN statement and a corresponding END statement, as shown in Example 2-7.

A program is terminated when the main procedure is terminated.

Example 2-7 BEGIN block

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

     .

     .

     statement-n
     end B;
```

2.6.2 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 with a generalized subroutine, macros allow generation of only the code that is needed for each individual use.

- ***SQL preprocessor.*** In general, the coding for your PL/I program will be the same whether or not you want it to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data and use other DB2 services, you must use SQL statements. You can use dynamic and static EXEC SQL statements in PL/I applications.

  To communicate with DB2, you need to do the following:

  - Code any SQL statements you need, delimiting them with EXEC SQL.
  - Use the DB2 precompiler or compile with the PL/I PP(SQL()) compiler option.

  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: You can find more information about using PL/1 on z/OS in 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. These books are available on the Web at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/find_shelves.html`

2.6.3 Using the SAX parser

The PL/I compiler provides an interface called PLISAXx (x = A or B) that provides you with basic XML capability to PL/I. The support includes a high-speed XML parser, which allows programs to consume 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 run-time environments, including CICS, IMS, and MQ Series, as well as z/OS batch and TSO.

2.7 Using C/C++ on z/OS

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

- System-level code
- Text processing
- Graphics

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.

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. See Example 2-2 on page 29, 3.3.7, “How is a linkage editor used?” on page 75, and Figure 2-3 on page 33 to see this process.

The relationship between JCL and program files is the same for PL/I as it is for COBOL and other HLLs. See Figure 2-3 on page 33 and to Example 2-5 on page 35.

Related Reading: You can find more information about using C and C++ on z/OS in the IBM publications, *C/C++ Language Reference*, SC09-4764, and *C/C++ Programming Guide*, SC09-4765. These books are available on the Web at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/find_shelves.html`

2.8 Using 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 and 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 program products 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 Language. These languages provide a set of interfaces or facilities for interacting with programs written in Java, as explained for COBOL in 2.4.4, “Communicating with Java methods” on page 33 and for PL/I in 2.6.3, “Using the SAX parser” on page 39.

The various Java Software Development Kit (SDK) program products 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.

2.8.1 IBM SDK products for z/OS

