IBM

Series/1

GC34-0121-0

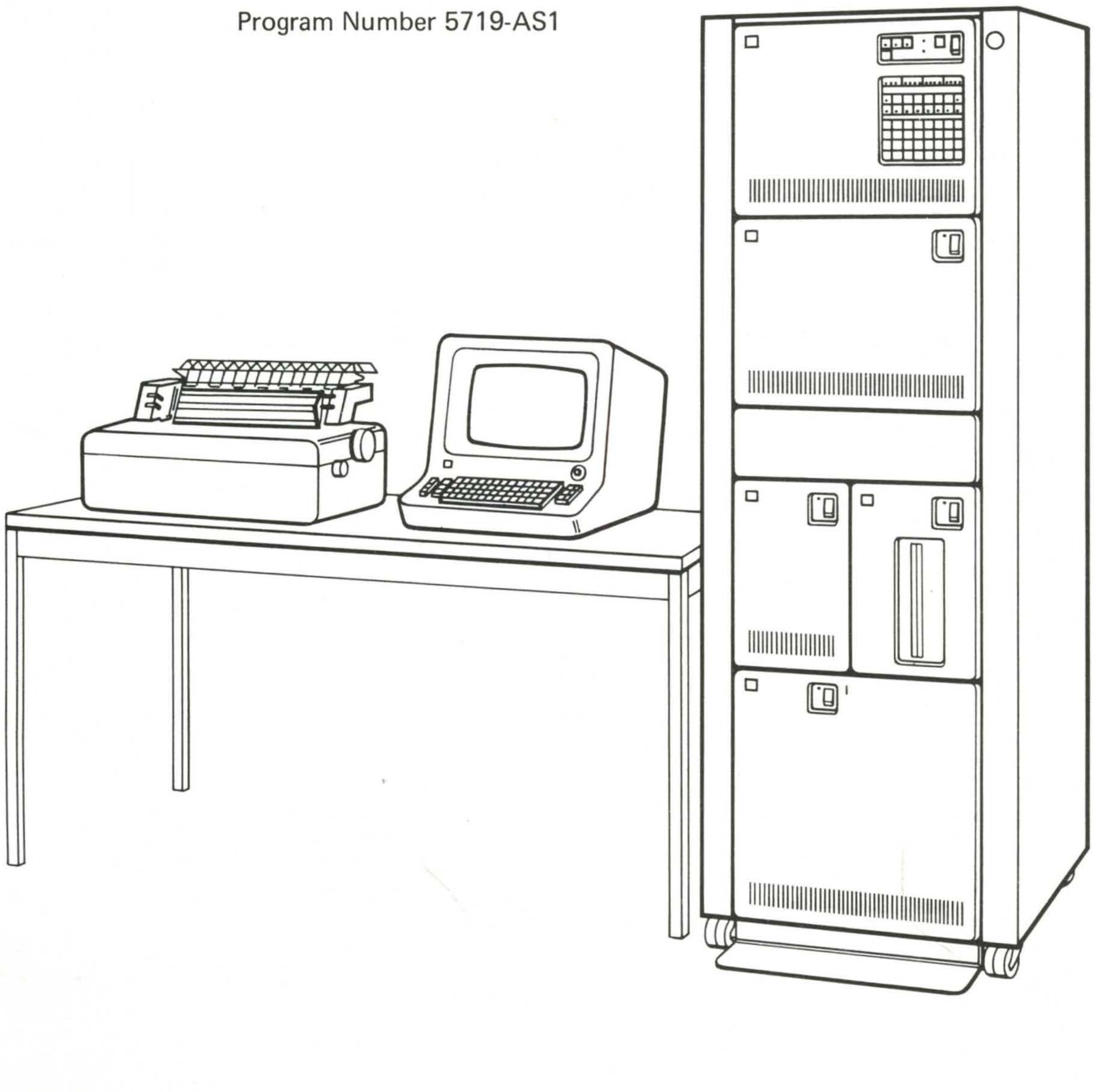
S1-20

PROGRAM
PRODUCT

PROGRAM PREP INTRODUCTION

IBM Series/1 Program Preparation Subsystem Introduction

Program Number 5719-AS1



Series/1

GC34-0121-0

S1-20

PROGRAM
PRODUCT

IBM Series/1 Program Preparation Subsystem Introduction

Program Number 5719-AS1

PROGRAM PREP INTRODUCTION

This publication is for planning purposes only. The information herein is subject to change before the products described become available.

First Edition (February 1977)

This manual applies to the IBM Series/1 Program Preparation Subsystem, 5719-AS1.

Significant changes or additions to the contents of this publication will be reported in subsequent revisions or Technical Newsletters. Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, send your comments to IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432. Comments become the property of IBM.

Contents

Preface

This book provides introductory information about the IBM Series/1 Program Preparation Subsystem for the reader who is evaluating the subsystem for applicability to his installation, as well as for the reader who simply wants a general understanding of the subsystem facilities.

The reader should have a basic knowledge of programming systems and be familiar with the information in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

How this Book is Organized

Chapter 1. Overview of the Program Preparation Subsystem. This chapter describes the purpose, organization, functions, and operating environment of the components of the Program Preparation Subsystem.

Chapter 2. The Job Stream Processor. This chapter explains how the job stream processor control statements are used to invoke batch programs and define the data sets and devices required by those programs.

Chapter 3. The Text Editor. This chapter explains how the text editor is used to create, modify, list and save text modules.

Chapter 4. The Macro Assembler. This chapter describes assembler language features and explains how the assembler program is used to create relocatable object modules.

Chapter 5. The Application Builder. This chapter explains how the application builder processes object modules to prepare them for execution as application programs.

For each of these components, there is a description of its purpose, data sets and devices used, the input required, the processing sequence, and the output created. Each of these chapters also directs the reader to the appropriate manual for more detailed information about the component.

Appendix A. Configuration Requirements. This appendix contains information about hardware and software requirements and compatibilities.

Appendix B. Using the Series/1 Programming Library. This appendix is a quick-reference guide to help you determine the other Series/1 manuals you may need for various types of programming activities.
A glossary is provided to define terms referenced in this manual.

Related Publications

The following hardware and programming publications contain more detailed information about topics covered in this book.

Note. Order numbers are shown only for the publications that are available at this time.

Programming Publications

IBM Series/1 Program Preparation Subsystem: Batch User's Guide

IBM Series/1 Program Preparation Subsystem: Text Editor User's Guide

IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide, SC34-0124

IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide

IBM Series/1 Realtime Programming System: Introduction and Planning Guide, GC34-0102

IBM Series/1 FORTRAN IV: Introduction, GC34-0132

IBM Series/1 PL/I: Introduction, GC34-0084

IBM Series/1 Mathematical and Functional Subroutine Library: Introduction, GC34-0138

Hardware Publications

IBM Series/1 Model 5 4955 Processor and Processor Features Description, GA34-0021

IBM Series/1 Model 3 4953 Processor and Processor Features Description, GA34-0022

IBM Series/1 4962 Disk Storage Unit Description and 4964 Diskette Unit Description, GA34-0024

IBM Series/1 4973 Printer Description, GA34-0044

IBM Series/1 4974 Printer Description, GA34-0025

IBM Series/1 4979 Display Station Description, GA34-0026

IBM Series/1 System Summary, GA34-0035-1

Chapter 1. Overview of the Series/1 Program Preparation Subsystem

The Program Preparation Subsystem is a set of programs in the IBM Series/1 software system that offers:

- A general-purpose disk-based job stream processor.
- Powerful program preparation facilities for creating realtime and batch applications.

The job stream processor reads, analyzes, and processes the *job input stream*, which is a sequence of your requests for invoking batch programs and defining the data sets and devices they use.

The program preparation facilities—the text editor, macro assembler, and application builder—allow you to create, update, and assemble source programs, and then build executable application programs.

The programs that make up the Program Preparation Subsystem will also be referred to in this manual as the subsystem programs or the subsystem components.

You can also use PL/I, FORTRAN IV, and the Mathematical and Functional Subroutine Library (MFSL) with the subsystem programs. Information about the Series/1 software—the program products and how they form a total system—is contained in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

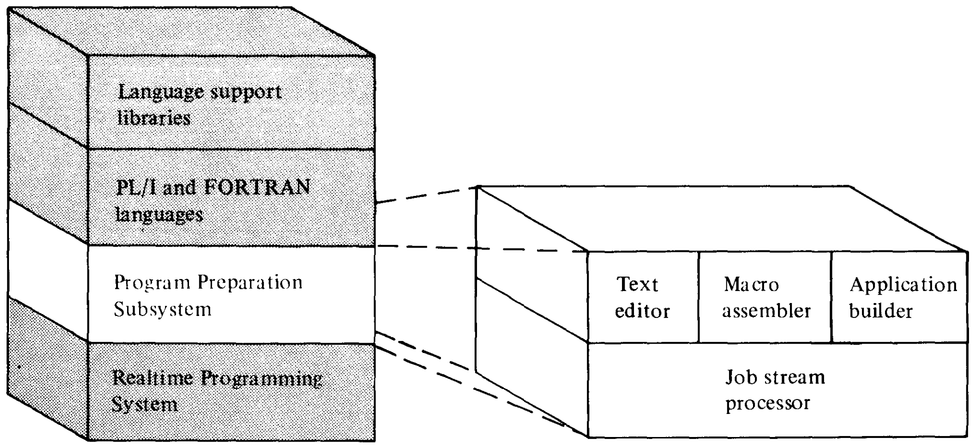

Language support libraries
PL/I and FORTRAN languages
Program Preparation Subsystem
Realtime Programming System
Text editor
Macro assembler
Application builder
Job stream processor

Figure 1-1. The IBM Series/1 software system

The subsystem programs can run concurrently with realtime applications under the realtime supervisor or, in the absence of realtime applications, can run under the realtime supervisor in a simple batch environment. Batch processing provides a convenient method of invoking and executing programs, with little or no operator intervention required.

The Subsystem Components

Each of the components in the Program Preparation Subsystem performs functions that allow you to prepare programs for execution—in the Realtime Programming System execution environment or in an environment you provide. The following section introduces the features and functions of each of the components to give you some background information about the component before you read the individual chapters describing each component. It covers the:

- Job stream processor.
- Text editor.
- Macro assembler.
- Application builder.

The Job Stream Processor

The job stream processor, which serves as the interface to the system resources, provides a simple set of control statements for invoking the programs and for defining data sets they use. These control statements and related data make up a job input stream composed of *jobs* and *steps*. Upon reading the control statements (such as JOB, EXEC, and PARM) the job stream processor analyzes the parameters you supply and processes your requests for executing programs. It also manages the data sets and devices used by the programs and handles the automatic job-to-job, step-to-step transition during job input stream processing.

The job stream processor provides the facility to:

- Specify your work requests to the system, using a simple control language.
- Process a job input stream of work requests, the source of which may be an operator station or a previously-created data set.
- Start the input stream from one device and redirect it to another device. For example, you could start the primary input stream from an operator station, then switch to a secondary input stream on a disk data set.
- Use data set definition (DSD) statements to define data sets and devices required by the subsystem programs or your own programs.
- Specify a group of data set definitions to be in effect at various levels of input stream processing—session level, job level or step level.
- Predefine lists containing all of the data set definitions (DSDs) required for executing particular jobs. These *environment lists* can be prepared and stored on the system. Jobs that require them can then refer to the appropriate environment list, and the job stream processor will process the DSDs as if you had entered them in the input stream.
- Use special control statements —ASMGO, FORTGO, or PL1GO—to initiate a "compile, load, and go" sequence. This lets you translate, build, and execute a program with a minimum of effort.
- Run the Realtime Programming System utilities under the Program Preparation Subsystem.

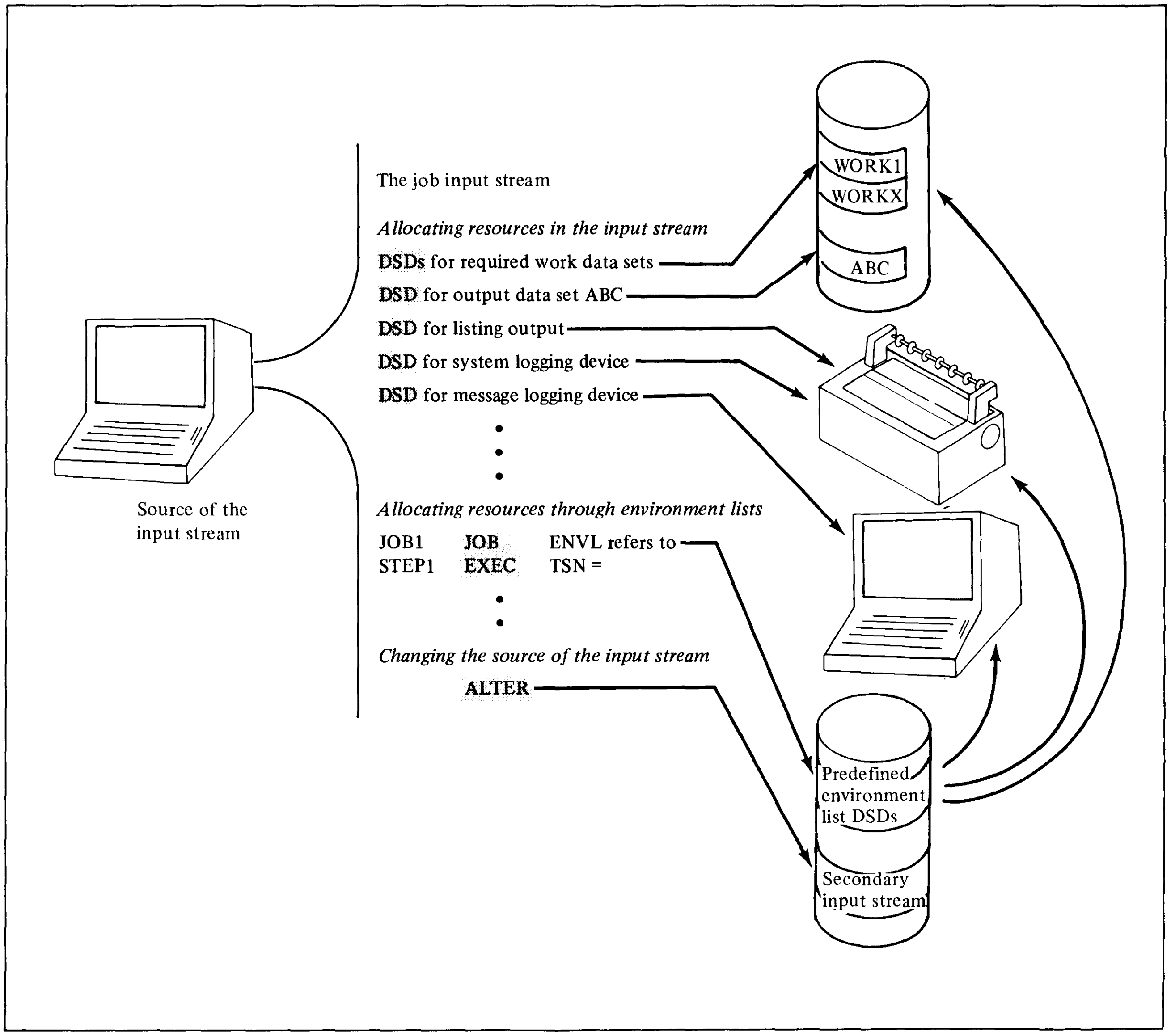

The job input stream
Allocating resources in the input stream
DSDs for required work data sets
DSD for output data set ABC
DSD for listing output
DSD for system logging device
DSD for message logging device
Allocating resources through environment lists
JOB1 **JOB** ENVL refers to
STEP1 **EXEC** TSN =
Changing the source of the input stream
ALTER
Source of the input stream
WORK1
WORKX
ABC
Predefined environment list DSDs
Secondary input stream

Figure 1-2. Using the job stream processor facilities

The Text Editor

The text editor allows you to create, modify, list, and save text modules. These text modules can consist of previously-prepared job input streams, source statements, or input data to a batch program. You can use the editor in an interactive mode, entering text and commands from an operator station, or you can use it in a noninteractive mode by defining a disk or diskette data set as the input source.

Using the editor, you can create a new text module or retrieve and update an existing text module. The actual editing takes place in two work data sets, referred to as the *editor workspace.* The output from the text editor is a newly-created or updated text module that can be saved on a disk or diskette data set and later used as input to a language translator or one of your own application programs.

The editor has 15 easily-learned commands that provide a variety of editing capabilities. You can:

- Replace text data within a field in one or more lines. (CF)
- Clear the editor workspace to prepare for a new editing session. (CL)
- Copy lines of text from one area of the editor workspace to another. (CO)
- Display the current status of the editing environment. (CS)
- Change a character string in one or more lines. (CT)
- Delete one or more lines in the editor workspace. (DE)
- End the editing session. (END)
- Search for text in the editor workspace and print each line containing the text. (FI)
- Get text from a data set and place it in the editor workspace. (GE)
- Insert new lines of text into the editor workspace. (IN)
- Set a line display range and set a line length. (LF)
- List one or more text lines of the editor workspace. (LI)
- Move one or more lines from one area in the editor workspace to another. (MO)
- Save text lines of the editor workspace to a data set. (SA)
- Set tabs to be used during an editing session. (TA)

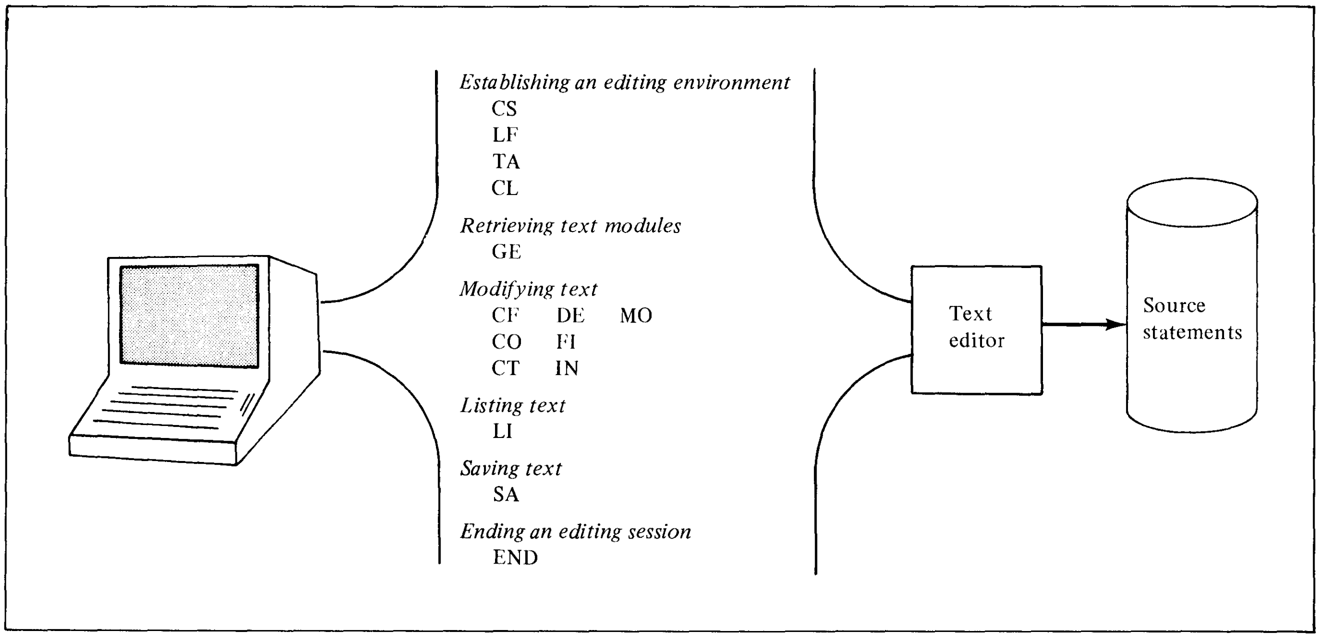


Figure 1-3. Program preparation—the text editor (part 1 of 3)

The Macro Assembler

Once source statements have been created, the next step in the program preparation sequence is a language translator. The macro assembler is a powerful language translator that provides:

- A function-oriented assembler language for specifying machine instructions.
- A flexible macro language facility.
- Conditional assembly capability within macros.
- Sectional assembly capability.
- Assembler options for data definition and listing control.
- Relocatable object module output.
- Listing output that can include the source program and object text, external symbol dictionary, relocation dictionary, cross-reference table, error messages, and statistics.

The main purpose of the assembler program is to process source statements and create one or more relocatable object modules. These object modules consist of machine instructions and information that will be used by the application builder in its processing.

Note. Using the PL/I and FORTRAN IV program products in program preparation is described in the *IBM Series/1 PL/I: Introduction,* GC34-0084 and the *IBM Series/1 FORTRAN IV: Introduction,* GC34-0132.

