UNISYS

BTOS II

Language Development

Programming Guide

Relative to
Release Level 2.0

Priced Item

August 1988
Distribution Code SA
Printed in U S America
5028707

UNISYS

BTOS II

Language Development

Programming Guide

Relative to
Release Level 2.0

Priced Item

August 1988
Distribution Code SA
Printed in U S America
5028707

Comments or suggestions regarding this document should be submitted on a User Communication Form (UCF) with the CLASS specified as 2 (System Software), the Type specified as 3 (DOC), and the product specified as the seven-digit form number of the manual (for example, 5028707).

About This Guide

The Language Development Software allows you to create executable run files using one or more languages or language tools (purchased separately).

This programming guide contains the following information:

- Language Development software installation information
- LINK, BIND, and LIBRARIAN command procedures
- general and troubleshooting information on linking programs
- descriptive and operational information for the Unisys Assembler and Assembly language used in the Unisys Family of Workstation applications

Who Should Use This Guide

This guide is for programmers. To understand some of the information in this guide, you must be familiar with the following:

- the BTOS Executive level operations
- the programming language your modules were written in (Pascal, FORTRAN, C, compiled BASIC, or Assembly)
- other programming tools (such as Forms, Font, and ISAM)

How To Use This Guide

If you are using Language Development Software for the first time, you should read section 1 and appendix B. They contain basic information you will need to understand and install the software.

In addition, if you scan the contents and review the topics before you start, you may find this guide easier to use. To find definitions of unfamiliar words, use the glossary; to locate specific information, use the index.

You may also want to review the compiler manuals to become familiar with the Math Server.

How This Guide is Arranged

This guide is divided into sections.

Section 1 presents a basic/conceptual overview of the software. Sections 2 through 5 contain general and procedural information on the Linker and Librarian; sections 6 through 11 describe the general operations and procedures for using the BTOS Assembler.

For software installation information, refer to appendix B.

For general troubleshooting information and suggested error message responses, refer to appendix A.

Additional technical information is included in appendices C through I.

Conventions

The following conventions apply throughout this guide:

- Where two keys are used together for an operation, their names are hyphenated. For example, **ACTION–GO** means that you press **GO** while holding down the **ACTION** key.
- The term BTOS refers to BTOS II.
- The term "character" includes spaces.
- Numbers are decimal except when suffixed with **h** for hexadecimal.
- "Memory address" refers to the logical memory address.
- Variable names are named according to a formal convention. The name of a variable should represent some of its characteristics.
- A variable name is composed of up to three parts: a prefix, a root, and a suffix. The following prefixes are used in this guide:

  b byte (8–bit character or unsigned number)

  c count (unsigned number)

  i index (unsigned number)

  n number (unsigned number, same as c)

p logical memory address (pointer: 32 bits consisting of the offset and the segment base address)

q quad (32–bit unsigned integer)

rb relative byte (a 16–bit offset from an arbitrary base address)

rg array of...

sb array of bytes, where first byte is the size

w word (16–bit)

- A prefix can be a compound. For example, the compound prefix **rbrg** indicates the position of an array relative to the beginning of the run file header.
- The root of a variable name can be unique to that variable, a commonly used root, or a combination of the two. Common roots are:

lfa logical file address

mp map

par paragraph

sa segment address

ra relative address

- A suffix identifies the use of the variable. The suffix used in this guide is Max. Max is the maximum length of an array or buffer (thus one greater than the largest allowable index). Examples of variable names are:

iProtoDescMax – the maximum SN index

rbrgrle – the offset of the array of relocation pointers from the beginning of the run file header

Related Product Information

For detailed information on BTOS, refer to your operating system reference documentation.

For an explanation of the BTOS Executive and its commands, refer to your Standard Software documentation.

For a complete description of the BTOS calls, refer to your system procedural interface documentation.

For a listing of BTOS and related application status codes, refer your status codes documentation.

For information and procedures on creating and editing forms, refer to your forms designer programming documentation.

For information and procedures on customizing an operating system and creating a debugger, refer to your system procedural interface documentation, and your Debugger documentation.

For more information about writing and compiling your programs, refer to the documentation of the language or compiler you are using. For information on the Math Server, refer to the Pascal or FORTRAN documentation.