As with Java SDKs for other IBM platforms, z/OS Java SDK program products are supplied for industry standard APIs. Each of the z/OS SDK products is 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 SDK1.3.1 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. The IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4 product at the SDK 1.4 level 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 is operational within the z/OS Version 1 Release 4 operating system or later, or z/OS.e Version 1 Release 4 operating system 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 is operational within the z/OS Version 1 Release 6 operating system 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/`

2.8.2 Using 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:

- Utilize 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 these ways, 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.

2.9 Using CLIST language on z/OS

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 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 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)
- One-time quick solutions to problems

2.9.1 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 together in a CLIST the instructions required to complete a task, you reduce the time, number of keystrokes, and errors involved in performing the task; thus, you increase your productivity. Such 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.

2.9.2 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 sub-commands and CLIST statements, but you can use the END sub-command in a CLIST to end EDIT or TEST mode and allow the CLIST to issue TSO/E commands.)

2.9.3 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

2.10 Using 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. A single language, NetRexx, may 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. See Figure 2-3 on page 33 and to Example 2-5 on page 35.

2.10.1 Compiling and executing REXX command lists

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.

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. See Example 2-2 on page 29, 3.3.7, “How is a linkage editor used?” on page 75and Figure 2-3 on page 33 to see this process.

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 runtime.
- 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 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
- *Procedures Language Reference (Level 1)*, C26-4358 SAA® CPI
- *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.ibm.com/software/awdtools/REXX/language/REXXlinks.html`

2.11 Compiled versus interpreted languages

During the design of an application, you might need to decide between using an 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 versus non-ad hoc.

2.11.1 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.

2.11.2 Advantages of interpreted languages

In “Advantages of compiled languages” we discussed the reasons for using languages that are compiled. In “Using CLIST language on z/OS” and “Using REXX on z/OS” we discussed the strong points of 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 suited for a particular language.

2.12 What is z/OS Language Environment?

As we mentioned in Chapter 1, “Designing and developing applications for z/OS” on page 3an 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.

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.

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.

2.12.1 How Language Environment is used

Language Environment establishes a common run-time environment for all participating HLLs. It combines essential run-time services, such as routines for run-time 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 run-time environment for your applications, regardless of the application's programming language or system resource needs.