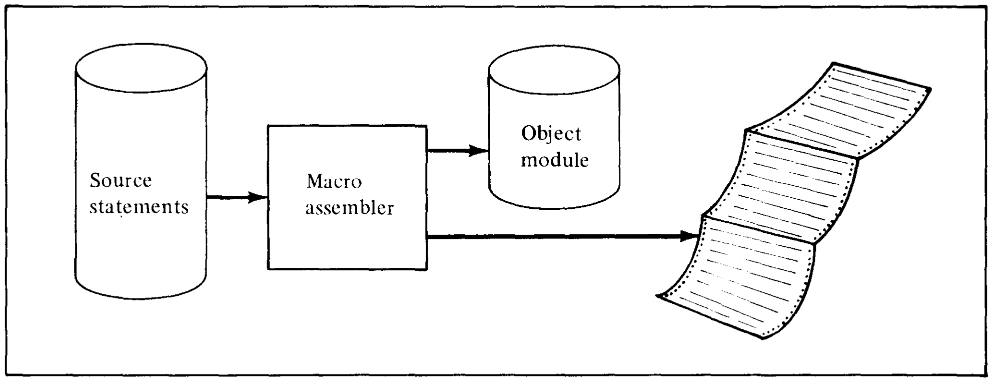


Figure 1-4. Program preparation—the macro assembler (part 2 of 3)

The Application Builder

The application builder provides a variety of services that can be used in creating an application program for execution in the Realtime Programming System environment or in an environment you provide. How you use these services will depend on the execution environment for which you are preparing the application program.

To execute in the realtime environment, an application program must be in the form of a *task set*. A task set is a planned program structure that consists of one or more modules grouped together and combined with various tables and control blocks (a control module) required by the realtime supervisor. To execute in an environment other than the realtime environment, an application program must be a load module in absolute format.

To create an application program, in the form of a task set, to execute in the Realtime Programming System environment, the application builder:

- Processes control statements that allow you to select the options and data set environment to perform the functions required.
- Assigns storage addresses and resolves external references to create a composite module in relocatable format, which will later become part of a task set.
- Produces composite modules that may be structured in a single segment (simple structure) or multiple segments (overlay structure).
- Provides a recycling capability to process multiple sets of control statements, where each set of statements creates a composite module.
- Can include modules automatically from a program library through an *autocall facility*.
- Creates a *control module*, which specifies the control blocks required by the Realtime Programming System, at application build time rather than at execution time. (This control module resides in your task set at execution time.)
- Creates a *prebind module*, which allows you to specify the resources to be prebound to a task set when it is installed rather than at execution time. This prebinding of resources will allow your task set to begin executing faster.

To create an absolute load module that is executable in an environment you provide, the application builder:

- Processes the control statements you specify to identify the contents, structure, and name of the absolute load module.
- Processes object modules to create absolute load modules that consist of a single segment. Addresses are assigned relative to the specified origin for that module, as well as having external references between the object modules resolved.

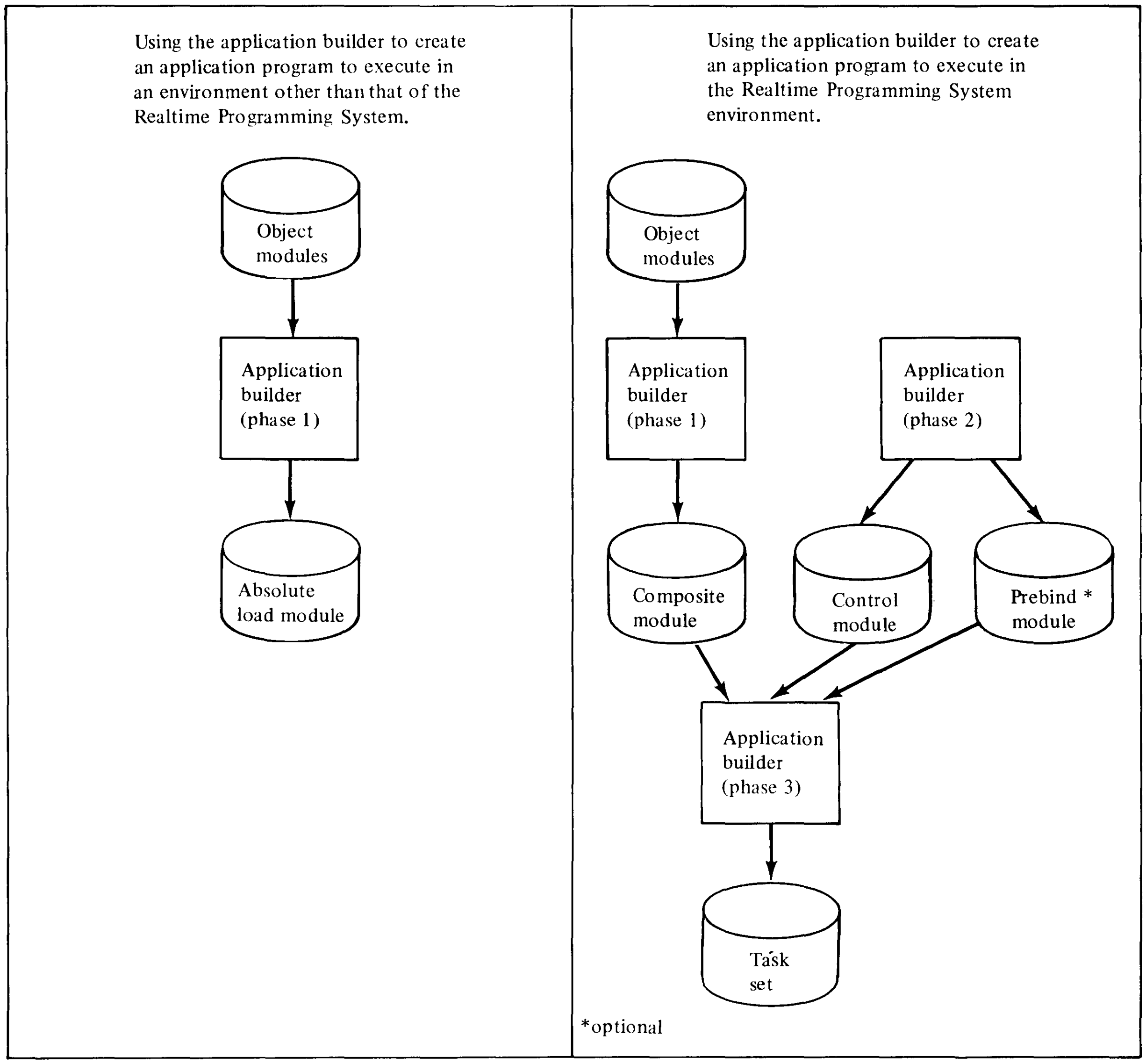

Using the application builder to create an application program to execute in an environment other than that of the Realtime Programming System.
Object modules
Application builder (phase 1)
Absolute load module
Using the application builder to create an application program to execute in the Realtime Programming System environment.
Object modules
Application builder (phase 1)
Application builder (phase 2)
Composite module
Control module
Prebind * module
Application builder (phase 3)
Task set
*optional

Figure 1-5. Program preparation—the application builder (part 3 of 3)

Using the Subsystem Components in Program Preparation

A major use of the subsystem components is preparing programs for execution under the realtime supervisor or your own supervisor.

The basic steps involved in creating an application task set are:

1. Use the job stream processor to manage the program preparation environment.
2. Use the text editor (or IBM 3741 Programmable Work Station) to create the source code.
3. Use the macro assembler or a compiler to translate the source code and create object modules.
4. Use the application builder to create composite modules or absolute load modules. (If you are creating an absolute load module to execute under your own supervisor, you can skip steps 5 and 6.)
5. Use the application builder to create a control module and (optionally) a prebind module.
6. Use the application builder to create an application task set that can be executed in the Realtime Programming System environment.

Figure 1-6 shows the sequence of steps in creating an application task set.

Typical Uses of the Program Preparation Subsystem

The facilities of the Program Preparation Subsystem meet the requirements of a wide range of users. Typical uses of these facilities include:

- Preparing application task sets to run under the realtime supervisor in a realtime environment.
- Creating your own supervisor and preparing programs to run under that supervisor.
- Using the subsystem programs in a batch environment.
- Using the compile, load and go capability for problem-solving purposes.

Using the Subsystem Programs in the Realtime Environment

Typically, you would create programs and data using the text editor, then store them on permanent data sets. The source statements are compiled or assembled to create object modules, which must undergo application builder processing to build your realtime application.

To execute under the realtime supervisor, an application must be in the form of a task set. Phases 1, 2, and 3 of application builder processing are required to create task sets that are executable in the realtime environment. Figure 1-6 illustrates this processing.

The subsystem programs may execute concurrently with your realtime applications.

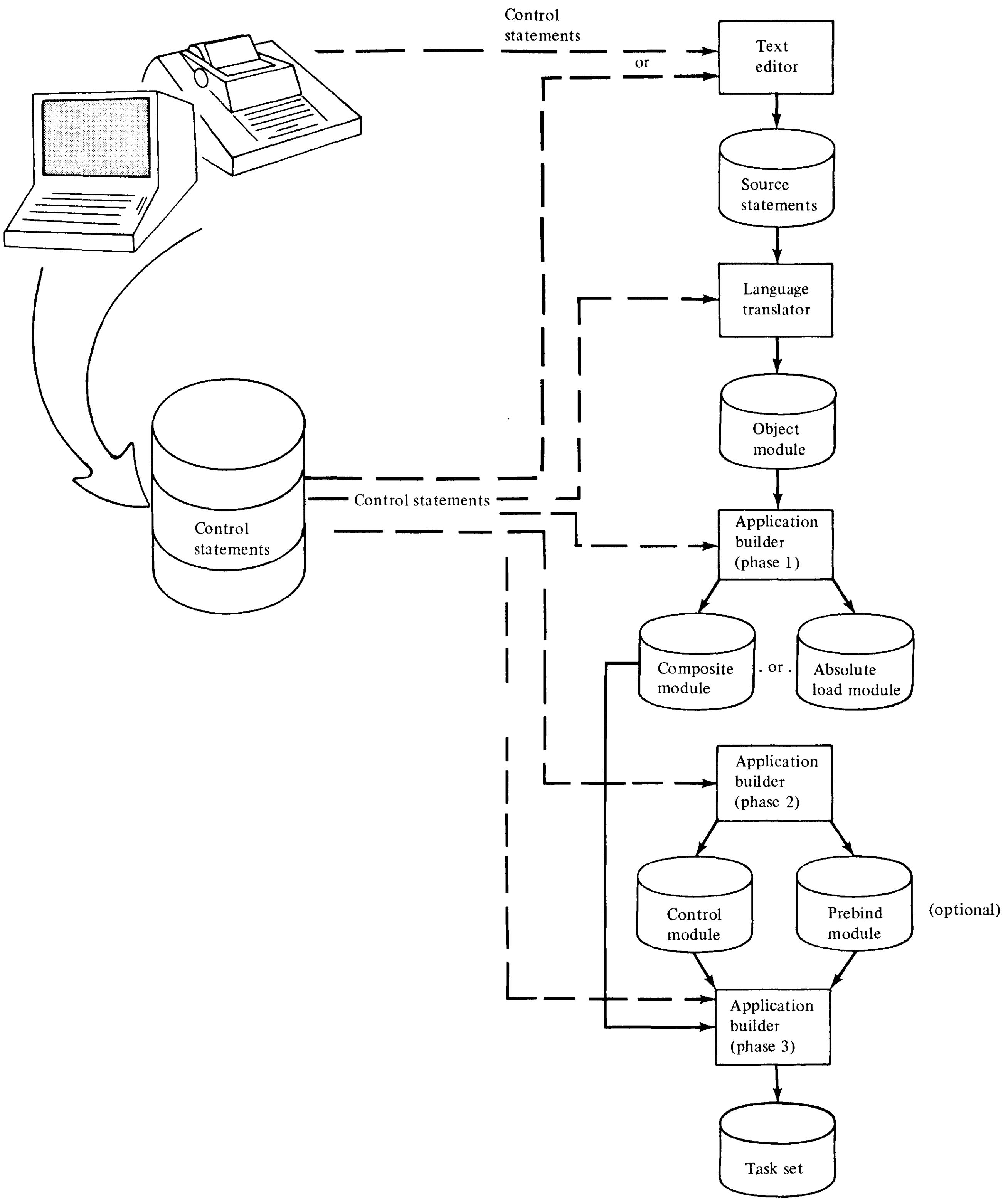


Figure 1-6. Program preparation sequence

Preparing Programs for Execution Under Your Own Supervisor

In this case, you prepare programs in much the same manner as you would for the Realtime Programming System, with one exception. Instead of creating programs, in the form of task sets, to execute under the realtime supervisor, you create absolute load modules to execute independently or under your own supervisor.

Phase 1 of application builder processing can create these absolute load modules. However, none of the Realtime Programming System support functions can be requested, and you must provide a means of loading the programs into storage and passing control to them. The Disk IPL Bootstrap Loader, which is described in the *IBM Series/1 Stand-Alone Utilities User's Guide*, GC34-0070, can be used for loading a program and passing control to it.

Figure 1-7 illustrates this processing.

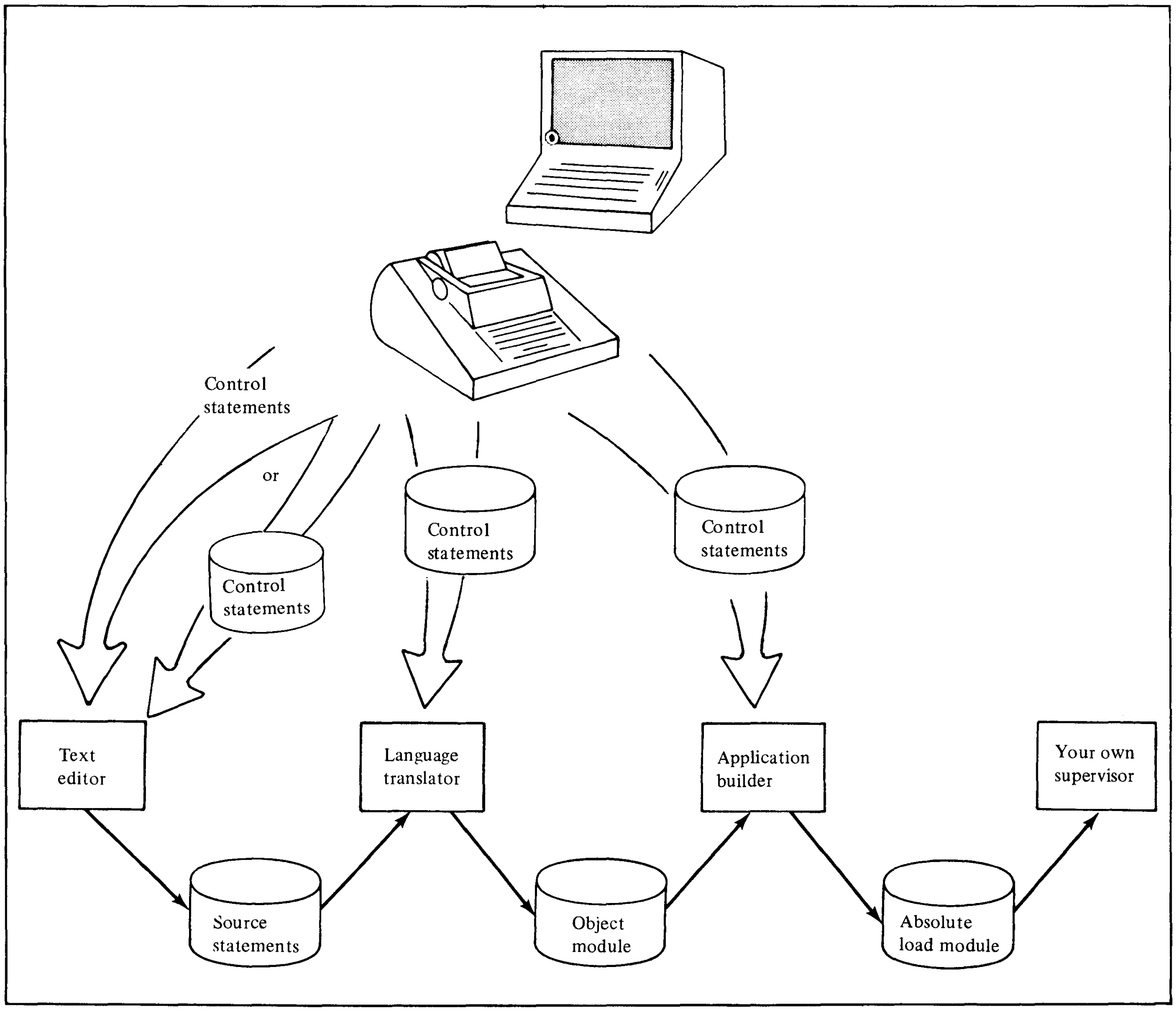


Figure 1-7. Preparing programs for execution under your own supervisor

Using the Subsystem Programs in a Batch Environment

You may use application programs that you have created to run in a batch environment. Application program input, stored in data sets, can be prepared or updated by using the text editor. The data set can be passed to the application program through the job stream processor.

For example, you may have a payroll program residing on a data set. Periodically you would use the text editor to enter such data as overtime for the last period of time. This data and the appropriate job stream processor control statements would then be entered in the job input stream for processing by your payroll program.

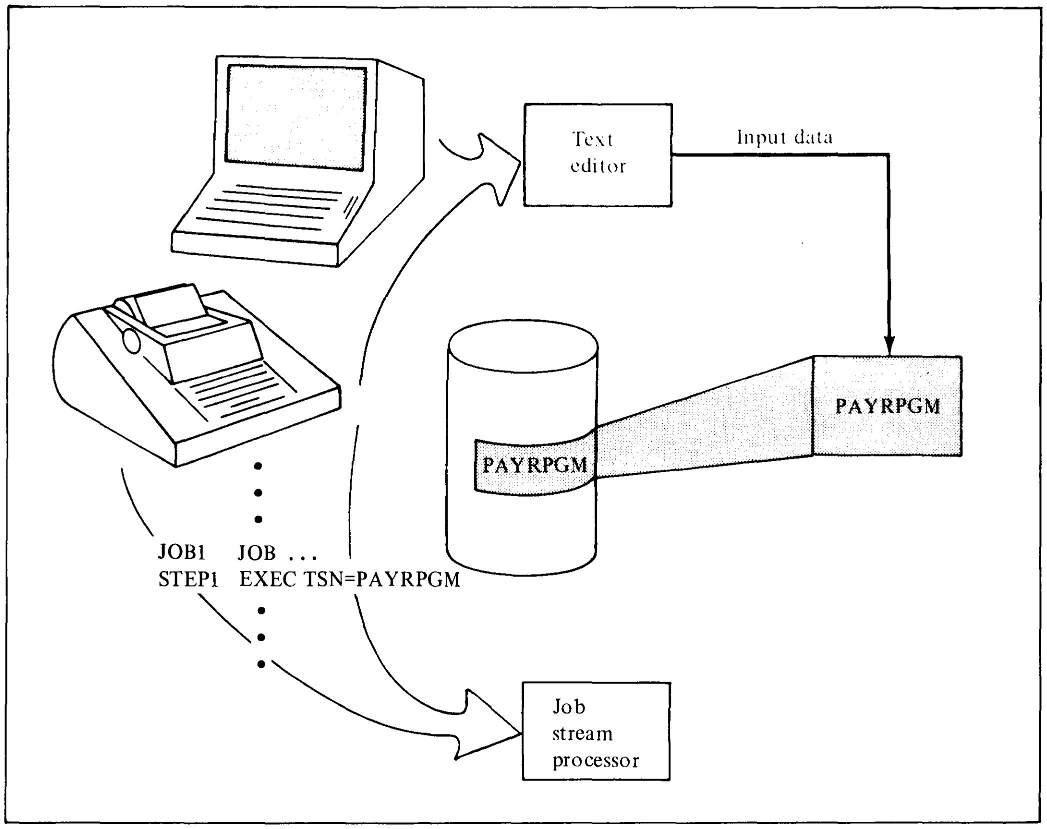


Figure 1-8. Using the subsystem programs in a batch environment

Problem Solving with the Compile, Load, and Go Facility

The Program Preparation Subsystem provides a simple method for performing problem solving activities through the compile, load, and go facility. This allows you to compile or assemble a program, build a task set, and execute that task set by specifying a few simple control statements. The DSD statements identifying the required data sets and devices are defined in advance in an *environment list*. (Compile, load, and go is described in the job stream processor chapter of this manual.)

Typically, you would write a small program in a high level language, enter this program through the text editor, and then use the appropriate statement (ASMGO, PL1GO, or FORTGO) to perform a compile, load, and go. Usually you would only be interested in the printed output and would not need to save object data or to specify intermediate input, output, or disk space requirements. Figure 1-9 illustrates the use of the compile, load, and go facility. CLGLST refers to a previously-prepared environment list.