Contents

Illustrations

Tables

BTOS II Language Development Overview

The BTOS II Language Development software adds the following programming tools to your workstation:

- **Linker**

  You use the Linker to join several object modules (machine code from a BTOS compiler, or from the Assembler) into a BTOS run file.

- **Librarian**

  You use the Librarian to create and modify libraries of object modules produced by the Assembler or BTOS Compilers, or libraries forms created by the Forms Designer.

- **Assembler**

  You use the Assembler to convert 8086 Assembly language programs to object modules.

- **Math Server**

  The Math Server allows more than one Pascal, FORTRAN or C application that uses floating point math to use the 80287 math coprocessor simultaneously without crashing the system.

  For information on the Math Server, refer to your compiler documentation.

- **Mouse Server**

  The Mouse Server contains the request and procedural interfaces for the 2-button and 3-button mouse and handles cursor control and tracking. For information on the Mouse Server, refer to your Standard Software documentation.

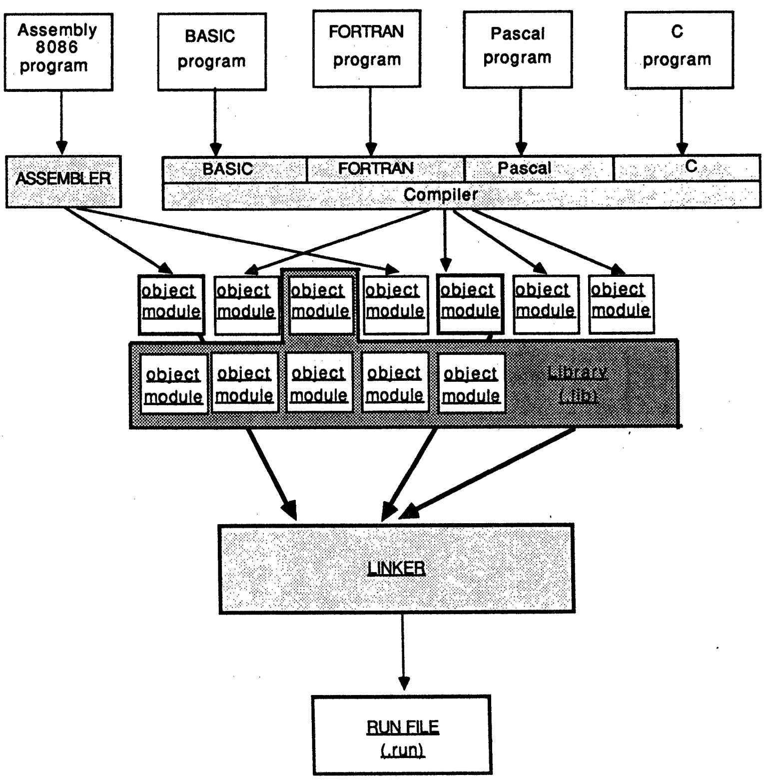
When you compile or assemble a program, the system translates the program into machine code. The resulting compiled or assembled program is called an object module.

You use the Linker to create a BTOS run file from object modules. When you link the run file, you can include any object modules, Forms, or ISAM files on your system.

The Librarian helps you to file the object modules. You can create libraries of related object modules and then link run files by entering the library name (rather than listing the object module names).

Figure 1–1 illustrates the relationship between the Linker, Assembler, and several other BTOS programming tools (which are available separately).

Figure 1–1 BTOS Programming Tools



Using the Linker

The Linker allows you to produce your application as a set of independently-compiled object modules that refer to each other by cross-module calls. Therefore, you can write the modules in different languages because the Linker resolves references to variables and entry points between different object modules.

In addition to writing modules in different languages, you can also use modules from any of several extensive libraries. However, all support libraries must be available at link time.

The Linker creates an executable run file that contains information the BTOS Loader uses to relocate the resultant program and initialize the processor.

The contents of BTOS object modules may contain any or all of the following: code, constants, or variable data. The Linker arranges the contents of a set of object modules into a memory image, typically with all code together, all constants together, and all variable data together. (This arrangement makes optimal use of the addressing structures of the processor.)