Figure 2-5 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 run-time library.

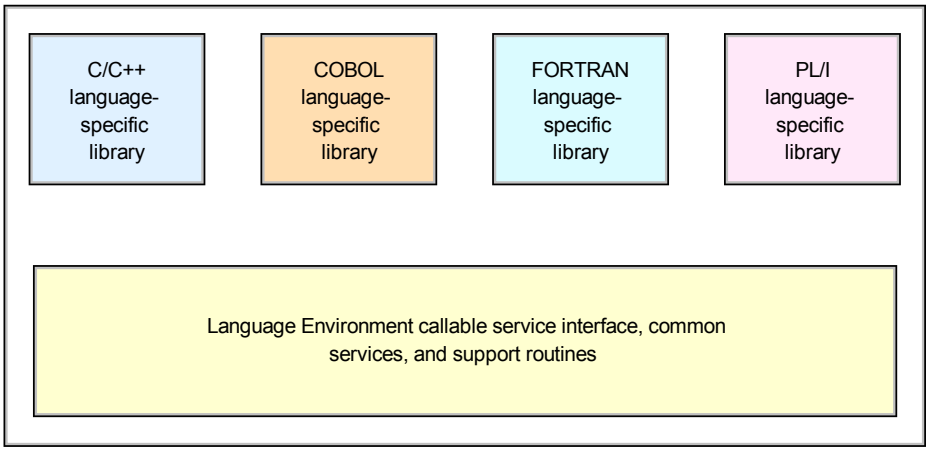

Figure 2-5 z/OS Language Environment components

Language Environment is the prerequisite run-time 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.

2.12.2 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 2-6 shows a common run-time environment established through Language Environment.

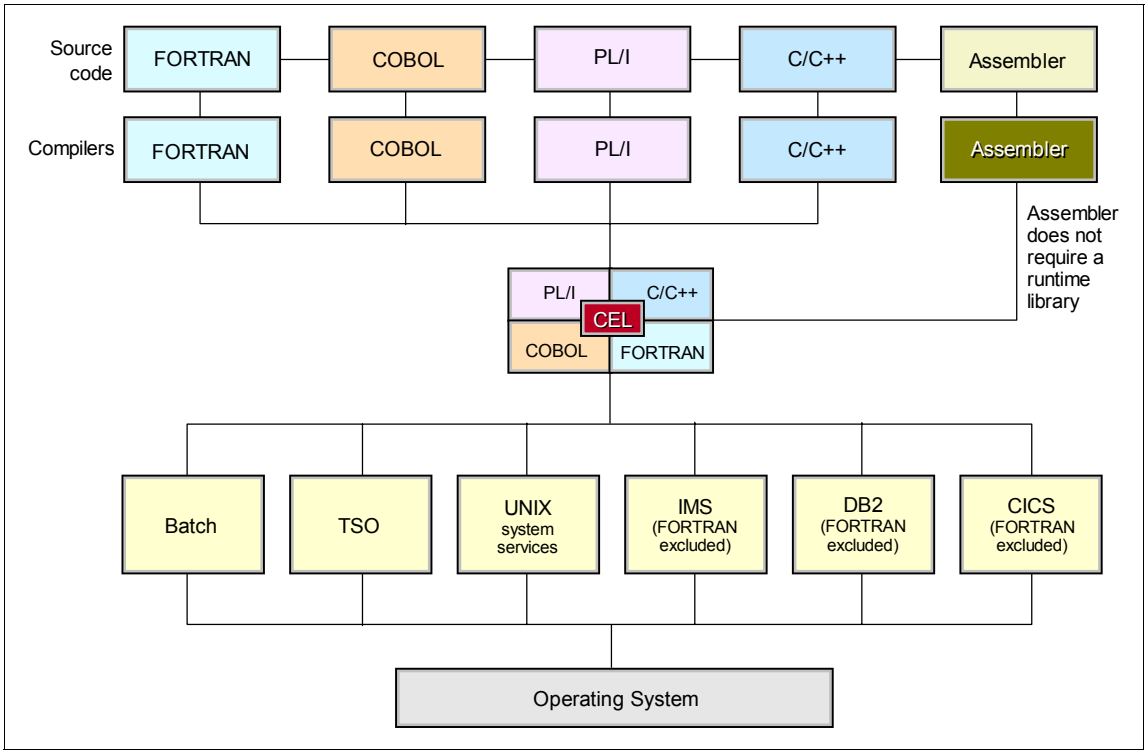

Figure 2-6 Language Environment's common runtime environment

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

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

Program management model

The Language Environment program management model provides a framework within which an application runs. It is the foundation of all of the component models (condition handling, run-time 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 components management 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 run-time 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 run-time 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.

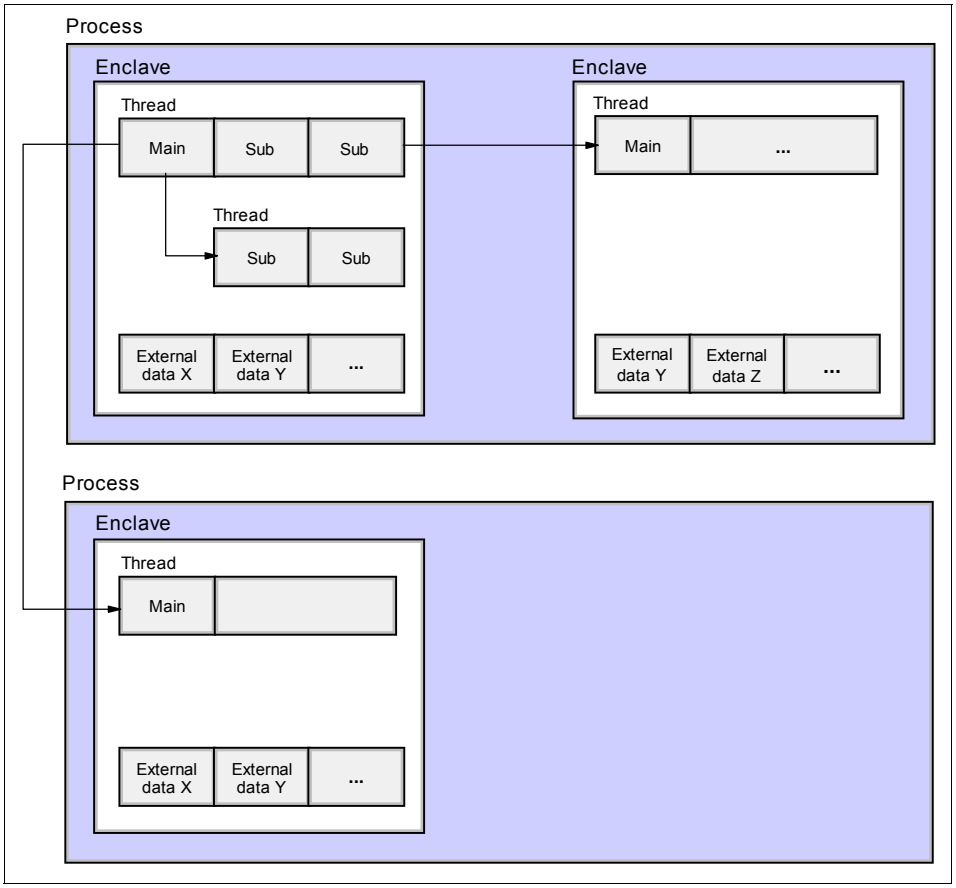

Figure 2-7 Full Language Environment program model

Figure 2-7 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. A main routine might not be active at all times in a POSIX application, if the thread in which the main routine executes terminates before the other threads it created.

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 run-time 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 run-time 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.

2.12.3 Running your program with Language Environment

After compiling your program you can do the following:

- Link-edit and run an existing object deck and accept the default Language Environment run-time options
- Link-edit and run an existing object deck and specify new Language Environment run-time 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 run-time options, you can use a Language Environment-provided link-edit and run cataloged procedure CEEWLG. The CEEWLG procedure identifies the Language Environment libraries that your object deck needs to link-edit and run.

Run-time 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 run-time library routines needed during execution of applications written in C/C++, PL/I, COBOL and FORTRAN. SCEERUN2 contains the run-time library routines needed during execution of applications written in C/C++ and COBOL.

Applications that require the run-time 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/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

There is a common set of callable services designed to supplement the programmer's language intrinsic capability. For example, 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/Termination Services
- Locale Callable Services
- Math Services
- Message Handling Services
- National Language Support Services

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

Language Environment calling conventions