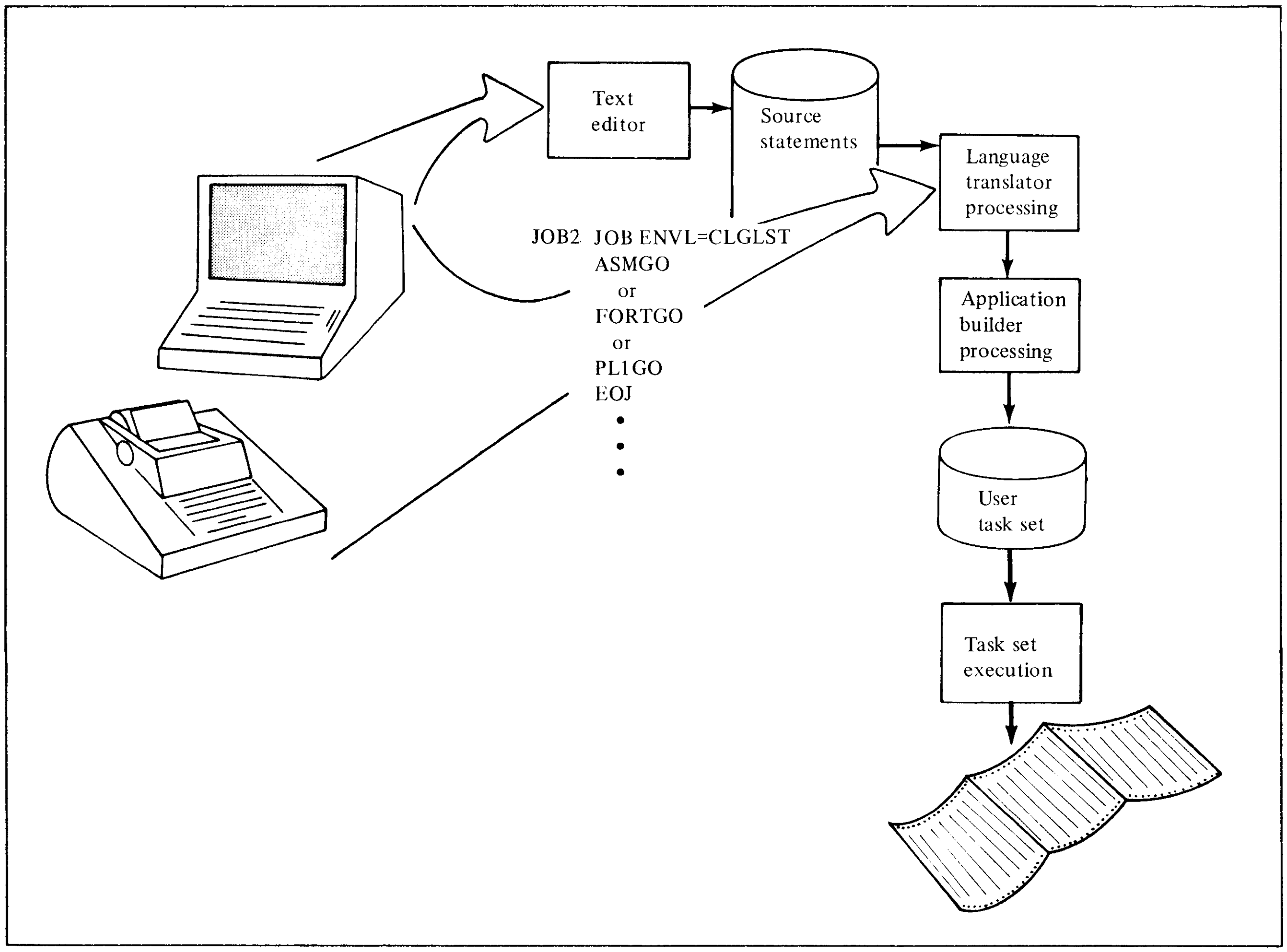


Figure 1-9. Example of using the compile, load, and go facility

Invoking and Executing the Subsystem Programs

The subsystem programs run under control of the Realtime Programming System as realtime applications in a predefined, fixed partition. The text editor, macro assembler and application builder are all task sets that are invoked through the job stream processor, which is also a task set.

To begin a batch session, you must invoke the job stream processor by entering a system command—*TSET STR,CPJ, ptn, qprty*—at the operator station. This command starts the job stream processor task set (CPJ), specifies the number of the partition in which the job stream processor is to execute (ptn), and establishes the queuing priority of the job stream processor (qprty). The job stream processor is then loaded and starts processing the input stream. The input stream, which consists of control statements and related input data, is logically subdivided into jobs and steps. The step is the basic unit of work in which you can specify the EXEC control statement to invoke a batch program.

Figure 1-10 illustrates this sequence.

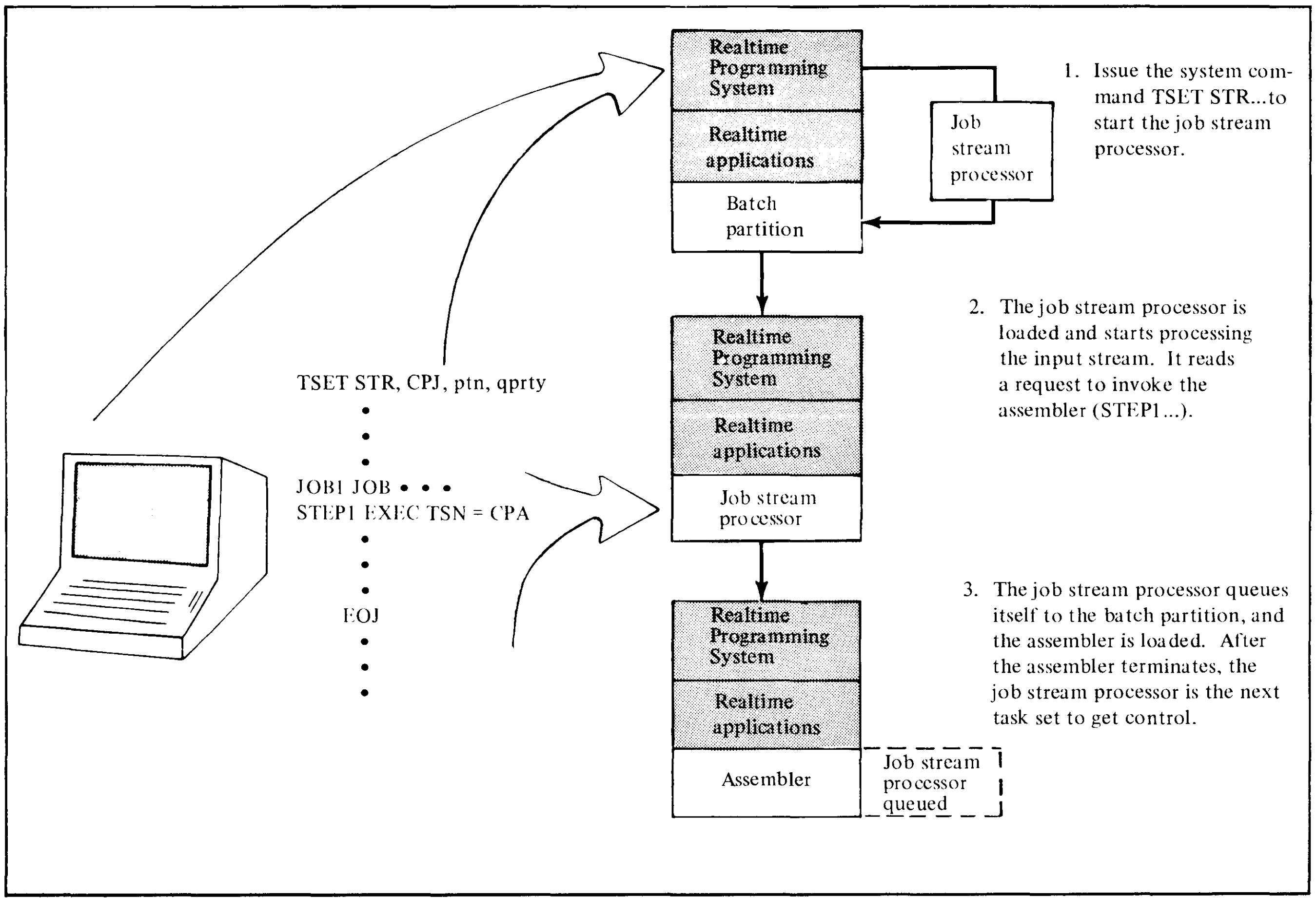

Realtime Programming System
Realtime applications
Batch partition
Job stream processor
1. Issue the system command TSET STR...to start the job stream processor.
2. The job stream processor is loaded and starts processing the input stream. It reads a request to invoke the assembler (STEP1...).
3. The job stream processor queues itself to the batch partition, and the assembler is loaded. After the assembler terminates, the job stream processor is the next task set to get control.
Assembler
Job stream processor queued
TSET STR, CPJ, ptn, qprty
JOB1 JOB • • •
STEP1 EXEC TSN = CPA
EOJ

Figure 1-10. Invoking and executing the subsystem programs

Note. The batch partition is not reserved for the subsystem programs. Realtime applications use it either when the subsystem programs are inactive or by preempting the partition during a batch processing session.

Data Sets and Devices Used by the Subsystem Programs

In the input stream there can be a variety of requests to execute programs—the text editor, the assembler, the application builder, PL/I, FORTRAN IV, or your own batch programs. Each of these programs uses certain data sets and devices to make up its operating environment. You can specify these resources to the program through data set definition (DSD) statements. DSD statements are used to establish a connection between a data set or device and a DSD name used in a program.

You can use DSD statements to create new data sets, identify existing data sets or delete data sets that are no longer needed. The length of time the DSD is in effect depends on where it is located in the input stream—in a step, in a job, or at the beginning of a batch session.

The Required Data Set Definitions

Certain DSDs are established when the Program Preparation Subsystem is installed to make required data sets available to the job stream processor when you start a batch processing session.

To simplify the installation process, default DSDs are supplied for the required data sets at installation time. You can change any of these DSDs if they do not suit your needs.

The required data set information and DSD names are summarized here. The data sets and devices described are those that are required by the job stream processor and, therefore, by all components invoked through the job stream processor. For information about the required data sets and devices that are unique to each component—the text editor, macro assembler, or application builder—refer to the chapter describing that component.

INSTREAM

The INSTREAM DSD defines the input stream source. Unless you specify another input source in the PARM field of the task set start command, the job stream processor always reads the input stream from the device defined by INSTREAM.

The default INSTREAM device is the operator station.

JSPWORK

JSPWORK defines a disk data set in which the job stream processor saves its control blocks when it relinquishes the batch partition to another task set.

MSGLOG

MSGLOG defines the messages log device or data set that the job stream processor and text editor use for writing prompts and messages when in the interactive mode. The device assigned to MSGLOG should be the same as the INSTREAM device.

The default MSGLOG device is the operator station.

PRINT

The device specified by the PRINT DSD is used by all of the subsystem programs for listing various types of printed output that you need in hard copy form.

(For example, the assembler and application builder print their listings and maps at the device defined by PRINT.)

The default PRINT device established at installation time is the printer.

SPOOL

The disk volume defined by the SPOOL DSD is used by the job stream processor to handle your inline input data for the batch programs. When the job stream processor is processing the input stream and encounters your input data for a batch program, it automatically creates a data set on the SPOOL volume and transfers the inline data to that data set before the batch program is invoked.

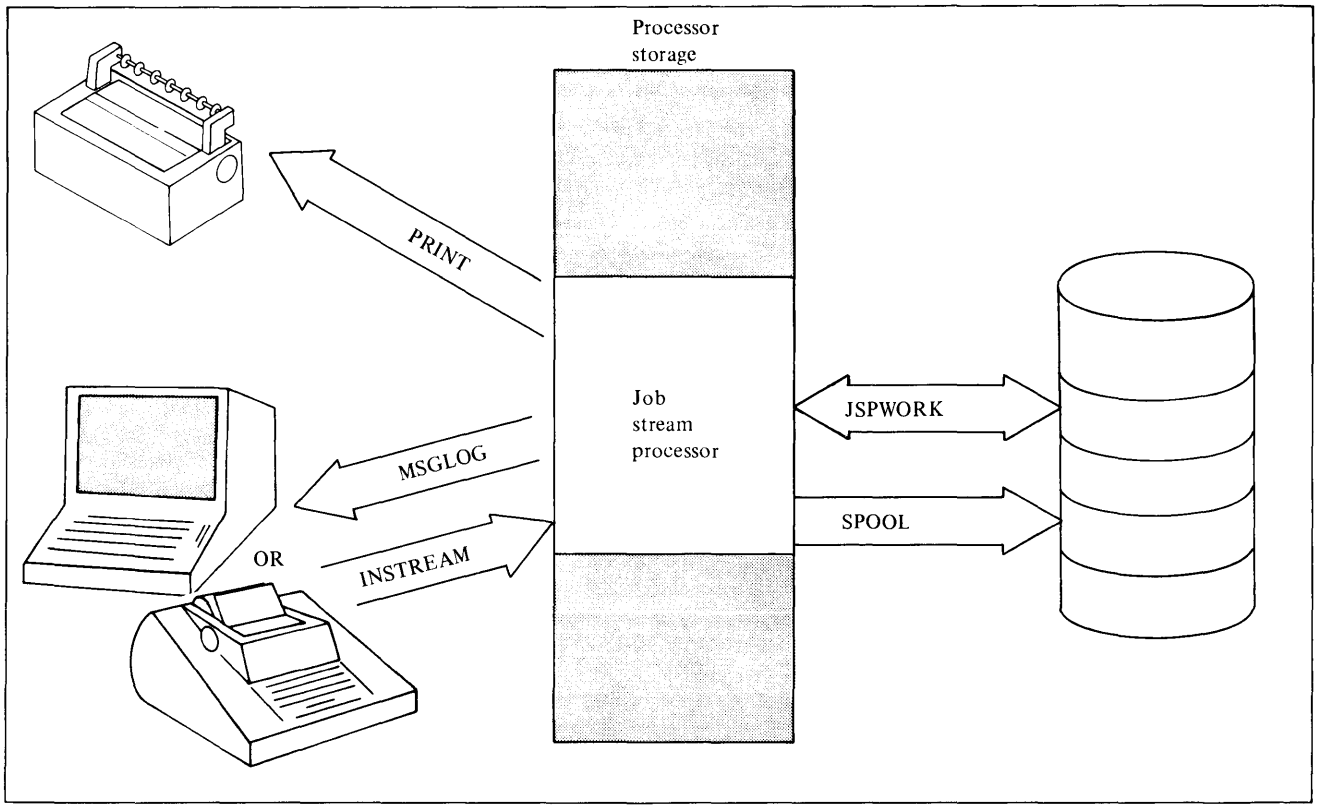


Figure 1-11. Default DSDs for required data sets and devices common to all subsystem components

Work Data Sets

The text editor, macro assembler, and application builder all require work data sets in order to perform their processing. Each of these components requires a different size and number of work data sets. You can specify the space required for a work data set by:

- Using individual DSD statements to indicate a specific volume and a specific amount of space for the WORK1—WORK*x* data sets.
- Using the WORKVOL DSD to define space for all the work data sets (WORK1—WORK*x*) and the volume on which the space is to be allocated.

Note. The DSD statements for WORK1-WORK*x* must always be specified for the assembler and for the application builder, whether or not you use the WORKVOL DSD. The operands you specify on these statements will vary, depending on the use of the WORKVOL DSD. The text editor only requires that you specify the WORKVOL DSD.

To maximize workspace efficiency, you can use the WORKVOL DSD to define a volume to which all available space will be allocated. Each component can then dynamically allocate this space between the number of the work data sets it needs and in the proportion it needs.

For example, suppose you were to specify the WORKVOL DSD (to create a temporary work volume) and dummy work data sets—WORK1, WORK2, and WORK3—for assembler processing. The assembler, which requires three work data sets, would allocate the space among WORK1, WORK2, and WORK3. Work data set requirements for each component are summarized in the individual chapters describing the component.

Environment Lists

Once the Program Preparation Subsystem is installed, you can also specify data sets to contain lists of DSDs that the job stream processor will use to create the environments for various batch programs. These *environment lists* can contain all the DSDs required to execute a job or step.

Many jobs can effectively use environment lists. For example, if you use the compile, load, and go facility, you should have environment lists prepared for the compile, load and go jobs. You might also want to prepare environment lists for jobs that use the program preparation facilities.

The chapters covering the text editor, macro assembler, and application builder give you more specific information about the data sets and devices used by each of these components. This information should help you to plan the data set and device assignments most efficient for your system and to set up the appropriate environment lists.

Required Hardware

The minimum hardware configuration for the Program Preparation Subsystem is the same as is required for the Realtime Programming System:

Required hardware	*Product name*
Processor	One IBM 4953 Processor or IBM 4955 Processor with at least 48K bytes of processor storage.
Disk/diskette	One (or more) IBM 4962 Model 2 or Model 2F Disk Storage Unit (combination disk/diskette unit). *or* One (or more) IBM 4962 Model 1 or Model 1F Disk Storage Unit *and* One (or more) IBM 4964 Diskette Units
Printer	One (or more) IBM 4973 Line Printers or IBM 4974 Printers
Operator station	One (or more) IBM 4979 Display Stations *or* Teletype* Model ASR 33/35 or an ASCII equivalent device that can be used an an operator station and is attached to the system through the Teletypewriter Adapter Feature #7850.

*Trademark of the Teletype Corporatioon

Timers may optionally be used with the subsystem programs.

For information about sensor I/O and communications support, which are not supported by the Program Preparation Subsystem, refer to the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

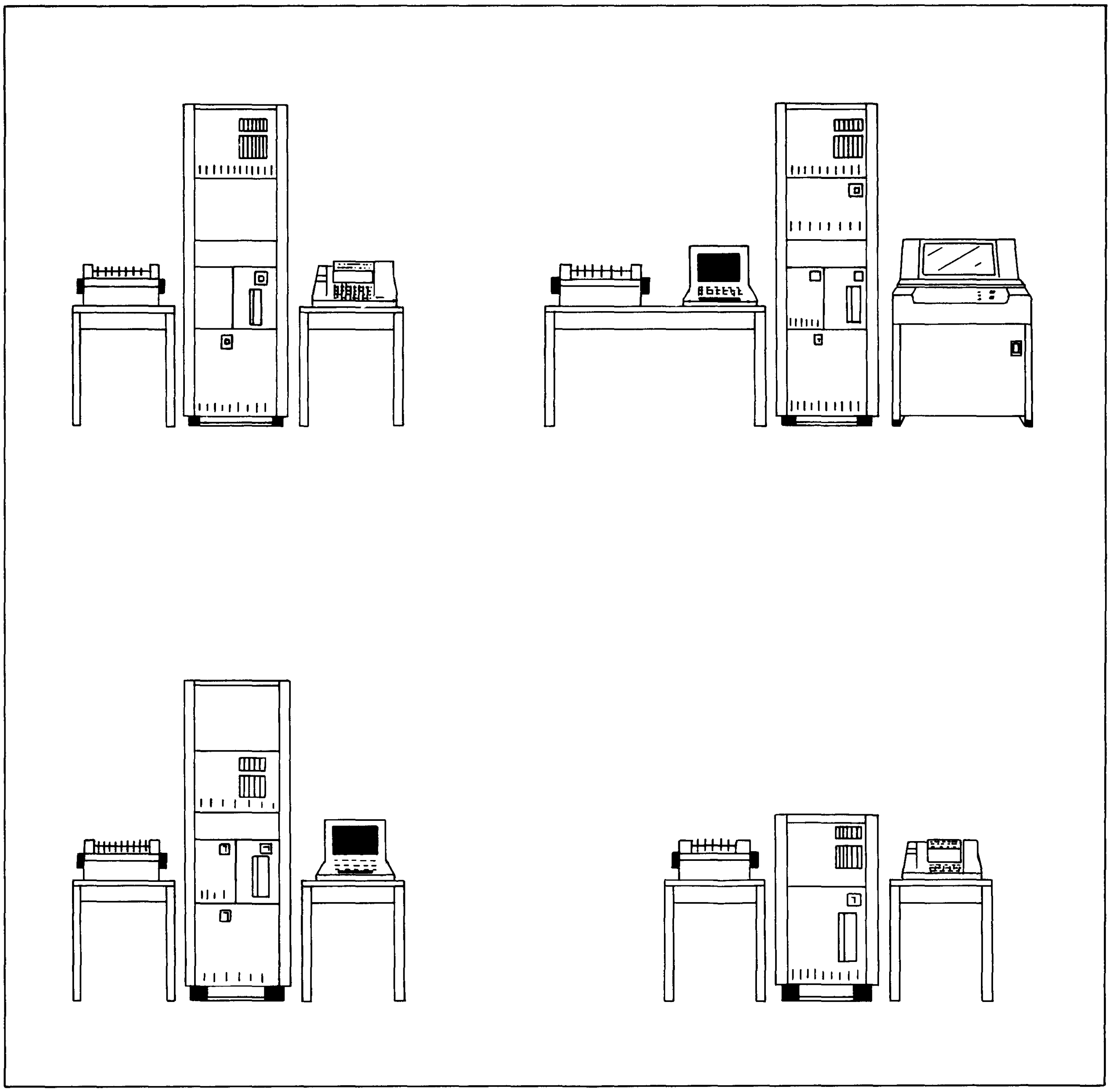


Figure 1-12. Sample hardware configurations

Installing the Program Preparation Subsystem

Once you have determined your data set and device requirements and the required hardware configuration, you can prepare to install your Program Preparation Subsystem. Before the subsystem programs can be installed, you must have already performed a system generation for the Realtime Programming System. The subsystem programs are installed in a stand-alone environment, and the installation process requires 48KB of processor storage.