The Linker performs the following functions:

- builds a run file the BTOS Loader can load efficiently
- produces a single run file with any of several configurations (This run file can be one task of a multi-task application.)
- searches libraries to select the object modules that an application requires
- optionally constructs a run file containing overlays for use with the virtual code segment management facility

Linker Commands

You can use two different Executive commands to generate run files: LINK or BIND. LINK produces a version 4 run file. BIND produces a version 6 run file (compatible with protected mode BTOS versions), unless version 4 is specifically requested.

The Linker's Two Passes

The Linker makes two passes through the object modules being linked. On the first pass, the Linker reads the object modules, extracts external and public symbol information, and builds a symbol table. It checks the symbol table for unresolved external references. If they exist, the Linker consecutively searches the library list you specified in the LINK or BIND command form for object modules whose public symbols resolve the external references.

On the second pass, the Linker assigns relative addresses (relocating as necessary) to the object module data, and then it links the object modules, constructing a run file ready for the BTOS Loader.

Using the Librarian

The Librarian is a program development utility that creates and maintains libraries of object modules. A library has three uses:

- It can be a parameter in the Libraries field of the LINK and BIND command forms, specifying that the Linker should search the library for object modules that satisfy unresolved external references.
- It is a convenient unit for collecting several object modules and distributing them as a single file. The Librarian extraction facility, also available in the Linker, can be used to extract specific modules from the library.
- It is a convenient unit for collecting forms created with the Forms Designer. (Refer to your Forms Designer programming documentation.)

You can collect many object modules in a single library file.

You do not need to remember the names of the object modules in the library; the Linker's library search algorithm selects the required object modules from the library. You can extract individual object modules from the library by entering the object module name.

The Librarian performs the following functions:

- builds a new library when you specify a new library file name and the object modules for the file
- modifies an existing library file when you specify object modules to be added or deleted (This includes the replacement of an existing object module with a new object module that has the same name.)
- extracts one or more object modules from a library file when you specify the object module name in the extraction field
- produces a sorted cross-reference listing of the object modules, and of the public symbols in the library when you specify a cross-reference list file name

Library File Names

Your standard software uses a **.lib** suffix to assist in file management (to help identify library files). You can use this suffix, but the system does not require it.

The Librarian creates the library name for an added object module from the object module file name; it drops the volume, directory, and file prefix names and any suffix beginning with a period.

For example:

If the file name is [Sys]<Jones>Sort.obj, the library object module name is Sort.

If the file name is [Jones]<Working>Sort, the library object module name is Sort.

Object module names within libraries must be different; the Linker searches the library for the names of object modules that define public symbols.

Note: You cannot use **none** for the library name; you can, however, use the parameter **none** in the Linker command **[Libraries]** field to direct the Linker not to search libraries.

Cross-Reference Lists

If you specify a cross-reference list file name, the Librarian produces a list of the object modules and public symbols in the library. The cross-reference list has two parts: object module names referencing the public symbol(s) defined, and public symbols referencing the object module that defines it.

Using the Assembler

The information in this guide describing the Unisys Assembler and Assembly language is directed toward those who understand Assembly language reasonably well.

The Unisys Assembler generates object code that can be run on the 80186, 80286, and 80386 CPUs. The Assembler can also generate the 80286 extensions to the code that run on the B28, B38, and B39 workstations. You should use these extensions carefully; they cause Invalid Opcode Exceptions (INT 6) on the B24, B26, and B27 workstations.

(For information on determining which workstation your code is running on, refer to your operating system reference documentation, and to your system procedural interface documentation.)

Features and Characteristics

The Unisys Assembly language features a powerful instruction set, sophisticated code and data structuring mechanisms, strong typing (the ability to check that data usage is consistent with its declaration), a conditional assembly facility, and a macro language with extensive string manipulation capabilities.

This Assembly language differs from most other Assembly languages, which usually have one instruction mnemonic for each operation code (opcode). With Unisys' Assembly language, you can assemble a particular instruction mnemonic into any of several opcodes. The type of opcode depends on the type of operand.