Language Environment services can be invoked by HLL library routines, other Language Environment services, and user-written HLL calls. In many cases, services will be invoked by HLL library routines as a result of a user-specified function, such as a COBOL intrinsic function. Following are examples of the invocation of a callable math service from three of the languages we have described in this topic. Also look at the referenced examples in 2.9.3, “Other uses for the CLIST language” on page 44.

Example 2-8 shows how a COBOL program invokes the math callable services CEESDLG1 for log base 10.

Example 2-8 Sample invocation of a math callable service from a COBOL program

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

2.13 Summary

This topic outlines the many decisions you might need to make when you design and develop an application to run on 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.

Key terms in this topic		
assembler	binder	compiler
debugging	dynamic link library	generation
I/O (input/output)	interpreter	load modules
preprocessor	programming language	variable

3

Compiling and link-editing a program on z/OS

Objective: As your company's newest z/OS application programmer, you will be asked to create new programs to run on z/OS. Doing so will require you to know how to compile, link, and execute a program.

After completing this topic, you will be able to:

- Explain the purpose of a compiler
- Compile a source program
- Explain the difference between the linkage editor and the binder
- Create executable code from a compiled program
- Explain the difference between an object deck and a load module
- Run a program on z/OS

3.1 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 3-1.

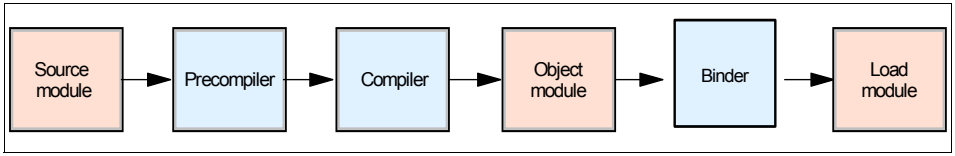

Figure 3-1 Source, object, and load modules

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.

3.2 What are source libraries?

Source programs (or *source code*) is a set of statements written in an computer language, as discussed in Chapter 2, "Using programming languages on z/OS" on page 23. 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 Example 3-1 on page 63, we insert the text in member INPUTRCD from the library DEPT88.BOBS.COBLIB into the source program that is to be compiled.

Example 3-1 Copybook in COBOL source code

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

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.

3.3 Compiling programs 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. See 3.4, “Creating load modules for executable programs” on page 80 for more information on relocatable programs. 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 Example 3-2.

Example 3-2 SYSIN DD statement for the source code

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

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.

3.3.1 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.

3.3.2 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

The JCL in Example 3-3 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 Example 3-4.

Example 3-3 Basic JCL for compiling a COBOL source program inline

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

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 Example 3-4 make up the IGYWC cataloged procedure used in Example 3-3. As mentioned previously, the result of the compilation process, the compiled program, is placed in the data set identified on the SYSLIN DD statement.

Example 3-4 Procedure IGYWC - COBOL compile