The installation package you receive from IBM contains diskettes that contain the information required for converting the IBM-supplied code into task sets designed to run in the batch partition.

IBM also supplies a *generation program* that prompts you to supply information about the system to be built. To simplify the installation process, defaults are provided for all of the required DSDs. If you accept these defaults, the system can be installed as is. If you do not want to accept the defaults, you can make changes to the DSDs through your responses to the generation program prompts.

After you have supplied the information requested by the generation program, a job input stream will be generated. When this job input stream is executed, you will have a Program Preparation Subsystem tailored to your requirements. The generation program can also perform a system verification to ensure that your system is properly installed. System verification can only be performed against a complete Program Preparation Subsystem.

Although the Program Preparation Subsystem requires only a 16KB partition for execution of batch programs, you may want the partition to be larger for program preparation activity. This would make more efficient use of the system because the text editor, assembler, and application builder execute faster in a larger partition.

The *IBM Series/1 Program Preparation Subsystem: Batch User's Guide* will give a step-by-step description of installing and verifying the Program Preparation Subsystem.

Chapter 2. The Job Stream Processor

The job stream processor is the subsystem component that reads, analyzes, and processes the job input stream. The job input stream is made up of your requests for executing programs along with related information, such as data set definitions. The job stream processor provides a simple set of control statements for specifying these requests.

To start a batch session, you would invoke the job stream processor with a task set start command. Once started, the job stream processor will read the input stream from the device you designated as the input stream source. The input stream source could be an operator station or a disk or diskette data set. This can be specified at installation time or when the job stream processor is started. The job stream processor reads and processes the control statements and data that make up the input stream until the end of the input stream is reached, *or* a task set stop command is issued for the job stream processor, *or* a severe error occurs.

Detailed information about the job stream processor will be contained in the *IBM Series/1 Program Preparation Subsystem: Batch User's Guide.*

The Job Input Stream

The job input stream is logically divided into independent units of work called jobs. Each job may be made up of steps, which are basic units of work for the subsystem.

Jobs and Steps

Jobs are self-contained elements and, as such, have no relation to other jobs in the input stream. The input stream can consist of many jobs, which are processed one after another in the sequence that they appear in the input stream. If a job terminates because of an error, the jobs that follow it will not be affected, and the job stream processor will automatically continue with the next job in the input stream.

Each job usually contains one or more related steps. A step is the basic unit of the input stream that allows you to invoke a batch program. A step is identified by the EXEC statement, which specifies the batch program to be executed. This example shows a request to execute a task set called PGM1.

```
EXEC TSN=PGM1
```

When there is more than one step in a job, execution of a given step depends on successful execution of the previous step. The job stream processor can determine whether or not a batch program was successfully executed by the *return code* it receives from the batch program. Return codes are used by all of the subsystem programs to indicate the results of their processing. If a step is not successfully executed, the job stream processor does not attempt to execute the steps that follow, but goes on to the next job in the input stream.

Within a job or step, you can also include data set definition (DSD) statements that define data sets and devices to be used for that job or step. This is discussed in the section that follows.

Figure 2-1 shows an example of the structure of an input stream.

```
• • •   DSD    • • •
• • •   DSD    • • •
JOB1    JOB    • • •
• • •   DSD    • • •
STEP1   EXEC   TSN=PGM1
        EOS
STEP2   EXEC   TSN=PGM2
• • •   DSD    • • •
• • •   DSD    • • •
        EOS
        EOJ
JOB2    JOB    • • •
STEPX   EXEC   TSN=PGMX
        EOS
        EOJ
         •
         •
         •
```

Figure 2-1. Sample input stream

Data Set Definition Statements

Before a data set can be used by a batch program, it must be defined to the system. To do this, you can use DSD statements. (You can also use the Realtime Programming System's BLDDSD macro, DEFDSD macro, or DEFINE utility to establish the connection between a DSD name and the associated data set or device.) Once the definition has been established, a program can access the data set by using the DSD name associated with it.

You can use DSD statements to establish new data sets or to identify existing data sets. Both permanent and temporary data sets can be created. Permanent data sets can also be deleted with a DSD statement. In addition to defining data sets, DSDs can be used to identify a device to be used by a program.

If you enter DSDs at the start of a batch session, these DSDs remain in effect for the entire session unless you override them with the appropriate DSD statements at the job or step level. To override an existing data set or device definition, enter a new DSD statement with the same name as the DSD name of the definition that you want to override. The new DSD statement overrides all parameters on the previously-established DSD of the same name.

DSDs placed within a job are defined for the duration of that job; DSDs placed within a step are defined for the duration of that step. At the end of the associated job or step, the definition is deleted although the data set itself may be permanent.

DSD Environment

The DSDs in effect at any given time in the input stream make up the *DSD environment*. At installation time, required DSDs are established when you either accept the default DSDs that IBM supplies or modify them to suit your needs.

After the program preparation subsystem has been installed, you can prepare lists of DSDs to create environments used by the subsystem programs or your own batch programs. These *environment lists* are very useful because they can reduce the number of statements that must be entered to execute a particular job. On the JOB statement, you simply include the ENVL parameter to refer to the desired environment list. An example of this is shown in the following section.

Compile, Load, and Go Statements

There are special control statements you can use to perform a compile, load, and go with a minimum of effort and knowledge of the Program Preparation Subsystem. These special statements and their functions are:

- *ASMGO*—causes execution of the assembler program, application builder, and the task set created by the application builder.
- *FORTGO*—causes execution of the FORTRAN IV compiler, application builder, and the task set created by the application builder.
- *PL1GO*—causes the execution of the PL/I compiler, application builder, and the task set created by the application builder.

When you specify one of these statements, a series of steps is initiated to translate, build, and execute a program. By using these statements, you let data set definitions default to data sets that have been previously defined in environment lists. All you have to do is specify the statement (ASMGO, FORTGO, or PL1GO) and include the ENVL parameter on the JOB statement and all the necessary steps are performed for you. The *IBM Series/1 Program Preparation Subsystem: Batch User's Guide* will provide the information you need in order to prepare for compile, load, and go processing.

Figure 2-2 illustrates the use of the ASMGO statement.
Here, the ENVL parameter indicates that ALIST refers to a list of DSDs for this compile, load, and go job.

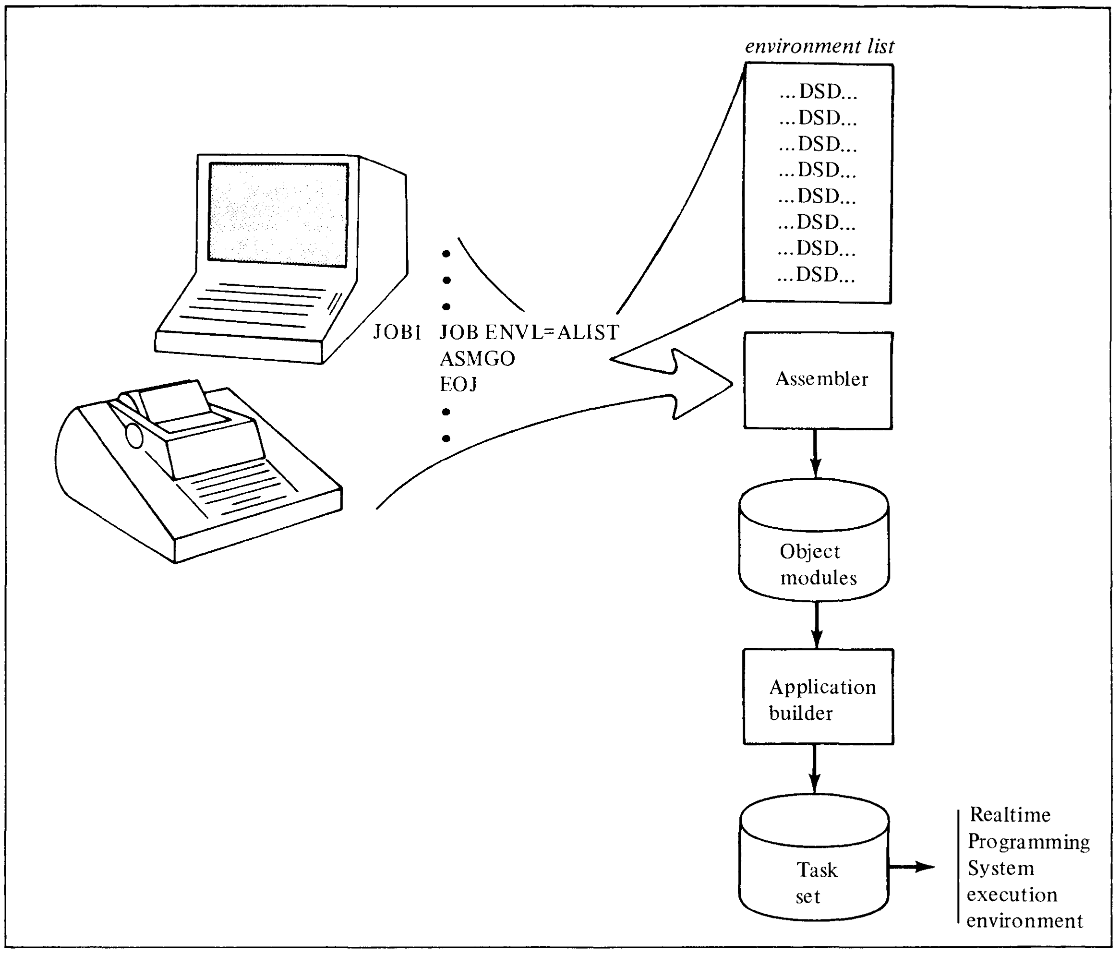


Figure 2-2. Using the ASMGO statement

Input Data

Along with the control statements in the input stream, you may also include input data for the subsystem programs or your own batch programs. Before the program is invoked, this inline data will be spooled out to a disk data set (which you must have preallocated for that purpose), then passed to the program for use. Spooling is done to preserve input stream integrity.

To include input data in the input stream, specify a DSD statement with an asterisk (*) parameter preceding the data and a slash asterisk (/*) delimiter following the data. Suppose you wanted to include data to be used by the assembler. Figure 2-3 shows how the input data is spooled by the job stream processor, then passed to the assembler.

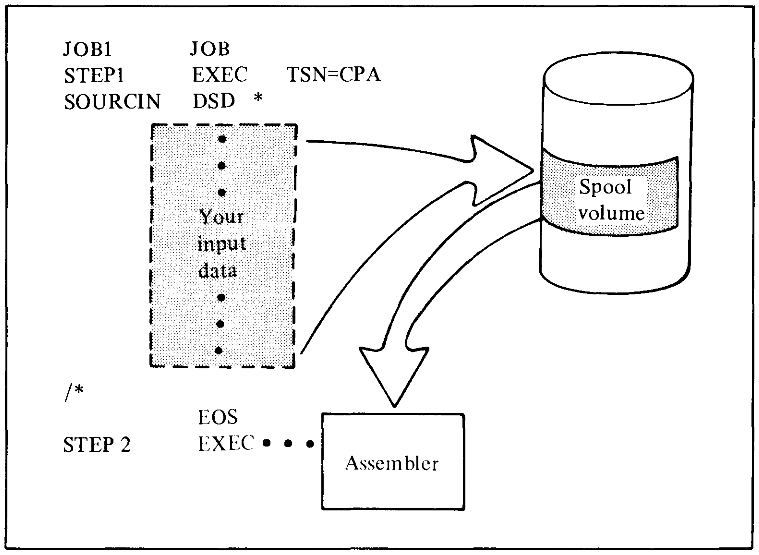


Figure 2-3. Spooling input data

Note. Your input data to the subsystem programs must be defined by specific DSD names—SOURCIN for the assembler, APBIN for the application builder, and CMDIN for the text editor. You cannot assign an interactive device to SOURCIN or APBIN.

Input data for a batch program need not be located in the input stream. It could be on a completely separate data set. In this case, you would define the data set using a DSD statement with the appropriate DSD name (SOURCIN, APBIN, or CMDIN) and the job stream processor would pass the data set to the subsystem program, which then controls that data set.

Sources of the Input Stream

The source of the input stream may be an operator station or a previously-created text library.

From the Operator Station

The *operator station* may be an IBM 4979 Display Station or any ASCII interactive device that can be attached to the processor through the Teletypewriter Adapter Feature. When the input stream is to come from the operator station, the job stream processor control statements and data will be accepted even though the operator station is also being used as the interface to the Realtime Programming System.

From a Text Library

The source of the input stream can be a text library, which may be contained on a disk or diskette data set. In this case, you must have previously created the input stream by using the text editor or the IBM 3741 Programmable Work Station. Assembler and application builder input must come from a text library.

Redirecting the Input Stream

You may, at some time, wish to change the source of the input stream. The job stream processor allows you to have input streams residing on more than one device. You might have one main input stream along with secondary input streams that are only processed periodically. Suppose your main input stream source is an operator station, and you have a secondary input stream on a disk data set. You can start the input stream from the operator station and then use the ALTER statement to redirect the input stream to come from the disk data set.

Figure 2-4 gives an example of redirecting the input stream.

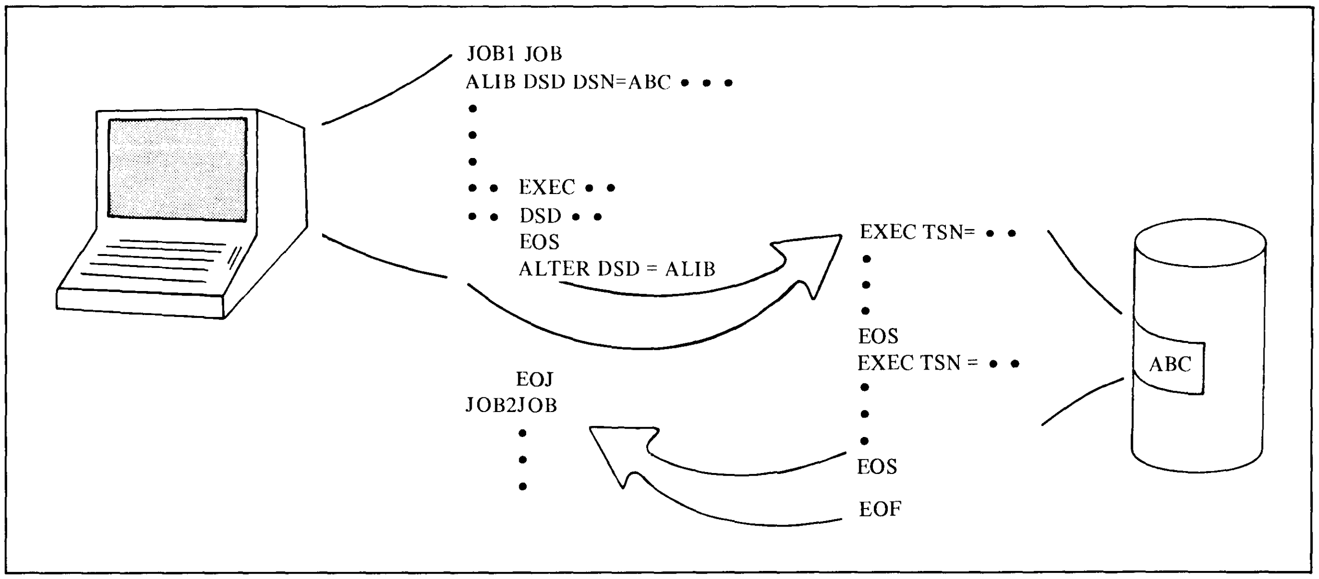


Figure 2-4. Redirecting the input stream

This feature gives you flexibility in creating your input stream because you can execute different jobs or steps within jobs from different locations. This is useful when you have various jobs that you want to execute under a single environment, but they are not all in one location. All you have to do is start the batch session from the operator station, enter the environment, and use ALTER statements to go to each job. This saves you the time of reentering each job individually.

Which Input Stream Source to Use

The input stream source you wish to use—operator station or a previously–created data set—depends on your time and flexibility requirements.

If you want to be able to change the input stream at the last moment, you would use an operator station. This way the input stream does not have to be created until the moment it is ready to be processed. The disadvantage to this is that it can be time-consuming and requires that someone be present at the operator station at all times.

If you anticipate no changes to the input stream, you can create it on a disk or diskette ahead of time. When you start the batch session, you can indicate that this previously-created input stream is the one to be processed— either by defining it as the primary input stream on the task set start command or by using an ALTER statement to switch to it. For standard jobs that are in different locations, you can put them in a text library and use ALTER statements to combine and execute these jobs as needed.

Summary of Job Stream Processor Control Statements

Statement	*Use*
ALTER	Redirects the job input stream from one data set to another.
ASMGO	Creates a compile, load, and go sequence for the assembler program.
DSD	Establishes a connection between a data set or device and a DSD name used in a program.
EOF	Indicates the end of an input stream file.
EOJ	Ends a job.
EOS	Ends a step.
EXEC	Starts a step.
FORTGO	Creates a compile, load, and go sequence for the FORTRAN IV programs.
JOB	Starts and names a job.
NOEXEC	Used at an interactive device to cancel a step.
PARM	Used to pass parameters to a task set to be executed for a step.
PL1GO	Creates a compile, load, and go sequence for the PL/I programs.
*(comment)	* indicates that a comment follows. Comments can be placed anywhere in the input stream except between continuation statements.
/*	Ends an inline data set started by a DSD * statement.

Summary of Job Stream Processor Features

- The job stream processor gives you a simple control language for specifying requests to execute programs and defining the data sets and devices they require.
- Your input stream can come from an operator station or from a previously-created data set.
- You can redirect the source of the input stream. The input stream can be started from one device, then redirected to come from another device by an ALTER statement.
- You can predefine environment list DSDs. These lists will contain all the DSDs needed to execute particular jobs.
- You can add to the environment or override DSDs in the environment. These changes can be in effect for an entire session, a job, or a step.
- You can initiate a compile, load, and go sequence for PL/I, FORTRAN IV, or assembler programs by specifying PL1GO, FORTGO, or ASMGO.
- You can pass parameters to a task set that is to be executed in a step.
- You can have multiple stacked jobs per job stream and multiple steps per job.

Chapter 3. The Text Editor

The text editor provides a facility for creating and editing text modules. It executes in the batch partition and is invoked through the job stream processor. Once you define the required data sets and start the editor, you can enter new text or retrieve and modify existing text. The actual editing takes place in two work data sets referred to as the *editor workspace.*

The editor commands let you manipulate the text in a variety of ways. You can easily make changes, additions, and deletions to the text, then save the updated text module on an output data set. Text modules can be independently or compositely retrieved, edited, and saved during a single editing session.

The printed output is a listing of commands and text that you entered, along with the prompts and messages issued by the editor. You can also get a complete listing of the text module that you have created or updated.

Detailed information about the text editor will be contained in the *IBM Series/1 Program Preparation Subsystem: Text Editor User's Guide.*

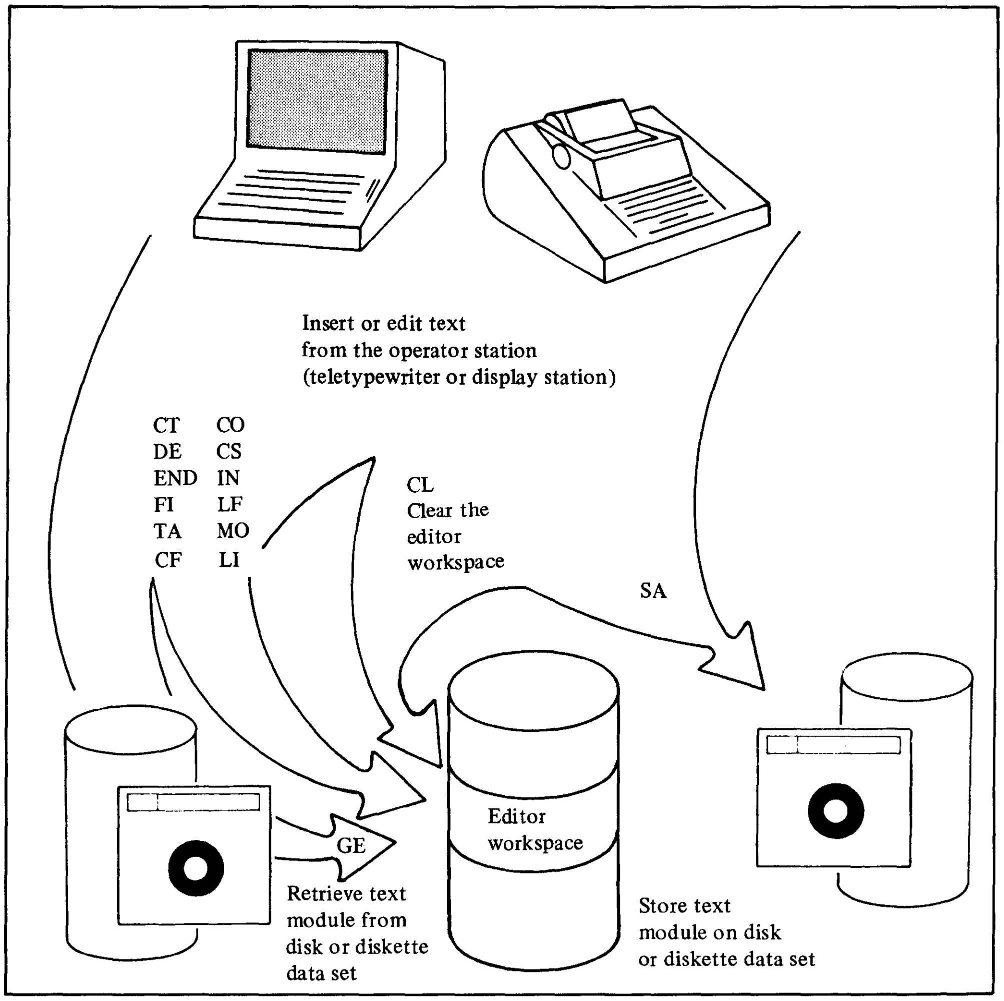


Figure 3-1. Retrieving, editing, and saving text with the text editor

Using the Text Editor

The editor features offer flexibility and ease of use in your editing sessions.