Unisys' Assembly language is a "strongly typed" language, since you cannot have mixed operand types in the same operation (for example, moving a declared byte to a word register). You cannot inadvertently move a word to a byte destination, thereby overwriting an adjacent byte; nor can you move a byte to a word destination, thereby leaving meaningless data in an adjacent byte. However, if you must override the typing mechanism, there is a special PTR operation that allows you to do this (refer to section 8).

Some of the other features and characteristics of the Unisys Assembly language are summarized in the remainder of this section.

Segments

BTOS Assembly language programs are composed of segments in which each instruction and variable is created. Afterwards, all segments are then linked together.

At Assembly time, you can define as many segments as you wish, as long as each assembly module has at least one segment. Each instruction of the program and each item of data must lie within a segment. The following examples are some of the types of segments you can define:

- data segments
- stack segments
- main program segments (code)

Addressing

You can address operands in several ways using various combinations of base registers (BX and BP), combinations of index registers (SI and DI), combinations of displacement (adding 8–bit or 16–bit values to a base, index register, or both), and combinations of direct offset (16–bit addresses used without the base or index register).

Procedures

The Unisys Assembly language formalizes the concept of a callable procedure by providing explicit directives to identify the beginning and end of a procedure. Where other Assembly languages start a procedure with a label and end it with a return instruction, the Unisys Assembly language differs by defining a procedure as a block of code and data, starting it with a PROC statement, and ending it with a ENDP statement.

Macros

You can use the macro capability of the Assembler to define abbreviations for arbitrary text strings including constants, expressions, operands, directives, sequences of instructions, and comments. These abbreviations can accept parameters; they are also string functions that the system evaluated during assembly.

Consequently, you can collect the macro definitions in a file, which in turn can be included in other Assembly language source files using the $Include directive. Building a library of such macros allows you to invoke frequently used text strings using a concise, standardized definition within several different source files.

The macro facility also provides interactive assembly by means of a macro time console I/O facility.

Choosing the Right Language

As a programmer working with a Unisys Information Processing System, you have many different languages to choose from. The choice involves several considerations:

- Does the program require the unique business features of COBOL or the scientific features of FORTRAN?
- Is an interpretive language suitable?
- Will the system programming and data structuring facilities of Unisys Pascal be particularly valuable in the program to be written?
- Should you you divide the program into different parts, write the different parts in different languages, and then combine them with the Linker?

  If the program (or program part) requires direct access to processor registers and flags, then Assembly language is an appropriate choice. Assembly language is also a better tool than other languages when memory usage and object code efficiency are more important than development speed and programmer productivity.

  However, you rarely write an entire application system in Assembly language. You should determine those parts in which direct access to machine features, efficiency, and memory usage are overriding concerns, write and use those parts in Assembly language, and then write the remainder of the application in a high-level language.

Creating BTOS Run Files with the Linker

The Linker separates object modules by component, combines like components for efficiency, and then creates an executable run file. The run file contains a memory image and other information that the BTOS Loader uses to relocate the resultant program, and to initialize the processor for program execution.

A BTOS run file is a memory image of tasks in the BTOS Loader format. The BTOS Loader can usually load it with a single disk access and data transfer.

The BTOS Run File Format

The BTOS run file format consists of the following components:

- a file header
- relocation data
- a memory image
- optional virtual code segments

The Linker supports task images as large as the processor's full address space. You can link up to 256 object modules; each object module can contain a code segment. You can also use the run file with various memory configurations, or as one task of a multi–task system.

Run File Header

The run file header performs the following functions:

- describes the run file
- provides initial values
- provides an array of pointers that allows the BTOS Loader to relocate the run file in memory

Relocation Data

You do not have to specify the eventual memory address of the task. The Linker computes and includes information in the run file that the BTOS Loader uses to relocate the task to any desired memory location.

The BTOS Loader uses this information when it brings the task image into memory. A single run file can then be used with various size operating systems, or be used with other diverse tasks of multi–task applications.

Memory Image

The run file memory image contains the code that is resident during program execution. The Linker does not assign absolute memory addresses (the BTOS Loader assigns these). The memory image includes a checksum that the task loader verifies.

Virtual Code Segments

Each unit of code that the BTOS Loader brings into memory is called a virtual code segment. The BTOS Loader brings virtual code segments into memory only when required. The system then overwrites the segments as it needs their address space for other virtual code segments.