```
//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))
```

COBOL pre-processor and compile and link procedure

The JCL in Example 3-5 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 Example 3-6 on page 67. 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 Example 3-5, you can see that the JCL is a bit more complicated than in the simple compile job (Example 3-3 on page 65). 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 Example 3-6 on page 67, 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 Example 3-5 (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.

Example 3-5 Basic JCL for pre-processing, compiling, and linking a COBOL source program inline

```
//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)
/*
```

The statements shown in Example 3-6 make up the DFHEITVL cataloged procedure used in Example 3-5 on page 67. 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.

Example 3-6 Procedure DFHEITVL - COBOL preprocessor, compile, and link

```
//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
//*
//*      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))
//*
//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
```

COBOL compile and link procedure

The JCL in Example 3-7 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 Example 3-8. 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 modules in the data set referenced by the SYSLMOD DD statement.

The end result of running the JCL in Example 3-7 (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.

Example 3-7 Basic JCL for compiling and linking a COBOL source program inline

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

The statements shown in Example 3-8 make up the IGYWCL cataloged procedure used in Example 3-7. 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.

Example 3-8 Procedure IGYWCL - COBOL compile and link

```
//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))
```

COBOL compile, link and go procedure

The JCL in Example 3-9 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 (Example 3-7 on page 69). However, whereas in Example 3-7 on page 69 we override the SYSLMOD DD statement in order to permanently save the load module,

in Example 3-9, we do not need to save it in order to execute it. That is why the override to the SYSLMOD DD statement in Example 3-9 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 Example 3-9, you can see that the JCL is very similar to the JCL used in the simple compile job (Example 3-3 on page 65). Looking at the JCL in Example 3-10 on page 71, the only difference between it and the JCL in Example 3-8 on page 70 is that we have added the GO step. The end result of running the JCL in Example 3-9 (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.

Example 3-9 Basic JCL for compiling, linking and executing a COBOL source program inline

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

The statements shown in Example 3-10 make up the IGYWCLG cataloged procedure used in Example 3-9 on page 71.

Example 3-10 Procedure IGYWCLG - COBOL compile, link, and go