Interactive and Noninteractive Use

You can use the editor in an interactive or noninteractive mode. It executes in a noninteractive mode when the source of commands and data is a disk or diskette data set rather than an operator station.

Most of the time you will probably use the editor interactively, entering commands and data from a teletypewriter or display station.

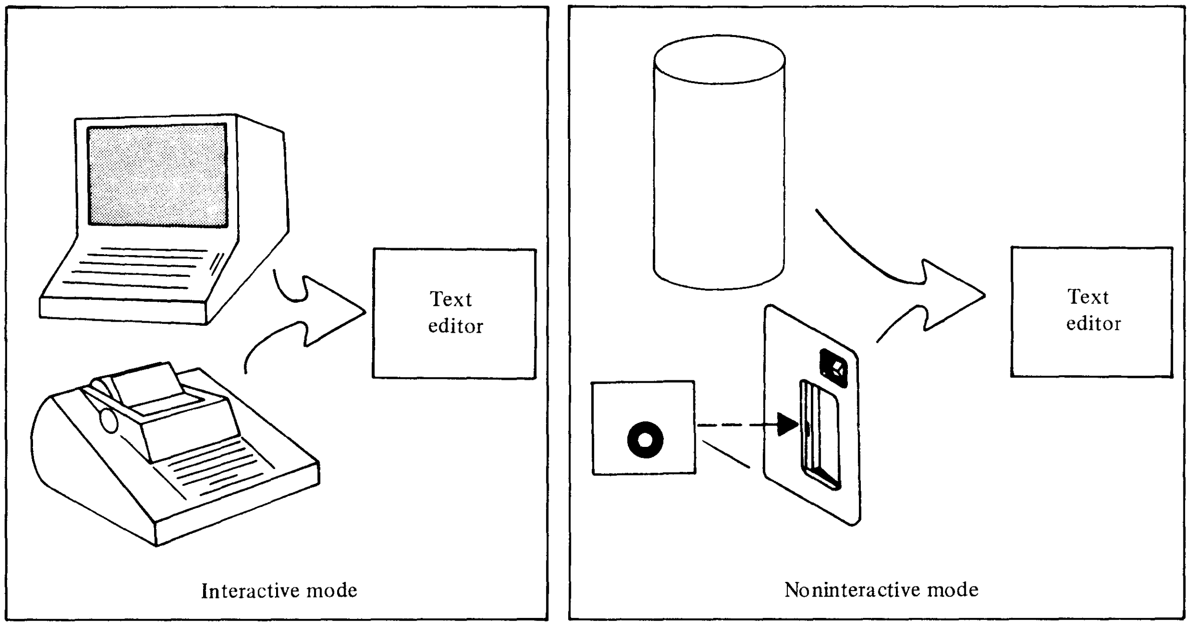


Figure 3-2. Interactive and noninteractive use of the text editor

If you use a display station that is also being used by the system, the lower portion of the display screen is reserved for system use, and the upper portion is for editor use.

Line Numbers

The basic unit of text is a line. As lines are placed in the editor workspace (through the get command or insert command), these lines are assigned temporary line numbers for reference. The line numbers are displayed at the beginning of the line of text but are not part of the text. You can specify the starting line number and the increment to be added to each succeeding line or let both values default to 10.

If you anticipate updating the text module, you have more editing flexibility if you let the starting line number and increment both default to 10. This way you can insert up to nine lines before or after each one of the original lines, without having to rearrange text or reassign line numbers. Figure 3-3 illustrates this concept.

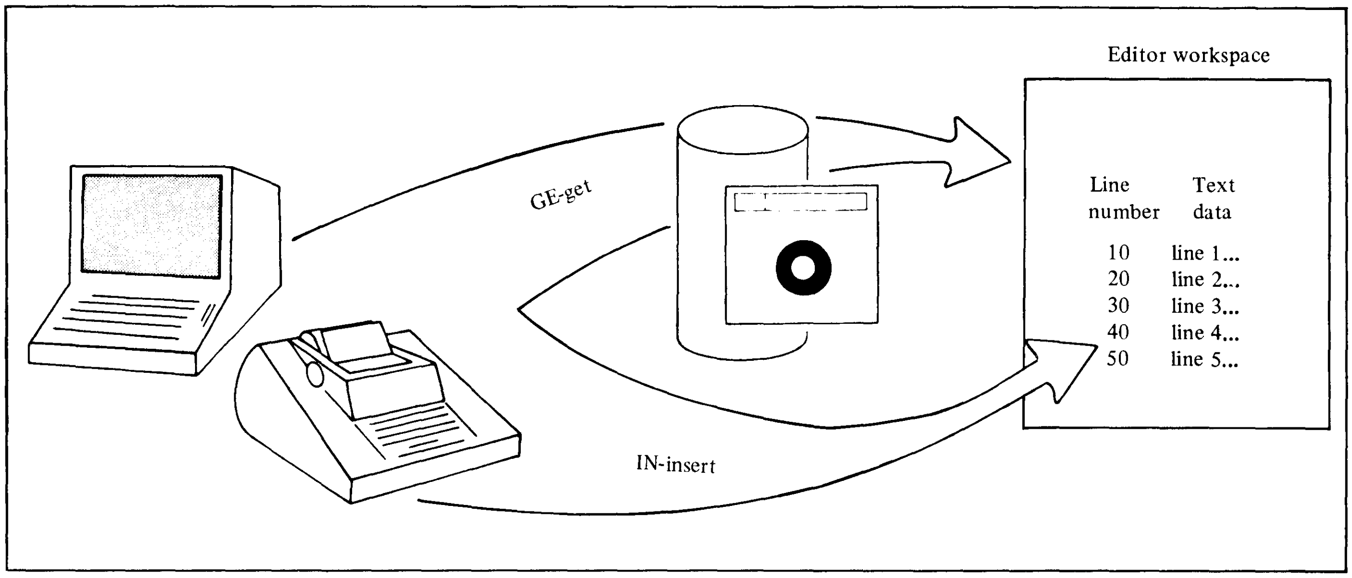


Figure 3-3. Editing by line number

Tailoring the Editing Session

The editor has features that allow you to tailor an editing session. You can set a line length, line display range, and tabs that will remain in effect for an entire editing session unless you change them.

Line Length

You can establish a line length—1 to 132 characters—for each record to be edited. This line length does not have to match the record length associated with the text modules to be edited. The line length defaults to 80 characters if you do not specify otherwise.

Line Display Range

You can also specify a line display range to control the length of a text line to be printed or displayed. This feature is useful when you want to avoid line wraparound. Line wraparound occurs when a line is too long to be printed on one device line and is therefore continued on the next line. This can detract from readability of the text lines. To eliminate line wraparound, you can specify a certain number of characters to be printed so that only as many characters as will fit on a device line will be printed.

The line display range can also be useful when you are editing from a teletypewriter and do not need to see the entire line printed out. Specifying a line display range to print only as much of the line as you need to see can reduce printing time.

If you do not specify a line display range, it defaults to the line length established by the line format (LF) command.

Tabs

The editor tabbing feature adds to your formatting flexibility. Through the tab (TA) command, you can establish tab settings that will be in effect throughout an editing session unless you change them. You can specify up to ten levels of indentation.

The tabbing feature is easy to use. Through the tab command, specify a *tab character* (which will invoke the tab function) and the column positions of all the tabs you want to set. Suppose you have specified a "/" as the tab character and set tabs at 10, 16, and 35. When you enter lines of text, simply type the "/" wherever you want a tab. Each time the editor encounters a tab character in a text line, it shifts the data following the tab character to the position you specified in the tab command (10, 16 or 35). The resulting "tabbed line" will have the tab character replaced by the proper number of blanks and will be placed in the editor workspace.

Example:

Text line as you entered it

```
LABEL/MVA/TABLE,R1
```

Text line in editor workspace

```
LABEL    MVA   TABLE,R1
```

Preserving Editing Sessions

To ensure that no text is inadvertently lost from the editor workspace when you end an editing session, the editor automatically preserves the text and editing environment for that session. This way, if you were to enter the END command before saving your text module, the text is not deleted from the editor workspace. The text and editing environment remain intact until you restart the editor. At that time, you can either continue editing the text in the editor workspace, or you can use the clear (CL) command to clear the editor workspace and start a new editing session.

In addition to preventing loss of text, this feature offers you another convenience. If you are interrupted during an editing session, you can end it and restart it later—without having to save the text, retrieve it, and respecify such information as line length, line display range, and tab settings.

Defining Editor Data Sets

Before you can invoke and use the editor, certain data sets must be defined. The DSD names and descriptions of data sets used by the editor are summarized in the following chart.

DSD Name	*Description*
CMDIN	Input device for commands and new data—usually the operator station.
MSGLOG	Message log device for listing text data, editor messages, and editor prompts—usually the operator station.
SYSLOG	System message log device—usually a printer device, so that you can have a hard copy of messages.
TXTIN	Text module input data set—usually a disk or diskette data set that defines the location of one or more text modules to be retrieved.
TXTOUT	Text module output data set that defines the location where edited modules are to be saved—usually a disk or diskette data set or data set member.
PRINT	Output print device for listing lines of text—usually a printer.
WORKVOL	Volume from which space for work data sets 1 and 2 are obtained. This must be specified.

You can specify these DSDs when you invoke the editor or you may have previously defined them in an environment list. An environment list can contain all the DSDs required to execute the editor. Using environment lists can simplify editing sessions. Once these lists are prepared, you can refer to the environment list you need without having to respecify all the DSDs needed to create the environment for a particular editing job.

Invoking the Editor

If you have already specified the necessary DSDs in an environment list, you can invoke the editor as easily as in this example:

```
TEJOB    JOB   ENVL=TELIST
TESTEP   EXEC  TSN=CPE
         EOJ
```

where TEJOB is the name of your job, TELIST is the environment list, and CPE is the text editor task set name.

Messages are printed at the operator station to tell you that the editor is started and ready to accept your editing commands.

The Editor Commands

The editor provides a variety of commands. You can use them to retrieve existing text or create new text. You can modify the text—insert and delete text lines, move and copy text lines, and change fields or character strings. When editing is completed, you can save the edited text on an output data set.

The following chart summarizes the editor commands and their functions.

Command	*Function*
CF *(change field)*	Replaces text data within a field in one or more lines.
CL *(clear)*	Clears the editor workspace to prepare for a new editing session and sets the line length to 80.
CO *(copy)*	Copies one or more lines from one area in the editor workspace to another (original lines not deleted).
CS *(current status)*	Prints the current status of the editor—the line length, last line number used, line display range, the tab character and tab settings.
CT *(change text)*	Changes a character string in one or more lines.
DE *(delete)*	Deletes one or more lines in the editor workspace.
END *(end)*	Ends the editing session.
FI *(find)*	Searches for text in one or more lines in the editor workspace and prints each line containing this text.
GE *(get)*	Gets text data from a disk or diskette data set member and places it in the editor workspace.
IN *(insert)*	Inserts new lines of text into the editor workspace.
LF *(line format)*	Sets the line display range for lines to be listed. Sets a line length.
LI *(list)*	Lists one or more text lines of the editor workspace.
MO *(move)*	Moves one or more lines from one area of the editor workspace to another.
SA *(save)*	Saves all text lines in the editor workspace to a data set or data set member on a disk or diskette (workspace contents not affected).
TA *(tab)*	Sets tabs to be used during an editing session.

Summary of Text Editor Features

- You can use the editor in an interactive mode (entering data and commands from an operator station) or noninteractive mode (data and commands coming from a data set or a data set member on a disk or diskette).
- You can use a teletypewriter or display station for editing in the interactive mode.
- You can establish tab settings, line length and a line display range to suit your needs for a particular editing session.
- At any time during the editing session, you can display the current status of the editor—the tab settings, tab character, the line length, line display range and last line number used.
- The editor automatically preserves the editor workspace contents when you end an editing session, so that you can restart the session later.
- The editor gives you a variety of easy-to-use commands to retrieve, create, delete, update, list and save text.

Chapter 4. The Macro Assembler

This chapter describes the macro assembler—the assembler program and the assembler language. Throughout the chapter, the term *assembler* is used to denote the *macro assembler.*

For detailed information about the assembler, refer to the *IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide*, SC34-0124.

The Assembler Program

The assembler program processes the machine, assembler and macro instructions you have coded in assembler language (the source program) and produces an object module in machine language.

The assembler also produces information for other programs. The application builder uses such information to combine object modules into load modules. The assembler processing sequence is illustrated in Figure 4-1.

Defining Assembler Data Sets

Before you can use the assembler program, you must define certain data sets to establish the DSD environment for assembler processing.

The DSD names and descriptions of data sets the assembler uses are summarized in the following chart.

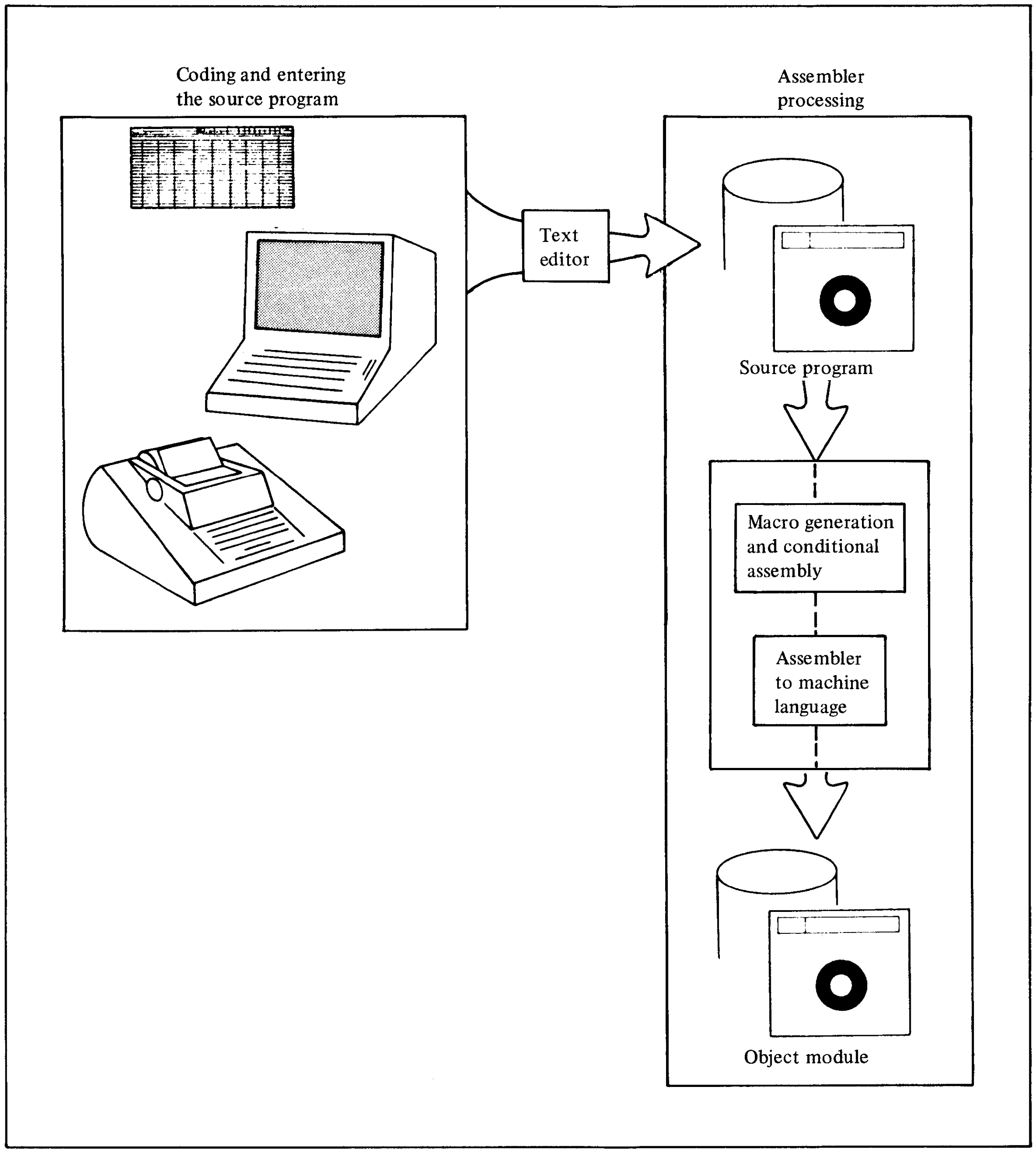
DSD Name	*Description*
LIB1 AND/OR LIB2	System or user macro data sets—disk or diskette data sets that define the location of macro files. (Optional)
OBJOUT	Object module output data set that defines the location where object modules are to be saved.
PRINT	Output print device—usually a printer where listings are to be printed.
SOURCIN	Assembler input data set—a disk or diskette data set that defines the location of input source modules.
SYSLOG	Device or data set where error messages can be printed. (Optional)
TSOVLY	Assembler phase and overlay data set.
WORK1,WORK2,WORK3	Work data sets used in assembler processing.
WORKVOL	Volume from which space for WORK1,WORK2 and WORK3 is obtained.

Invoking the Assembler

If you have already set up the recommended program preparation default environment—with DSDs for assembler work data sets, system logging device, object data sets, printer and system macro library—you can invoke the assembler through these simple control statements.

```
ASM1      JOB   ENVL=ALIST
STEP1     EXEC TSN=CPA
          EOJ
```

where ASM1 is the name of your job, ALIST refers to the environment list containing the required DSDs, and CPA is the assembler task set name.



Coding and entering the source program
Assembler processing
Text editor
Source program
Macro generation and conditional assembly
Assembler to machine language
Object module

Figure 4-1. Assembler processing sequence

The Assembler Language

Assembler language is a symbolic programming language containing statements that represent instructions and comments. There are three types of instruction statements:

- Machine instructions.
- Assembler instructions.
- Macro instructions.

Machine Instructions

A machine instruction is the mnemonic representation of a single hardware instruction. The mnemonic implies the function and the type of data operated on. For example, AB means add byte, SW means subtract word, and MVW means move word. The assembler program translates these mnemonic machine instructions into binary instructions that the system can execute.

Each machine instruction generates a hardware instruction. That instruction depends on the operation code and syntax of the operand. Based on the syntax of the operands, the assembler will generate one of several possible hardware instructions. If more than one hardware instruction can perform the operation specified by the mnemonic and its operand, the assembler generates the one that is most efficient in timing and storage usage.

There are more than 200 machine instructions that perform a wide variety of functions. The types of instructions included are:

- Data movement instructions.
- Arithmetic instructions.
- Branching instructions.
- Shift instructions.
- Stack instructions.
- Compare instructions.
- Logical instructions.
- Processor status instructions.
- Privileged instructions.
- Floating point instructions.

The individual instructions contained in each of these groups are described in detail in the *IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide,* SC34-0124. Input/Output instructions and data format information will be contained in the *IBM Series/1 Realtime Programming System: Macro User's Guide—Data Management.*

Assembler Instructions

An assembler instruction is a request to the assembler program to perform certain operations during the assembly of a source module. Some of these operations are defining data constants, defining the end of the source module, and reserving storage areas. Except for the instructions that define constants or provide boundary alignment, the assembler does not translate assembler instructions into object code.

Summary of Assembler Instructions

The following chart summarizes assembler instructions and their functions.

Instruction	*Function*
ALIGN	Allows you to ensure the setting of the location counter to an odd byte or word address during program assembly.
COM	Initiates a common control section or indicates its continuation.
COPY	Allows you to copy predefined source statements from a library and include them in your source module.
CSECT	Initiates an executable control section or indicates its continuation.
DC	Allows you to define data constants needed for program execution.
DROP	Allows you to terminate the USING domain for one or more registers.
DS	Allows you to reserve areas of storage, provide labels for these areas and use these areas by referring to the symbols defined as labels.
DSECT	Initiates a dummy control section or indicates its continuation.
EJECT	Stops the printing of an assembly listing on the current page and continues printing on the next page.
END	Marks the end of a source module to indicate to the assembler where to stop assembly processing.
ENTRY	Allows you to identify symbols defined in the source module containing the ENTRY instruction so you can refer to them in another source module. These symbols define locations called entry points.
EQU	Allows you to assign absolute or relocatable values to symbols.
EQUR	Allows you to define a register symbol by assigning to the symbol the value of an absolute expression.
EXTRN	Allows you to identify those symbols that are referred to in the source module containing the EXTRN instruction but are defined in another source module. These symbols are called external symbols.
GLOBL	Initiates a global control section or indicates its continuation.
ICTL	Allows you to change the begin, end, and continue columns to establish a different coding format for your source statements.
ISEQ	Causes the assembler to sequence-check the statements in your source module.
ORG	Alters the setting of the location counters and thus controls the structure of the current control section. You can, therefore, redefine parts of a control section.
POP	Restores the control section to the section on the top of the internal assembler section stack.
PREF	Allows you to generate a 1- to 5-word parameter list.
PRINT	Allows you to control the amount of detail you want printed in the listing of your program.
PUSH	Saves information about the current control section in an internal assembler stack.
SPACE	Allows you to insert one or more blank lines in the assembly listing.
START	Initiates the first executable control section in your source module.
TITLE	Allows you to provide headings for each page of the assembly listing.
USING	Allows you to specify that a register is available for use as a base register.
WXTRN	Allows you to identify those symbols that are referred to in the source module containing the WXTRN instruction but defined in another source module.