You can write your program with as much code as required, as if all code were simultaneously resident in memory. The BTOS Loader initially loads only the resident code. When the program calls any subroutine in a virtual code segment, the BTOS Loader brings the segment into memory.

You can use a maximum of 256 virtual code segments, each no greater than 64 Kb. This allows up to 16 Mb for code.

A memory pool is used to hold the virtual code segments; all segments that can fit in the pool can be simultaneously resident in memory. You specify the pool size during execution.

Calls to entry points in virtual code segments go through an indirect table; calls to a code segment that is in memory take only a single instruction.

Library Search Algorithm

After building a symbol table during its first pass, the Linker then runs through all the symbols, checking to see whether any of them occur in the first library listed for searching. If it finds a symbol declared in a library module in the library, it extracts that module from the library and links it into the program. The extracted library module can also contain yet other undefined symbols.

The Linker cycles over the entire list of symbols, old and new, comparing them to the first library until it can extract no further library modules. It then continues to the second and subsequent libraries and repeats this process.

When the Linker completes the search of the last library, it goes back to the first library and again searches for undefined symbols. In this manner, it repeatedly cycles through all the libraries until it cycles through without extracting any new modules. At this point, it stops and reports any symbols that remain undefined.

Note: If the same public symbol is defined in more than one library, and if that symbol is declared external in an extracted library module, the definition used is not necessarily in the first library listed for searching. The Linker starts from the point at which it extracted the module, continues to the next library, and then extracts the first definition it encounters.

Segment Element Names and Classes

In the example in figure 2–1, three object modules are to be linked. They are listed in the Object modules field of the Linker command form in the following manner, using single spaces between the names:

Mod1.obj Mod2.obj Mod3.obj

Mod1.obj was written in one language; Mod2.obj and Mod3.obj were written in another. Each of these object modules consists of several segment elements, each of which the programmer declared public at the source level. All of these object modules have segment elements that contain code, data, constants, and stack, although this is not true of all object modules.

Each module segment element has both a name and a class. In high level languages, the compiler assigns name and class. In figure 2–1, a slash separates the name and class of each segment as follows:

Data/Data

Mod1 code/Code

Many compilers assign names to segment elements that are identical to the segment element class (for example, Data/Data). Usually, the code segment element carries the name of the module: in Mod1.obj, the Mod1 segment element is of class Code. Most compilers append the class name as part of the code segment element name, which in this case results in Mod1_code.

The most common classes are Code, Data, Const, and Stack. A compiler always arranges the segment elements by class and in a specific order.

With 8086 Assembly, you have more control over what the Linker does than you do when you use a compiled language. You can assign any name to any segment element and to any class. You can define more than one of a class and place them in any order within the module.