```
//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=*
```

3.3.3 Compiling object-oriented (OO) applications

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[1] environment structure to access JNI callable services, copy the file JNI.cpy from the HFS to a PDS or PDSE 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.

[1] 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.

As shown in Example 3-11, use the SYSJAVA ddname to write the generated Java source file to a file in the HFS. For example:

Example 3-11 SYSJAVA ddname for a Java source file

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

3.3.4 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.

3.3.5 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.

3.3.6 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 3-2 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.

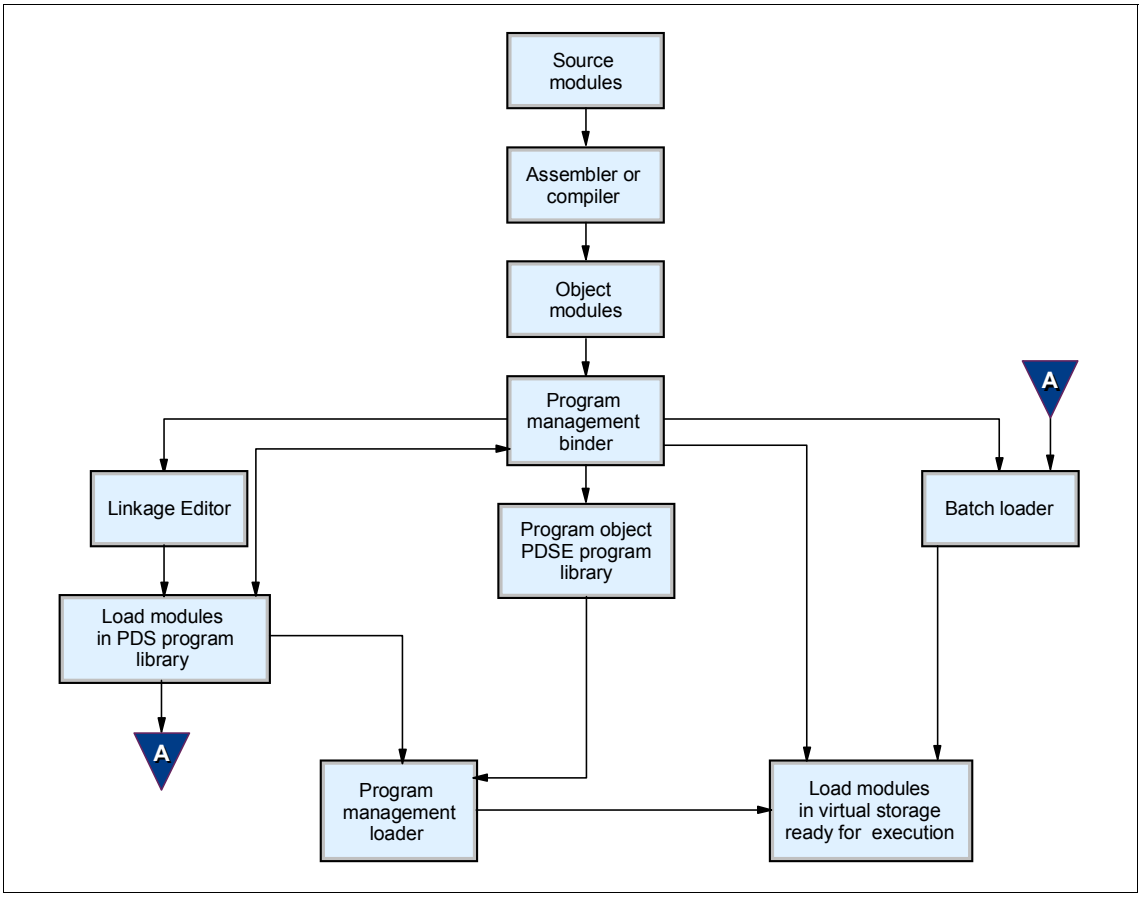

Figure 3-2 Using Program Management components to create and load programs

3.3.7 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 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.

3.3.8 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 3-3 on page 77 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.

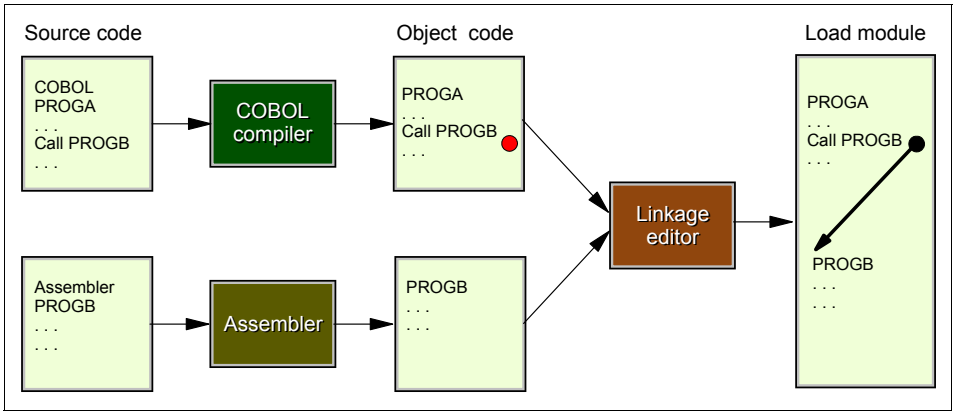

Figure 3-3 Resolving references during load module creation

Using 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 *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.

Binder example

Example 3-12 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).

Example 3-12 Binder JCL example