Macro Instructions

A macro instruction is a request to the assembler program to generate a predefined sequence of machine and assembler instructions. This predefined sequence of code is called a macro definition.

You can prepare your own macro definitions and invoke them by coding the appropriate macro instruction. From the macro definitions, the assembler generates an instruction sequence that is processed as if it were part of the original input in the source module.

The conditional assembly language is used within a macro definition to process input from a calling macro instruction. The conditional assembly language allows you to:

- Select statements for generation.
- Determine the generation order.
- Perform computations that affect the content of the generated statements.
- Produce assembly-time messages through the MNOTE instruction.

Macro instructions simplify coding for functions that are complex or occur frequently in a program. When you create your own macros, you can tailor the code with parameters you supply as operands. Each time you specify a macro in your program, the parameters you name are substituted for the prototype operands when the assembler program expands the macro.

The advantage to using macro language is that you reduce programming effort. You write and test the code for a macro definition only once. You and other programmers can then use the same code as often as you need by calling the definition. You only need to code one macro instruction to call for the generation of many assembler language statements from the macro definition.

Macro Instructions and Their Functions

The following chart summarizes the macro language instructions and their functions.

Instruction	*Function*
macro name	Provides the name of the macro definition.
prototype name	Provides a symbol that identifies the macro definition.
ACTR	Sets a conditional assembly loop counter.
AGO	Provides an unconditional branch.
AIF	Provides a conditional branch.
ANOP	No operation—branches to next sequential instruction.
GBLA	Allows you to declare initial value, type, and array dimensions for variable symbols (global SETA). *Note.* Global variables communicate values between macro definitions.
GBLB	Allows you to declare initial value, type, and array dimensions for variable symbols (global SETB).
GBLC	Allows you to declare initial value, type, and array dimensions for variable symbols (global SETC).
LCLA	Allows you to declare initial value, type, and array dimensions for variable symbols (local SETA). *Note.* Local variables are communicated within a single macro definition.
LCLB	Allows you to declare initial value, type, and array dimensions for variable symbols (local SETB).
LCLC	Allows you to declare initial value, type, and array dimensions for variable symbols (local SETC).
MACRO	Serves as the macro definition header. It is the first statement of every macro definition.
MEND	Indicates the end of a macro definition and provides an exit from the end of the macro definition when it is processed during macro expansion.
MEXIT	Causes the assembler to stop processing a macro definition and provides an exit from the middle of a macro definition when it is processed during macro expansion.
MNOTE	Allows you to generate an error message with an error condition code attached or to generate comments whereby you can display the results of preassembly operations.
SETA	Allows you to assign values to variable symbols (arithmetic).
SETB	Allows you to assign values to variable symbols (binary).
SETC	Allows you to assign values to variable symbols (character).

Assembler Coding Features

The assembler features make assembler language coding easier.

Symbolic Representation of Program Elements

Use of symbols can reduce programming effort and errors. Symbols can represent storage addresses, displacements, constants, registers, and other elements of the assembler language. Besides being easier to remember and code, the symbols are listed in a symbolic cross reference table in your program listing. This makes it easy to find a particular symbol when searching for an error in your code.

Variety of Data Representation

You can represent data in character, decimal, binary, ASCII, or hexadecimal. The assembler converts these values to binary values required by the machine instructions.

Relocatable Object Modules

These object modules can be relocated from the originally-assigned storage area to any other suitable main storage area without affecting program execution. Assignment to a physical memory location can be deferred until the code is linked with other object modules at application build time.

Address Assignment

To locate data to be operated on, most machine instructions refer to a storage address. You need not be concerned with specific main storage locations when you write a program because the assembler keeps track of locations of statements in your program (relative to the beginning of your program), then assigns addresses and displacements required when it produces the object program.

Flexible Register Usage

There are eight general-purpose registers that can be used to hold a value, an address, or displacement for manipulating data, maintaining counters, or determining the address of a particular instruction or storage location.

Program Sections

You can code a program in sections and, later, at application build time, combine the sections into an executable program. Advantages to this are:

- Many programmers can work on a large program, with each programmer working independently coding, assembling, and debugging his own section of the program.
- Less computer time is required because you can reassemble a section for the already-assembled program rather than reassembling the entire program.
- Sections common to more than one program are assembled only once, then linked to the unique sections of each program. This reduces computer time and results in shorter assembly listings that are easier to debug.
- You can configure a program to various main storage requirements much more easily by linking the sections into different combinations of storage loads or phases.

The four types of program sections that can be defined in the assembler language are summarized in the following chart.

Section Type	*Assembler Instruction*	*Function*
Control section	START or CSECT	Defines the object code–that is, machine instructions and data definitions.
Common section	COM	Defines an area of storage that can be shared with the program sections in multiple assemblies within a task.
Global section	GLOBL	Defines either task set global or system global. Task set global is addressable by all programs executing in the same partition. System global is contained in the shared task set and is addressable by all programs linked to the shared task set.
Dummy section	DSECT	Describes the format of data located elsewhere.

Symbolic Linkages Between Separately-Assembled Source Modules

Symbols can be defined in one module and referred to in another. If a linkage symbol is referenced by another module, it must be identified by an ENTRY assembler instruction unless it is a START or CSECT statement. If a module refers to a linkage symbol defined in another module, the symbol must be identified by an EXTRN or WXTRN statement.

Comprehensive Assembler Listings

The listings generated by the assembler contain source statements, macro expansions, ESD, RLD and cross reference tables, error messages and statistics. You can control the content of the listing output with assembler options.

Assembler Options

The following chart summarizes the assembler options. (Defaults are underlined). You can specify parameters to the assembler through the PARM control statement.

Option	*Explanation*
LIST/NOLIST	LIST—write all assembly listings to the PRINT data set. NOLIST—write only error messages to the PRINT data set.
TEXT/NOTEXT	TEXT—write source and object program listing to the PRINT data set. NOTEXT—suppresses the option.
XREF/NOXREF/FULLXREF	XREF–write cross reference listing to the PRINT data set only for referenced symbols. NOXREF—suppresses this processing. FULLXREF—write the cross-reference listing to the PRINT data set for all defined and referenced symbols.
ESD/NOESD	ESD—write the external symbol dictionary before the source program listing. NOESD—suppresses the option.
RLD/NORLD	RLD—write relocation dictionary after source program listing NORLD—suppresses the option.
SYSPARM('...')	Defines up to 8 characters of information substituted for the &SYSPARM value during macro processing.
OBJECT/NOOBJECT	OBJECT causes the object module to be written to the OBJOUT data set. NOOBJECT suppresses this processing.
LINECOUNT (n)	LINECOUNT specifies *n* as the number of lines per page of the PRINT data set. Default line count is 55. The value of *n* can be from 1-999.
MACRO/NOMACRO	MACRO—macro processing is desired. NOMACRO—suppresses macro processing.

Structured Macro Usage

A set of structured macros, which can be used as input to the assembler, is provided in a separate macro library. Some advantages of using structured macros are:

- They eliminate branch and jump instructions in source code.
- They generate reenterable code.
- They allow you to use all hardware facilities (such as indirect addressing) and data types (such as bit processing).
- They permit up to 20 levels of structure nesting.
- They print English-language assembly-time diagnostic messages, which can include the variables you have coded. This helps you to find and correct errors.
- They are easy to read and understand.

Use of structured macros can reduce errors, increase productivity, and reduce development and maintenance costs.

Summary of Macro Assembler Features

- You can code machine instructions using a function-oriented assembler language—the mnemonics imply the function.
- You can code macro instruction statements to generate a series of assembler language statements.
- You can use conditional assembly instructions within macro definitions to selectively generate sequences of instructions.
- You can use the structured macros.
- Source statements can come from an input data set or from a library.
- The assembler can process four types of program sections—common sections, control sections, global sections, and dummy sections.
- Programs may reference symbols defined in separately-assembled programs. These external references are later resolved during application builder processing.
- The assembler produces object modules that include information for module relocation and external symbol resolution.
- The assembler also generates comprehensive listings. You can control the listing output by options you specify to the assembler.

Chapter 5. The Application Builder

The application builder handles the final steps in the program preparation sequence. It is the subsystem component that processes object modules to prepare them for execution as application programs.

To fully understand the significance of application builder processing, you should be familiar with the information presented in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102, particularly the sections describing task sets.

Detailed information about how to use the application builder to build task sets will be contained in *IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide.*

Note. Throughout this chapter, the *Realtime Programming System* is also referred to as the *operating system.*

Execution Environments Supported

Regardless of the execution environment for which you are preparing the application program, you must at least use the application builder to convert from object module format to composite or absolute load module format. An *object module* produced by a language translator, which is the initial input to the application builder, may contain one or more object programs and control information. The output of application builder processing will depend on how you use the application builder phases.

There are three phases in application builder processing, as shown in Figure 5-1.

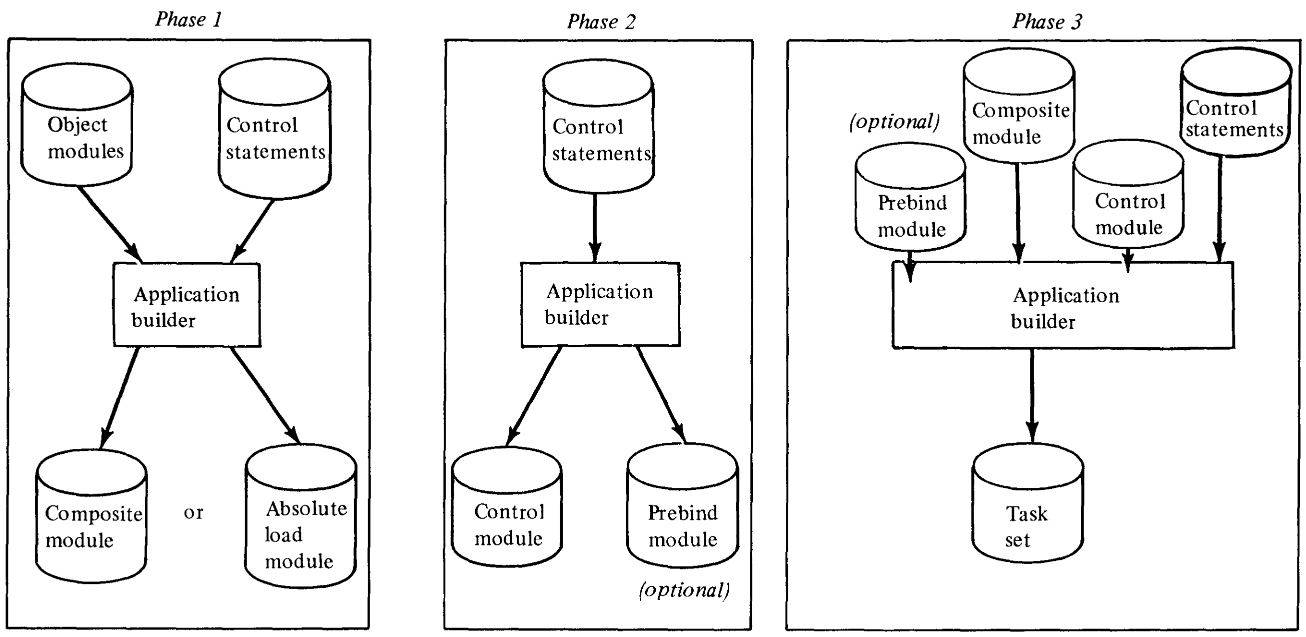


Figure 5-1. The three phases of application builder processing

Phase 1. This phase can create a load module in absolute format or a *composite module* in relocatable format. Absolute load modules are not executable under the Realtime Programming System but can be executed in an environment you provide. A composite module, in relocatable format, will be used as input to phase 3 processing. A composite module is made up of object modules (programs) structured into a resident segment and optional overlay segments.

Phase 2. This phase builds control modules and prebind modules. The *control module*, which is required by the operating system, provides the tables and control blocks that the system needs in order to execute the functions requested (data management, queuing, and tasking, for example). The *prebind module*, which is optional, allows you to specify the information needed for binding the required resources to an application task set when it is installed rather than at execution time. This prebinding of resources allows a task set to start execution faster.

Phase 3. This phase creates task sets that will be executable in the Realtime Programming System environment. A task set is a planned program structure—a named collection of programs, data, and control blocks designed to execute within a partition under the operating system.

If you are creating an application program for execution in an environment you provide, you need only use the application builder for phase 1 processing. If you are creating an application program for execution under the operating system, you must use the application builder for all three phases in order to supply the information needed by the system to execute your application program, which must be in the form of a task set.

To simplify describing the phases of the application builder, this chapter first gives a brief explanation of what you must do to create an application program for execution in an environment other than that of the Realtime Programming System. The rest of this chapter assumes that you are using the application builder to create an application program (a task set) that will execute under the operating system.

An Environment You Provide

This can be any execution environment other than the Realtime Programming System environment. Creating an absolute load module for execution in an environment you provide requires only phase 1 processing.

Creating Absolute Load Modules

Phase 1 combines object modules produced by a language translator to create an absolute load module. The application builder control statements allow you to identify the contents, structure and name of the output load module. Through phase 1 control statements, you can specify the:

- Object modules to be included in the output load module (INCLUDE statement).
- Entry point of the output load module (ENTRY statement).
- Member name of an output load module, if more than one absolute load module is being created (NAME statement).
- Format of the output module—absolute or relocatable (FORMAT statement).

You *must* specify an origin on the FORMAT control statement in order to create a load module in absolute format. Phase 1 processing assigns origin addresses to control sections and resolves external references within the control sections. The origin address will be whatever you specify on the FORMAT statement. References in one control section to items in another control section of a different object module are resolved relative to the address assigned to the item in the output load module.

The output load module, in absolute format, consists of a header record (containing descriptive information about the load module–such as its name, origin address, length of text, etc.,) and text records, which make up an executable program with all address constants resolved against a specific storage address. An absolute load module is not executable under the Realtime Programming System, and you cannot use any of the system support functions. You must also provide a means of loading the module into storage and passing control to it. The DISK IPL Bootstrap Loader, which is described in the *IBM Series/1 Stand-Alone Utilities User's Guide*, GC34-0070, can be used for loading a program into storage and passing control to it.

The Realtime Programming System Execution Environment

To create an application program that can execute in the Realtime Programming System environment, you must use phases 1, 2, and 3, of the application builder. Phases 1 and 2 create modules that are input to phase 3, which combines them into a task set and writes that task set to a *task set library*. A task set library is a volume containing all of the modules and tables associated with a single task set.

In phase 3 processing, you must supply partition information because task sets under the Realtime Programming System are executed within a partition. Task sets must be prepared for a specific partition because they are not relocated when loaded.

A task set can have a simple structure, where all segments are resident during execution, or complex structure (overlay), where some segments reside on disk. In overlay structure, a storage-resident segment of code can request the system to retrieve a unit of code from disk and pass control to it. The specific structure and characteristics of the task set are defined by the information you supply on control statements that are input to the application builder.

Figure 5-2 represents the system and partition storage organization for the Realtime Programming System execution environment.

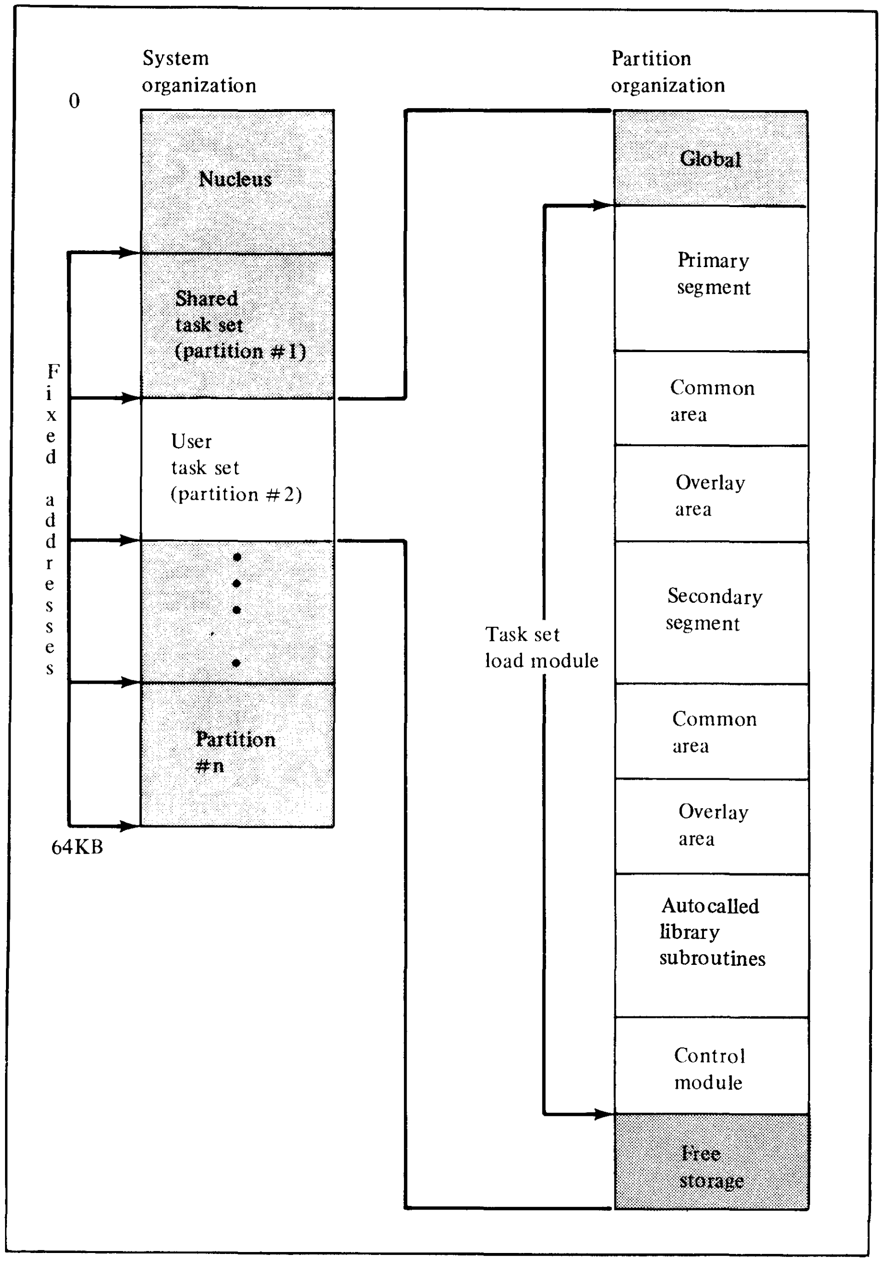


Figure 5-2. System and Partition Storage Organization

Task Set Load Module. The *task set load module* is a structure that contains all resident segments, their associated common and overlay areas, and the control module, for a single task set.

Resident Primary Segment. The *resident primary segment* is a segment that remains in primary storage for the duration of task set execution. (A program in a resident segment may call a program in another resident segment.) This resident segment contains the initial entry point of a task set. The entry point of a task set must be a primary program within the primary segment.

Common Area. The *common area* reserves an area of storage and can be referred to by resident and overlay segments that are associated with a given primary or secondary segment within a task set.

Overlay Area. The *overlay area* is where disk overlays are loaded and passed control. Each resident segment that calls overlays has its own disk overlay area.

Resident Secondary Segment. The *resident secondary segment* can be any resident segment other than the primary segment of a task set. It differs from a primary segment in that it does not contain the task set's entry point.

Resident Library Subroutines Autocalled. These are modules whose purpose is to complete a resolution within the task set load module. These routines are obtained from a disk-resident autocall library and are added to the task set as if they were another resident secondary segment.

Control Module. A *control module* is required by the operating system for task set execution. It consists of a set of tables and control blocks that the system needs in order to execute the functions requested.

Application Builder Processing

The following sections describe the purpose and functions of each of the three phases of application builder processing. Throughout these sections, it is assumed that you are using the application builder to create a task set for execution under the operating system. Other execution environments are *not* considered.

Phase 1 Processing

Object modules produced by a language translator are not in a format that allows them to be loaded or executed. These object modules contain:

- An external symbol dictionary (ESD) containing information to be used in resolving symbolic references between control sections of different modules.
- Text data that is the actual program instructions and data areas.
- A relocation list dictionary (RLD) containing an entry for each relocatable address constant.
- An end of module record.