Figure 2-1 How the Linker Builds a Run File

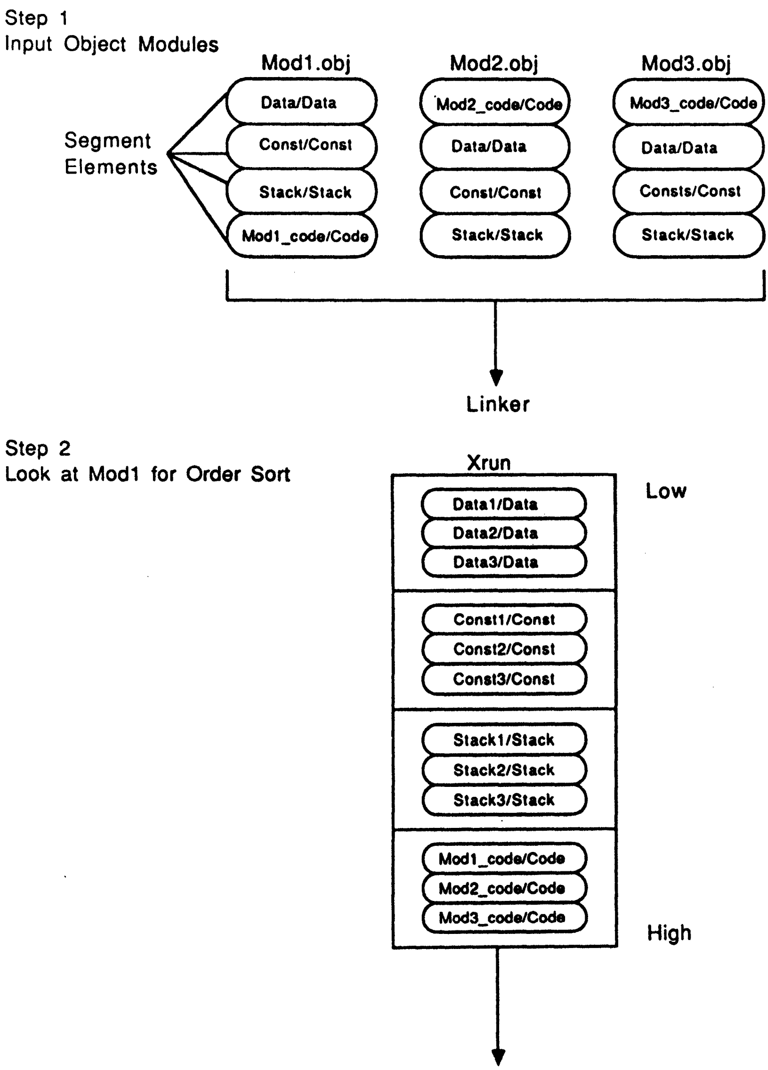
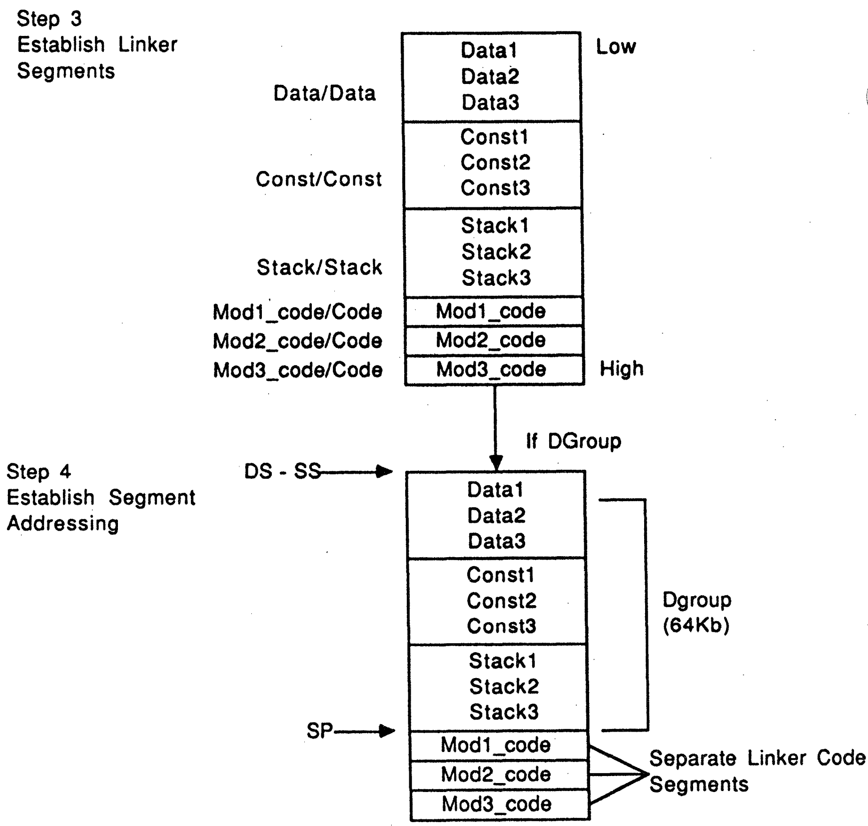


Figure 2–1 How the Linker Builds a Run File (continued)



Creating Linker Segments

After the Linker resolves all external references in the modules, it builds the run file. Starting with the first module listed (Mod1.obj), it takes the first segment element in that module, creates a category for its class, and places the segment element in that category. It then creates a second category for the second class of segment element that it encounters, and so on through the first module.

In the example in figure 2–1, the result is the creation of four categories arranged in the same order as the segment element classes in Mod1.obj: data, constants, stack, and code. These categories eventually become Linker segments.