```
//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)
/*
```

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.

3.4 Creating 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. See 3.3, “Compiling programs on z/OS” on page 63 for more discussion on relocatable programs.

3.4.1 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.

3.4.2 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 Example 3-13 we have a compile, link-edit, and execute job, in this case for an assembler program.

Example 3-13 Compile, link-edit, and execute JCL

```
//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
                  .
/*
//
```

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.

3.4.3 What is a load library?

A *load library* is a library that 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 Example 3-13 on page 81, 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,....
```

3.5 Overview of compilation to execution

In Figure 3-4, 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.

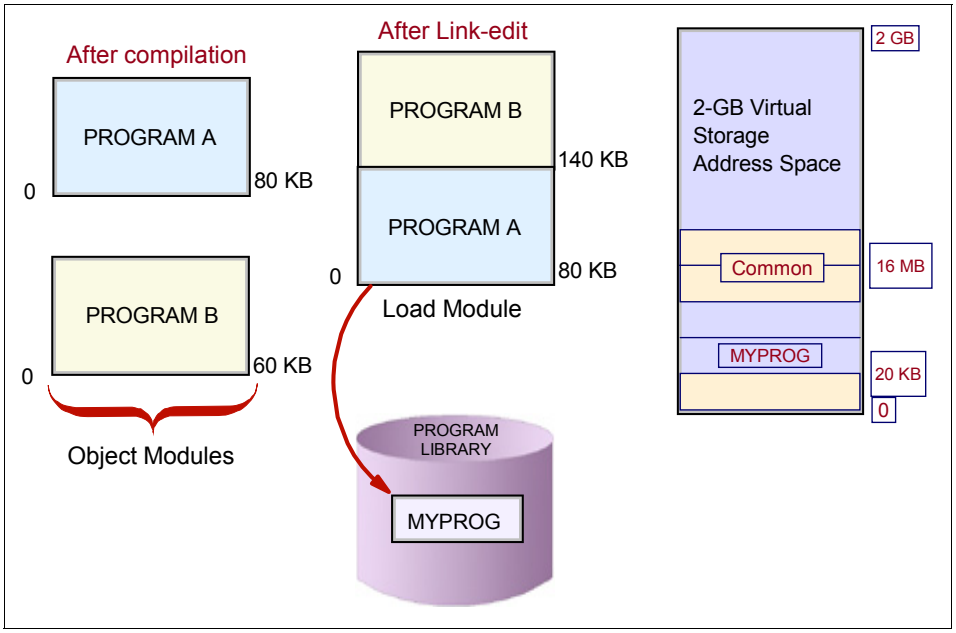

Figure 3-4 Program compile, link-edit, and execution

3.6 Using procedures

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, as we have seen in Example 3-13 on page 81.

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, we do not require a PEND statement.

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

Example 3-14 Sample definition of a procedure

```
//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
```

Three symbolic parameters are defined in the cataloged procedure shown in Example 3-14: &STATUS, &LIBRARY, and &NUMBER. Values are assigned to the symbolic parameters on the PROC statement. These values are used if the procedure is called and no values are assigned to the symbolic parameters on the calling EXEC statement.

In Example 3-15 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.

Example 3-15 Testing a procedure inline

```
//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
//
```

3.7 Summary

This topic described the process for translating a source program into an executable load module, and executing the load module. The basic steps for this translation are to compile and link-edit, 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 first 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, to 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 real 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[2].

Key terms in this topic		
binder	copybook	linkage editor
load module	object deck	object-oriented code
procedure	procedure library	program library
relocatable	source module	

[2] 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.