Phase 1 combines one or more object modules to form a single composite module, in relocatable format, that will be used as input to phase 3 processing.

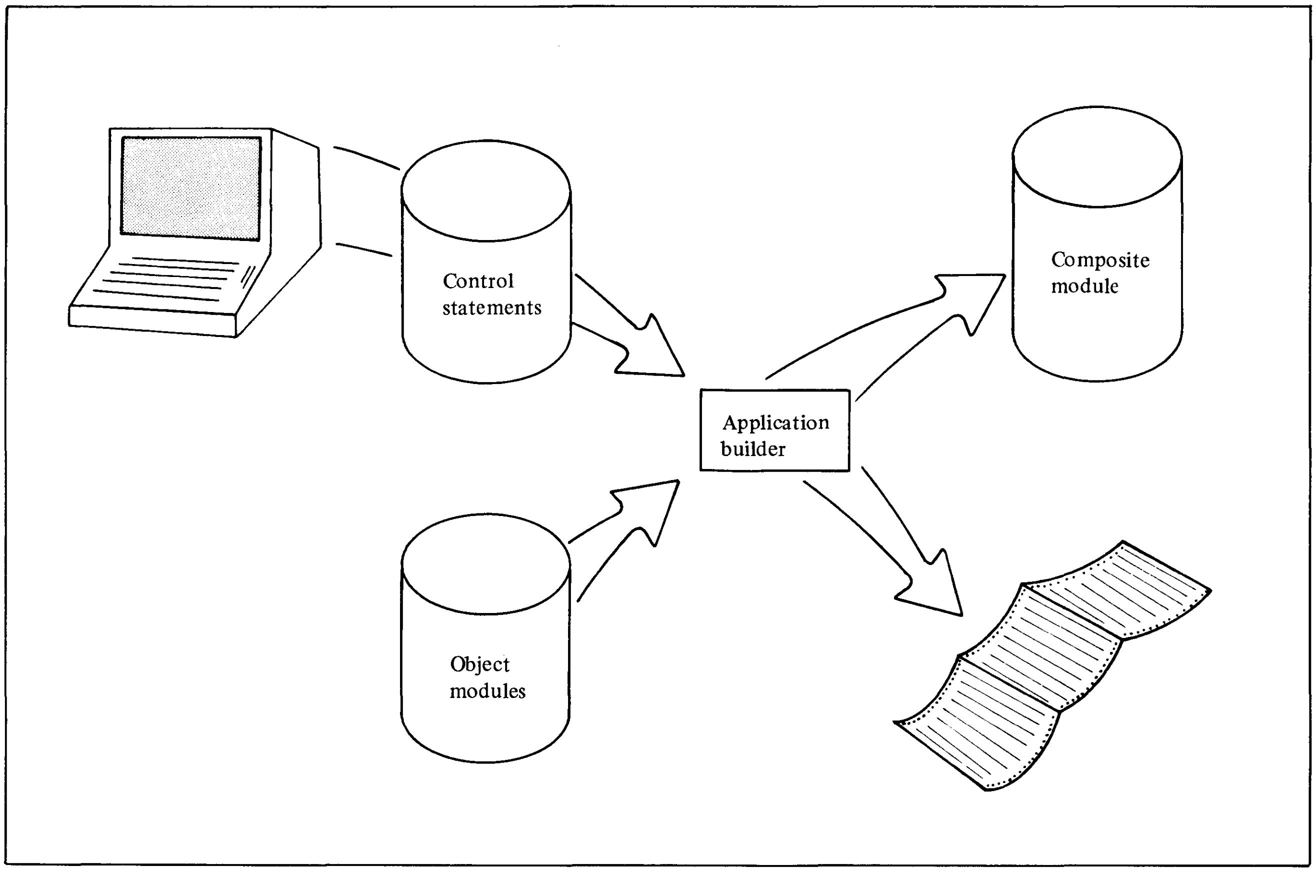


Figure 5-3. Application Builder phase 1 processing

Figure 5-4 shows the basic functions performed during phase 1 processing.

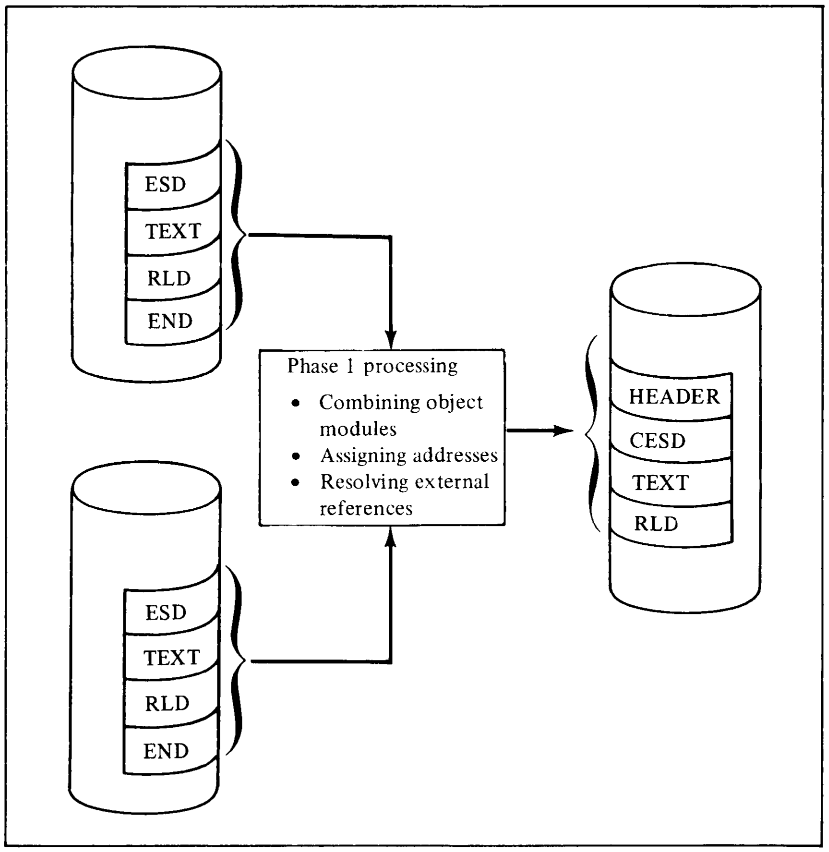


Figure 5-4. Combining object modules

Composite Module Output

Composite modules produced by phase 1 can be simple or overlay structured. Overlay structured modules consist of multiple segments. The first segment is the resident segment, and successive segments are the overlay segments.

A composite module can be placed in a consecutive data set or placed as a member of a partitioned data set. Composite modules grouped within a partitioned data set form a *composite module library*.

Printed Output

In addition to the output composite module, phase 1 produces:

- A listing of your control statements.
- A composite module map identifying names and assigned addresses of control sections.
- Diagnostic messages indicating results of phase 1 processing.

Phase 1 Control Statements

The phase 1 control statements allow you to identify the contents, structure, and name of the output composite module. The following chart summarizes phase 1 control statements.

Control Statement	*Purpose*
PHASE1	Introduces phase 1 control statements and causes phase 1 to be invoked.
INCLUDE	Specifies object modules to be included in the composite module.
OVERLAY	Identifies the beginning of an overlay segment.
ENTRY	Identifies an entry point to a program in the primary resident segment.
NAME	Identifies the member name of the composite module.
FORMAT	Identifies the type of output module (absolute or relocatable).

Phase 2 Processing

Phase 2 creates the control module and, optionally, a prebind module to be used in the Realtime Programming System execution environment.

Control Module

A control module provides the tables and control blocks required by the system to execute the functions requested. This may include:

- The task set control block.
- A DSD list of data sets to be prebound.
- Preallocated control block stacks.
- The variable control block area, which is an area of storage available for variable length control blocks (for the supervisor and data management).
- Sensor I/O tables.

The control module, which is required by the system for execution of task sets, is included in the task set produced by phase 3 processing and occupies storage in the partition in which the task set will execute.

Prebind Module

A prebind module is used to specify the required resources to be prebound to an application task set at installation time rather than at execution time. The prebind module is optional and does not occupy storage in the user partition. Through the application builder control statements, you can specify prebinding to be performed in order to:

- Prebind tasks by allocating control blocks and task work stacks.
- Predefine queues by allocating control blocks and preopening disk data sets for disk queues.
- Establish the conditions under which scheduled task sets are to execute, by updating the system scheduler table.
- Preopen data sets.

Phase 2 processing also generates a list of your control statements and a list of error messages. Figure 5-5 illustrates phase 2 processing.

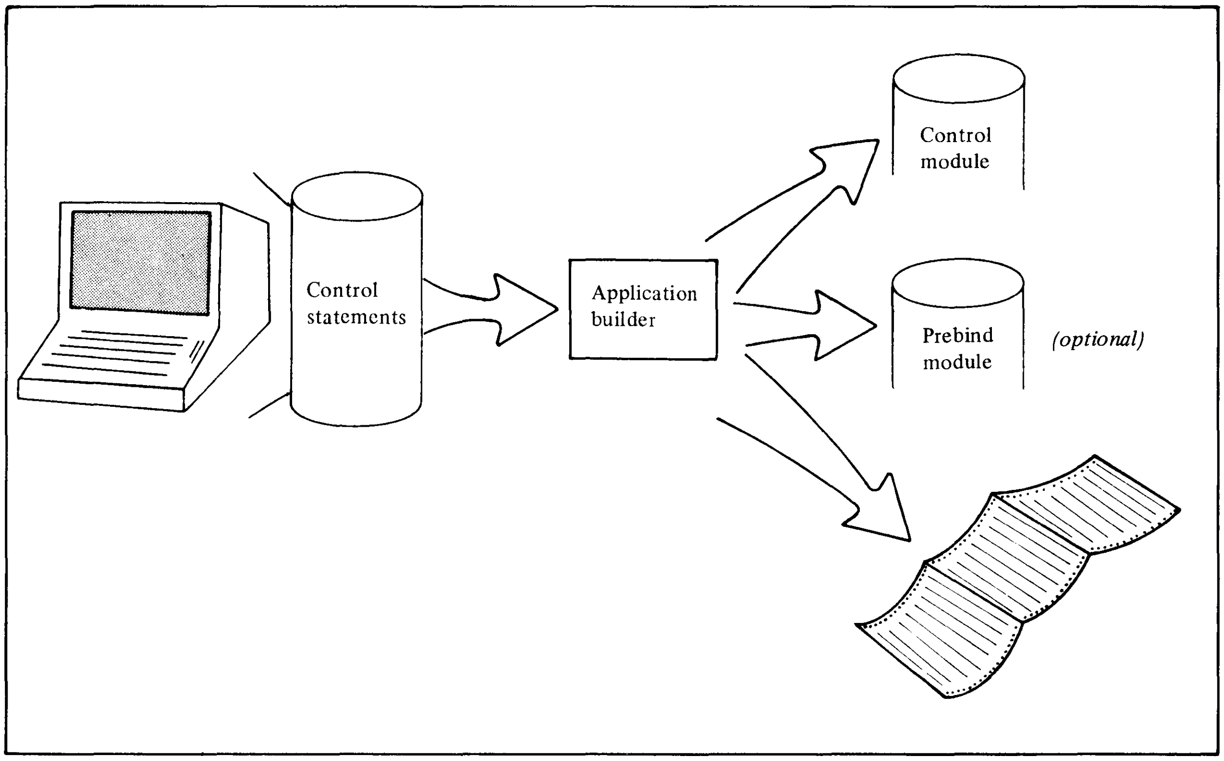


Figure 5-5. Application builder phase 2 processing

Control Statements

Phase 2 control statements can be used to define the control module or both the control and prebind modules. The following chart summarizes phase 2 control statements.

Control Statement	*Purpose*	*For Control Module*	*For Prebind Module*
PHASE2	Introduces phase 2 control statements and causes phase 2 to be invoked.	X	X
TASKSET	Describes the task set (name, partition, etc.).	X	
TASK	Describes a task (entry point, dispatching priority, etc.).	X	X
SIO	Defines sensor I/O used by task set.	X	
CTBLKS	Preallocates control blocks needed at execution.	X	
QUEUE	Defines the queue to be prebound to the task set.		X
DATASETS	Defines data sets to be prebound to a task set.	X	
EXCOND	Describes a condition under which the task set is to be executed.		X
SHARING	Defines resource names that are otherwise not defined in the control module or task set. This lets you define names of resources (in a shared task set) that may be used by other task sets.	X	

Phase 3 Processing

Phase 3 combines the modules created in phases 1 and 2 to create a task set that is executable in the Realtime Programming System environment. The primary input to phase 3 consists of the modules produced by phases 1 and 2. These modules may be obtained from:

- Data sets containing the control module and prebind module.
- An autocall library.
- A consecutive data set or a partitioned data set (library) that you specify through the phase 3 INCLUDE control statement.

Figure 5-6 illustrates phase 3 processing.

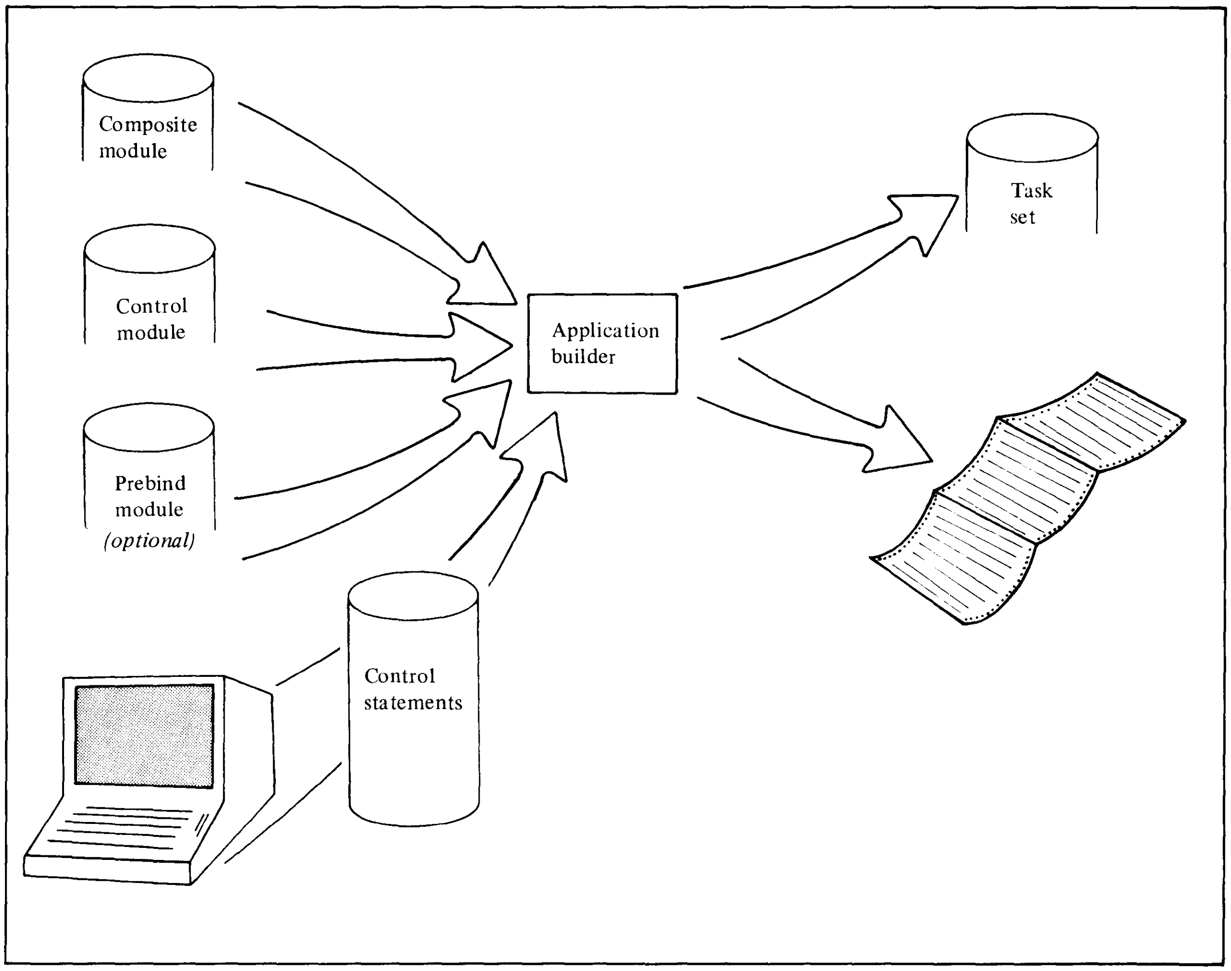


Figure 5-6. Application builder phase 3 processing

Basically, phase 3 processing involves the following operations.

Control Statement and Composite Module Processing. Your input control statements are analyzed and processed. Composite modules are combined into a single task set, with all addresses resolved relative to a partition origin.

External Reference Resolution. External references from a composite module are resolved to definitions in other modules, a shared task set, or a module from the autocall library that is automatically included in the task set.

Global Processing. Global sections from the composite modules are combined into a task set global area.

Overlay Processing. Overlay segments from the composite modules are combined into a separate data set that will be used during task set execution. Space is allocated in the task set load module for each overlay area.

Task Set Creation. Resident programs, overlay areas, and the task set control module make up a task set load module with addresses assigned and references resolved. The task set load module is written to a data set in the task set library.

Indicating Results of Processing. A return code indicates whether or not phase 3 was completed successfully. You get a listing of control statements, a task set map, and diagnostic messages.

Control Statements

Through phase 3 control statements, you can specify the contents, structure, and characteristics of the output task set. The following chart summarizes phase 3 control statements.

Control Statements	*Purpose*
PHASE3	Introduces phase 3 control statements and causes phase 3 to be invoked.
INCLUDE	Identifies composite modules that are to be combined to form a task set.
PARTN	Specifies characteristics of the task set and the partition in which it is to execute.

Summary of Application Builder Data Sets

The following chart summarizes the DSD names and descriptions of data sets used by the application builder to obtain its input, perform the processing requested, and produce the appropriate output.

DSD Name	*Description*
APBIN	Contains the control statements you specify.
AUTO1—AUTO3	Indicates libraries used for automatic call processing. (Autocall processing allows modules to be obtained automatically from a program library.)
BUILDIN1—BUILDIN5	Identify data sets (partitioned or consecutive) containing object modules to be included as input to phase 1.
CONTROL	Identifies a partitioned data set to contain the control module and (optionally) the prebind module.
LMODOUT	Identifies the data set (partitioned or consecutive) to contain the output module created in phase 1 processing.
LOAD1—LOAD5	Identify data sets (partitioned or consecutive) containing composite modules to be included as input to phase 3.
PBMDS	Identifies the data set that is to contain the resolved prebind module for a task set.
PRINT	Identifies the output data set to contain listing information such as control statements and maps.
SYSLOG	Identifies system message log device.
TSLIB	Identifies volume in which the unbound task set, disk overlay module, and resolved prebind module data sets are defined.
TSLOAD	Identifies the data set that is to contain the output task set.
TSOVLY	Identifies the data set to contain disk overlays.
TSRTDS	Identifies the data set that is to contain the task set reference table.
TSSHARE	Identifies the consecutive data set containing the task set reference table of the shared task set to be used for resolution when building your task set.
WORK1, WORK2, WORK3	Identifies intermediate work data sets.
WORK4	Identifies a data set to contain overflow queues.
WORKVOL	Volume from which space for work data sets can be allocated. Any or all of the WORK1—WORK4 data sets can be allocated from that volume.

Invoking the Application Builder

Assuming you have defined the required data sets and devices and have described the content and structure of the output module, the following example shows the minimum control statements needed to invoke the application builder.

```
APB1      JOB   ENVL=APBLST
STEP1     EXEC  TSN=CPD
          EOJ
```

where APB1 is the name of your job, APBLST refers to the environment list containing the required DSDs, and CPD is the task set name of the application builder.

Listing and Processing Options

You can specify certain listing and processing options to the application builder through the job stream processor PARM statement. These parameters are passed to the application builder by the job stream processor.

The following chart summarizes the options. Defaults are underlined.

Options	*Explanation*
LIST = <u>YES</u> NO	YES causes a listing of the specified input to be created. NO suppresses the LIST option
MAP = <u>ALL</u> LM TS NO	The MAP option causes a map of the output data set to be created. The map indicates the structure and symbols, both resolved and unresolved. ALL causes a map for the composite module and task set to be created. LM causes a map to be created for only the composite module. TS causes a map to be created for the task set only. NO suppresses the MAP option.
AUTO = YES <u>NO</u>	YES causes a further attempt to resolve symbols that are unresolved from within a task set by checking a library you specify for the unresolved symbol. NO suppresses this option.
STS = YES <u>NO</u>	YES causes a shared task set file to be checked to resolve unresolved symbols before an attempt is made using the AUTO option. <u>NO</u> suppresses this option.

Summary of Application Builder Features

- The application builder performs the necessary processing operations—such as assigning storage addresses and resolving external references—to combine input object modules and create an absolute load module or a composite module.
- It allows for creating an absolute load module that is executable in an environment you provide.
- It allows for creating a composite module that will undergo further application builder processing to ultimately become part of a task set that will be executable in the Realtime Programming System environment.
- It allows modules to be obtained automatically from a program library.
- It provides for recycling to create multiple composite modules.
- It allows for creating a control module, which provides the control blocks that the system needs to execute such functions as data management, queuing, and tasking.
- It allows for creating a prebind module, which contains specifications that can be used to prebind resources to a task set at installation time rather than at execution time.
- It allows you to create task sets that will be executable in the Realtime Programming System environment.