Having pulled apart Mod1.obj in this way, the Linker goes on to Mod2.obj. It takes each segment element in Mod2.obj, examines its class, and places it in the Linker segment already created for that class. If there is no Linker segment for that class, the Linker creates a new one for it at the end of the Linker segment list.

When the Linker has sorted the parts of all three modules, the result is as shown in step 2 of figure 2–1.

Note: Linker segments are ordered by class in the same order that appears in the first module listed. Thus, you can impose an ordering template on the Linker by writing an Assembly language module that does nothing except declare segment elements in the desired class order. You then place this module first in the list of modules to be linked. This template object module is often called First.obj.

Combination Rules

The model is incomplete without an indication of how the Linker combines or superimposes segment elements to form Linker segments.

In most cases, the Linker appends one segment element to another as it goes through the modules, and does not distinguish boundaries between segment elements from one module to the next. This is true for data and constant segment elements.

For stack segment elements, the Linker combines them by overlaying them with their high addresses superimposed, but adds their lengths together. It then forces the total length of this aggregate stack segment to a multiple of 16 bytes. You can see this arrangement in figure 2–2. The fact that high addresses are superimposed is unimportant unless you have created a label at the high end of one of the stack segment elements. In this case, the label floats to the high end of the aggregate stack.

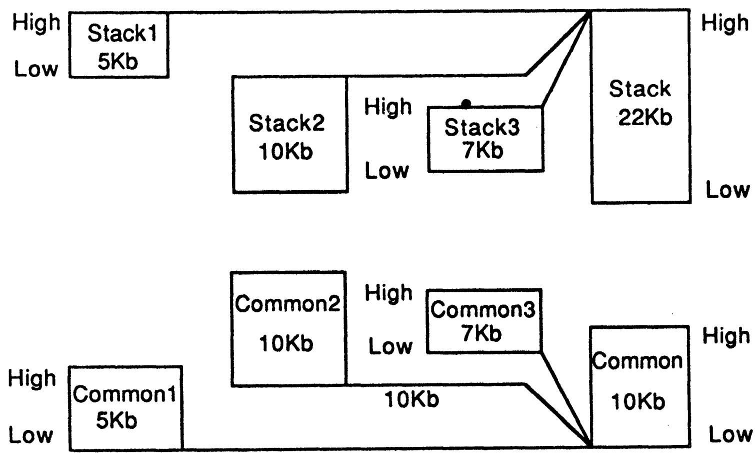
Compilers construct stack segments automatically. However, if your entire program is written in Assembly language, you must define an explicit stack segment. (Refer to section 11 for details.)

Segment elements that have the combination attribute COMMON in Assembly language are special. When COMMON segment elements are combined, they are overlaid with low addresses superimposed. The length is that of the largest element, as shown in figure 2–2.

The Linker places the code segment elements together, but it does not combine them unless they have identical names and are in the same class. (This rule applies to all segment elements, but it is most obvious with code segment elements.)

Figure 2–2 shows how the Linker combines the stack and COMMON segment elements shown in step 3 of figure 2–1.

Figure 2–2 Combination of Stack and COMMON Segment Elements



Summary of Segment Ordering

All public segment elements having the same segment name and class name are combined in the order the Linker finds them. Similarly, all segment elements having the same class name are placed together in the order the Linker finds them.

The Linker places all the first class segment elements in the run file. Then it places all the second class segment elements in the run file, and so on.

A group definition does not affect segment ordering. A group definition asserts that all segments in a group are contained within a 64 Kb region in the run file. This grouping is required if the data in the group is addressed using a single value in a segment register. In version 6 run files, all segments in a group must be contiguous or the Linker stops with an error message.

Alignment Attributes

Segment elements have alignment attributes. Most compiled languages assign these attributes automatically, but in Assembly language, you assign them explicitly. (Refer to section 6 for details.)

A segment can have one of the following alignment attributes:

- byte (a segment that can be located at any address)
- word (a segment that can be located only at an address that is a multiple of two)
- paragraph (a segment that can be located only at an address that is a multiple of 16)

The Linker packs segments containing data and code end-to-end. Alignment characteristics can cause a gap between the segments. The Linker adjusts the relative addresses in the segments accordingly.