Appendix A. Configuration Requirements

Hardware

The Program Preparation Subsystem uses the hardware required and supported by the Realtime Programming System. The minimum hardware (and optional hardware) to be used for the subsystem programs is described in chapter 1 of this publication.

Storage Requirements

To install the subsystem programs, you need a minimum of 48KB of processor storage. For installation, the entire 48KB is used by the starter system, but once the subsystem programs are installed, a minimum of 16KB is required for executing these programs.

Disk Requirements

The subsystem programs are installed by using the starter system provided by the Realtime Programming System and copying the programs from diskettes to disk.

Once these programs are installed, the starter system is deleted from disk and therefore is not considered disk space that must be permanently reserved.

The subsystem programs will require the following estimated space.

Note. The total number of cylinders on the disk is 302.

Program Residency Space

The amount of space required for each subsystem program is:

Subsystem program	*Cylinders*
Job stream processor	0.7
Text editor	1.0
Macro assembler	5.0
Application builder	3.0
Total	9.7

Message Data Set Space

The amount of space required for the subsystem programs that have defined a separate data set for their messages is:

Subsystem program	*Sectors*
Job stream processor	25
Text editor	34
Macro assembler	—
Application builder	—
Total	59 sectors (0.5 cylinders)

Note. Message data set space for the macro assembler and application builder is included in the program residency space.

Work Data Set Space

The amount of space required to satisfy typical use of the subsystem program during its execution is:

Subsystem program	*Cylinders*
Job stream processor	00.5
Text editor	57.0
Macro assembler	50.0
Application builder	41.0
Total	57.0**

**Since only one batch program will be executing at any given time, it is expected that the same workspace will usually be used by all the subsystem programs. Therefore, only the largest workspace requirement is accounted for in determining workspace requirements.

Total Disk Space

The estimated total disk space required for the subsystem programs is:

Disk space	*Cylinders*
Program residency space	09.7
Message data set space	00.5
Work data set space	57.0
Total	67.2 (68)

Note. This total amount does not account for control blocks/tables (i.e., data set definition table (DSDT) that is defined within each task set library), which will be required for subsystem components. However, any additional space is considered negligible compared to the total above.

Diskette Requirements

The only requirement the subsystem programs have for diskettes is during installation. The Program Preparation Subsystem program product is on six diskettes.

1. The job stream processor, generation program, application builder, and verification compare program.
2. Job stream processor and text editor composite modules.
3. Macro assembler object modules.
4. Application builder composite modules.
5. Structured macros.
6. Structured macros.

The data copied to disk from the first diskette will not be retained after the Program Preparation Subsystem is installed. If the subsystem programs must be reinstalled, the starter system and first diskette are again copied to disk, and the process is repeated.

Timer Requirements

The timer is an optional feature. If available, the subsystem programs will use its services as follows.

Subsystem Program	*Use*
Job stream processor	Date/time stamping jobs (job accounting).
Text editor	Date/time stamping editing sessions that are saved for restarting at a later time.
Macro assembler	Date/time stamping assembled output listing.

Printer Requirements

The Program Preparation Subsystem generation program generates job stream processor control statements of up to 80 characters in length. These control statements will be listed at the printer.

Each subsystem program will require a printer to display its output. Depending on the subsystem program in use, from 120 to 132 characters will be required for use from the 132-character printer line length.

The job stream processor and text editor do not necessarily require a printer for execution.

Programming Requirements

In order to support the Program Preparation Subsystem program product, the only requirement in the SYSGEN of the Realtime Programming System is that a batch partition of a minimum of 16KB of processor storage be defined. The basic Realtime Programming System provides all the support functions (that is, supervisor, data management, utilities) required for installation and for the subsystem programs to operate.

Compatibilities

The Base Program Preparation Facilities and the Program Preparation Subsystem

The Program Preparation Subsystem assembler language syntax is compatible with that of the Base Program Preparation Facilities assembler language. Any statement using the Base Program Preparation Facilities assembler syntax will assemble properly using the Program Preparation Subsystem macro assembler.

The commands of the Base Program Preparation Facilities and Program Preparation Subsystem text editors are compatible. Although the subsystem text editor has additional commands, the commands common to both text editors have the same syntactical operands. The one exception to this is the data set references on the get (GE) and save (SA) commands for the subsystem text editor.

The Realtime Programming System and the Program Preparation Subsystem

The Program Preparation Subsystem uses the Realtime Programming System standard system and data management interfaces to perform the following types of operations:

- Space allocation.
- Task set transfer control and queuing.
- Data set creation.
- Accessing and deleting data sets.
- Parameter passing.

Appendix B. Using the Series/1 Programming Library

From the time you plan and install your system until the time you are actually running your application programs on the Series/1, you will be involved in a variety of activities. In the course of these activities, you will be gathering information from various publications. Knowing which publication to use for which type of activity can make your job much easier.

To make it easier for you to find the publication you need, this appendix summarizes the steps involved in various types of programmer activities and ties them to the appropriate Series/1 programming publication (or publications).

Note. For an overview of Series/1 system functions and a comparison of how they are supported by different languages, operator commands, or macros, refer to the "Summary of System Functions" in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

The following table shows the programmer activity, the associated system step, and the related publications. Order numbers are given for publications that are available at this time.

For the following activity...	*The associated system step is...*	*Refer to Publications...*
PLANNING AND SYSTEM DESIGN	None.	
• Understanding the Realtime Programming System and the purpose of each of the program products in the Series/1 software system.		*IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102
• Understanding the purpose and features of:		
– The Program Preparation Subsystem.		*IBM Series/1 Program Preparation Subsystem: Introduction*, GC34-0121
– PL/I.		*IBM Series/1 PL/I: Introduction*, GC34-0084
– FORTRAN IV.		*IBM Series/1 FORTRAN IV: Introduction*, GC34-0132
– The Mathematical and Functional Subroutine Library (MFSL).		*IBM Series/1 Mathematical and Functional Subroutine Library: Introduction*, GC34-0138
• Understanding the total Series/1 hardware and software offering.		*IBM Series/1 System Summary*, GA34-0035
BUILDING THE REALTIME PROGRAMMING SYSTEM	System generation.	*IBM Series/1 Realtime Programming System Generation and Installation Procedures*
BUILDING THE PROGRAM PREPARATION SUBSYSTEM	System generation.	*IBM Series/1 Program Preparation Subsystem: Batch User's Guide*

For the following activity...	*The associated system step is...*	*Refer to Publications...*
CODING YOUR PROGRAMS	None.	
• Assembler language programs.		*IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide*, SC34-0124
		IBM Series/1 Program Preparation Subsystem: Macro Assembler Reference Summary
– Using the routines in the Mathematical and Functional Subroutine Library.		*IBM Series/1 Mathematical and Functional Subroutine Library User's Guide*
– Using the Realtime Programming System macros.		*IBM Series/1 Realtime Programming System Macro User's Guide—Supervisor*
• FORTRAN IV language programs.		*IBM Series/1 FORTRAN IV Language Reference*, GC34-0133
– Using the routines in the Mathematical and Functional Subroutine Library.		*IBM Series/1 Mathematical and Functional Subroutine Library User's Guide*
• PL/I programs		*IBM Series/1 PL/I Language Reference*
USING THE OPERATOR CONSOLE		*IBM Series/1 Realtime Programming System: Operator Commands and Utilities*
USING THE JOB CONTROL LANGUAGE	Job stream processor.	*IBM Series/1 Program Preparation Subsystem: Batch User's Guide*
CREATING SOURCE MODULES	Text editor.	*IBM Series/1 Program Preparation Subsystem: Text Editor User's Guide*
CREATING OBJECT MODULES	Assembler program.	*IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide*, SC43-0124
	PL/I compiler.	*IBM Series/1 PL/I User's Guide*
	FORTRAN IV compiler.	*IBM Series/1 FORTRAN IV User's Guide*
CREATING COMPOSITE MODULES	Application builder.	*IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide*
• PL/I considerations.		*IBM Series/1 PL/I User's Guide*
• FORTRAN considerations.		*IBM Series/1 FORTRAN IV User's Guide*
CREATING APPLICATION TASK SETS	Application builder.	*IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide*
• Planning application task sets.		*IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102
		IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor
• Building application task sets.		*IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide*
– PL/I considerations.		*IBM Series/1 PL/I User's Guide*
– FORTRAN IV considerations.		*IBM Series/1 FORTRAN IV User's Guide*
USING THE REALTIME PROGRAMMING SYSTEM UTILITY PROGRAMS	Utilities.	*IBM Series/1 Realtime Programming System: Operator Commands and Utilities*

For the following activity...	*The associated system step is...*	*Refer to Publications...*
CREATING A COMMUNICATIONS SYSTEM		*IBM Series/1 Realtime Programming System: Macro User's Guide—Communications*
INSTALLING TASK SETS	A series of steps using the utility DEFINE and COPY facilities and the install option on the start task set operator command.	*IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor*
EXECUTING TASK SETS	Direct—start task operator command or supervisor macro.	*IBM Series/1 Realtime Programming System: Operator Commands and Utilities* *IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor*
	Batch—job stream processor	*IBM Series/1 Program Preparation Subsystem: Batch User's Guide*
DEBUGGING TASK SETS	Error management.	*IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor*
	Messages and codes (Realtime Programming System).	*IBM Series/1 Realtime Programming System: Messages and Codes*
	Messages and codes (Program Preparation Subsystem).	*IBM Series/1 Program Preparation Subsystem: Messages and Codes*
	Messages and codes (PL/I).	*IBM Series/1 PL/I: Messages*
	Messages and codes (FORTRAN).	*IBM Series/1 FORTRAN IV User's Guide*
	Debugging.	*IBM Series/1 Realtime Programming System: Control Blocks and Debugging Guide*

This glossary contains only those terms that are referenced in this manual, and all definitions given are Series/1–oriented. You may also want to refer to the glossary contained in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

absolute load module. A combination of object modules having cross references resolved and prepared for loading into storage for execution at a specific address. This load module is not executable in the Realtime Programming System environment, but may be executable in a user-provided execution environment. It is an output of application builder phase 1 processing.

application builder. The subsystem program (operating in conjunction with the job stream processor under control of the Realtime Programming System) that prepares the object module output of language translators for execution. To create output that is executable in a user-provided environment, it can be used to create an absolute load module (an output of phase 1 processing). To create output that is executable under the Realtime Programming System, it can be used to create a task set (the final output from processing performed in phases 1, 2, and 3).

autocall library. Disk-resident composite module library used by the application builder to obtain those modules that contain a program that can resolve a reference from another program in a task set.

batch execution. Program execution initiated by the job stream processor in response to job control statements.

batch partition. The user partition that is used by and managed by the job stream processor.

batch program. Any of the subsystem programs or user task sets initiated by the job stream processor.

bound task set load module. A task set load module that has been bound to its execution environment. The bound task set load module contains the image of the partition at the completion of task set installation.

command. A character string that represents a request for action within the system from a source external to the system.

common control section. A type of control section that reserves an area of storage. It can be referred to by resident and overlay segments that are associated with a given primary or secondary segment within a task set.

composite module. Object modules (programs) structured into a resident segment and optional overlay segments. A composite module is in relocatable format; that is, its addresses can be modified to compensate for a change in its origin. It is the output of phase 1 and input to phase 3 of application builder processing.

consecutive data set. A collection of data, having a consecutive arrangement, to which the system has access.

control module. A set of tables and control blocks that contain control and parameter information pertaining to the task set. It is one of the modules produced by the application builder and subsequently included in the task set load module.

data set. A named collection of data which resides on a device.

data set definition (DSD). Describes and locates a data set being used by a task set. It exists in the using program in a DSD table data set in the task set library. The DSD is accessed when the data set is opened.

data set definition name (DSD name). The external name of a DSD table entry used within a task set to reference the data set described by that entry.

data set definition statement (DSD statement). A job stream processor control statement that allows the user to establish a connection between a data set or device and a DSD name used in a program.

data set definition table (DSD table). A table that contains parameters for data sets.

data set name (DS name). The term or phrase used to identify a data set. It is contained in the data set definition table of each task set referencing that data set.

device. A piece of mechanical, electrical, or electronic equipment used to contain data that is input to or output from the processor.

device line. The actual number of characters that can be displayed on one print line of the device. For example, the operator station line length is 72 characters, and the display station line length is 80 characters.

DSD environment–*See* environment.

DSD name–*See* data set definition name.

DSD statement–*See* data set definition statement.

DSDT or DSD table–*See* data set definition table.

editing session. A period of time beginning when the editor is invoked and ending when the editor has completed processing.

environment (data set environment). The data set definitions that are in effect at any point in time during a batch session.

environment list. A data set (or member) containing a group of data set definitions which make up a DSD environment.

external symbol dictionary (ESD). Control information, associated with an object or composite module, that identifies the external symbols in the module.

fixed line number. The line number assigned to a text record and associated with that text record for the duration of the editing session (unless specifically altered by the user).

fixed partition. A partition having a predefined beginning and ending storage address.

generation input stream. An input stream created by the generation program which, when executed, produces a system tailored to the user responses to the generation program questions.

generation program. The IBM-supplied program that is used at installation time in creating your Program Preparation Subsystem.

global area. An uninitialized portion of a partition accessible by any program of a task set in the partition at a given time. The same area may be used by other task sets that execute in the same partition. The size of the global area is determined by the collective sizes for the largest uniquely named (or unnamed) global section definitions. These definitions are declared by programs that make up a task set.

global control section. A type of control section that reserves an area of storage. It can be referred to by any primary or secondary program and their associated overlays within a task set. See also global area.

input stream. The sequence of job control statements and data submitted to an operating system through an input device designated for this purpose by the operator. Synonymous with input job stream, job input stream.

interactive. A realtime interface between a user and a program system.

job. A collection of related problem programs, identified in the input stream by a JOB statement followed by one or more EXEC and data set definition statements.

job input stream. *See input stream.*

job stream. *See input stream.*

job stream processor. The Program Preparation Subsystem component that reads and interprets job control statements and processes the requests made by those statements.

line. A string of characters accepted by the system as a single block of input from an operator station; for example, all characters entered before the carriage return key or the ENTER key is pressed. For the text editor, it represents the line number plus the text line.

line display range. That portion of a line to be displayed or printed out when a line is listed.

line length. Logical record length of lines being edited by the text editor.

noninteractive. An indirect interface between a user and a program system; for example, through a disk or diskette data set.

object module. The output of a single assembly or compilation containing one or more control sections—CSECTs. An object module is equivalent to a program.

object module data sets. Disk resident data sets that contain object modules.

object module library. A partitioned data set containing multiple object modules.

operator station. A device through which the primary interaction between the user and a program system occurs. It can be either a display station or any device that can be used as an operator station and is attached to the system through the ASCII Teletypewriter Adapter Feature.

overlay area. An area within a task set load module, associated with a given resident segment and used for execution of overlay segments that are located on disk.

overlay module. A structure that contains all overlay segments in a single task set. It is one of the modules produced by the application builder as part of a task set library.

overlay segment. A segment that resides on secondary storage and is loaded into the overlay area associated with its resident segment. All overlay segments in a composite module are associated with the resident segment in that composite module. A program in an overlay segment can call a program within the same overlay segment or a program in any resident segment.

partition. A segment of physical and addressable storage which may contain one task set at a time. A partition begins and ends on a 2KB boundary and has a unique numeric ID from 0 to 15. See also fixed partition and user partition.

partitioned data set. A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

prebind module. A module that contains the specifications used during task set installation to create the bound task set load module. It is one of the modules produced by the application builder as part of the task set library.
prebinding. The connection of a task set's control blocks (both internal and external) and resources before task set execution.

primary program. The first program executed under each task in the system. A primary program has a single entry point and must have a program header. A primary program is either reenterable, serially-reusable, or nonreusable and is only invoked by a start task request.

primary segment. The resident segment that contains the initial entry point of a task set. The entry point of a task set must be a primary program within the primary segment.

processor. (1) In hardware, the resource required to execute an instruction stream. (2) In software, a synonym for processing program.

processor storage. The storage provided by one or more processing units. This term pertains to physical locations in hardware devices.

program. (1) A named sequence of instructions that operates under the auspices of a task. (2) A program is the output of a single assembly or compilation, containing one or more control sections—CSECTs. A program is equivalent to an object module.

program preparation facilities. The subsystem components that are used to create user task sets; These components are the text editor, macro assembler, and application builder.

queue. A line or list formed by items in a system that are waiting for service.

resident segment. A segment that remains in primary storage for the duration of task set execution. A program in a resident segment may call a program in one of its overlay segments or a program in another resident segment.

resource. Any facility of the computing system or operating system required by a job or task, and including main storage, input/output devices, the central processing unit, data sets, and control or processing programs.

return code. A code used to influence the execution of succeeding steps in a job in the input stream. An indicator, which is passed from a batch program to the job stream processor, that reflects the status of the batch program at the time of its termination.

scheduling. The ability to imply that a task set should be started at a particular time of day or after a specified time interval.

secondary program. Any program other than the primary program of a task. A secondary program may have multiple entry points and may or may not have a program header. Secondary programs are invoked by a call request or by direct linkages, such as assembler branch instructions.

secondary segment. Any resident segment other than the primary segment of a task set.

segment. A structure containing one or more programs, which is a portion of a composite module or a task set load module.

source module. A collection of source statements which constitute the input to a language translator for a particular translation. These source statements may be created, modified, and listed using the text editor.

split screen. The division into sections of a display screen in a manner which allows two or more programs to use the display screen concurrently.

spooling. Writing a data set that is found in the input stream to secondary storage.

starter system. A supervisor in IPL format which supports the system generation process.

step. A request to the job stream processor to execute a program and, optionally, any accompanying statements defining data sets used by the program.

task. The dispatchable entity used by the supervisor to establish and track concurrent program execution within the system. Each task represents a single thread of execution through a program or set of programs. The first program executed under each task is a primary program. All others are secondary programs.

task set. A named collection of programs, data, and control blocks designed to execute within a partition. The program of a task set perform a related set of work and execute under one or more tasks.

task set library. A logical volume containing all of the data sets associated with a single task set. A task set library may also contain user data sets.

task set load module. A structure that contains all resident segments, their associated common and overlay areas, and the control module, for a single task set. A task set load module is loaded from a consecutive data set in the task set library into a partition when the request for the task set becomes the highest priority element in the partition queue. It remains in primary storage for the duration of task set execution. A task set load module is in absolute format; that is, its origin cannot be changed. See bound task set load module and unbound task set load module.

text editor. The program preparation facility that is used to create, modify, and list text modules. Text prepared using the text editor may be in the form of source modules, which may be input to the macro assembler, *or* text data, which may be input to a user program or one of the subsystem programs.

text module. A term used by the text editor to indicate the data (text) that may be created and maintained using the facilities of the text editor. This data is usually in the form of printable characters (for example, source modules or input data to a user program).

timer. A mechanism for defining an interval of time.

transient area. A main storage area used for temporary storage of transient programs.

transient program. Self-relocating program permanently stored on a system residence device and loaded into the transient area when needed for execution.

unbound task set load module. A task set load module that has not been bound to its execution environment. It is one of the modules produced by the application builder as part of a task set library. The unbound task set load module may be loaded into a partition without being prebound, or it may be used as input to task set installation to create a bound task set load module.

user partition. A partition that contains a user task set when in execution.

work data sets. The data sets and data set members used by the program preparation facilities as temporary work areas.

work stack. A list that is constructed and maintained so that the next information to be retrieved is the most recently stored information in the list; that is, a last-in-first-out or pushdown list. It is an area of unprotected main storage allocated to each task and used by the programs executed by that task.

Index

Program Preparation Subsystem:
Introduction
GC34-0121-0

READER'S
COMMENT
FORM

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Cut or Fold Along Line

What is your occupation?____
Number of latest Technical Newsletter (if any) concerning this publication: __________________
Please indicate your name and address in the space below if you wish a reply.

____

____

____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

GC34-0121-0

Your comments, please . . .

This manual is part of a library that serves as a reference source for IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

— Cut Along Line —

Fold Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

IBM Corporation
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold Fold

IBM Series/1 Program Preparation Subsystem: Introduction Printed in U.S.A. GC34-0121-0

IBM®

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

Program Preparation Subsystem:
Introduction
GC34-0121-0

READER'S
COMMENT
FORM

Cut or Fold Along Line

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

What is your occupation? ______________________________

Number of latest Technical Newsletter (if any) concerning this publication: ______________

Please indicate your name and address in the space below if you wish a reply.

____

____

____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

GC34-0121-0

Your comments, please . . .

This manual is part of a library that serves as a reference source for IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Cut Along Line

Fold | Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

IBM Corporation
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold | Fold

IBM Series/1 Program Preparation Subsystem: Introduction Printed in U.S.A. GC34-0121-0

IBM®

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N.E.
P. O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

IBM Series/1 Program Preparation Subsystem: Introduction Printed in U.S.A. GC34-0121-0

GC34-0121-0