Addressing Linker Segments

The Linker establishes the way in which the hardware addresses Linker segments when the program runs. In most cases, a group has been defined in the program.

A group is a named collection of Linker segments addressed at run time with a common base address: you can use 16-bit offset addressing throughout the group. All the locations within the group must be within 64 Kb of each other.

A program typically contains a group called DGroup, which consists of data, constants, and stack. (The medium-model compiled languages use DGroup. In Assembly language, you can define whichever groups you want, or none.) For DGroup, the hardware segment register is DS. Stack segment (SS) has the same value.

In a version 4 run file, other portions of the program can fall between the beginning and the end of a group, as long as the distance from the beginning to the end of the group does not exceed 64 Kb.

In a version 6 run file, all the Linker segments must be contiguous. The Linker combines all the segments of a group into one segment which is addressed with one selector. The base address is loaded into a descriptor whose selector is loaded into a segment register. (For a version 4 run file, the base address, in 16-byte paragraphs, is loaded directly into a segment register.)

The example in figure 2-1 contains DGroup, which is shown in step 4. This type of run file retains information about where the data, constant, and stack Linker segments begin and end. The value of the SS register is set equal to that of DS. SP is set to equal the highest address in the group, as shown in the figure.

Limits

In general, the maximum size of a linkable program and the speed at which the link takes place are directly related to the memory available on the system and inversely related to the number of public symbols in the program.

Structure of Run File Headers

The run file header that the Linker produces contains a variety of information describing the file.

The version 4 and version 6 run file header formats are shown in table 2–1. Keep in mind that while the current loader successfully handles all version 4 run formats, only the latest version 4 run format is being described here.

For offsets 0 through 36, the headers are similar except for the field names at offsets 14 and 22 (version 4 uses the sa prefix; version 6 uses the sn prefix). Offsets 30 through 86 are version 6 only.

The wSignature and ver fields (offsets 0 and 2) identify the run file and its version. The cpnRes field (offset 4) gives the run file size, excluding overlays.

The next four fields (offsets 6 through 12) provide information about relocation data in the file. The relocation directory is an array of locators the operating system uses in relocating the file. Table 2–2 shows the structure of these locators.

At offsets 14 through 22, the Linker assigns the initial values for the stack and code segments.

At offsets 24 through 28, the Linker locates the relocation directory and identifies the number of overlays.

The information at offsets 34 through 37 pertains to correction by the Linker and the operating system code of a known hardware problem with the IDIV instruction on early versions of the 80186 processor.

The Linker uses the fields qbMinData and qbMaxData (offsets 38 and 42, version 6 only) to size partitions on the 80186 processors, and to determine a limit on how much 80286/80386 processor data space a process can control.

The rbRgProtoDesc and iProtoDescMax fields at offsets 46 and 48 (version 6 only) contain the offset and maximum index of the prototype local descriptor table (LDT). The 80286/80386 loaders refer to this prototype data structure in building an LDT.

Table 2-3 shows the prototype LDT structure. The first field, limit, is the segment limit. The second, lfaLow, is the logical file address (lfa) of the segment. Since the lfa is a 24-bit quantity, the next field, lfaHi, supplies the high 8 bits of this address. The field at offset 5, bAccess, identifies the segment type.

The fields at offsets 50 through 58 (version 6 only) resolve issues involved in creating a run file that can run in both real mode (80186, 80286, and 80386 processors) and protected mode (80286 and 80386 processors only). Different types of addressing are used.

A version 6 run file uses call gates and global pointers to address certain operating system structures in protected mode on the 80286 and 80386 processors. The two fields at offsets 50 and 52 allow the version 6 file to be converted to the flexible additive address mechanism that must be used for such addressing if the task is to be run in real mode.

The fields at offsets 54 through 58 describe a table that maps each of the 80286 and 80386 protected mode selectors to a real mode segment address (SA).

The next six fields (offsets 60 through 68) separately identify and describe the code, data, and stack portions of a version 6 run file that runs in real mode on a variable-partition operating system.

At offsets 70 through 86 (version 6 only), several items are declared that simplify routine operations. The lfaSbVerRun field allows the operating system to find the version number in the run file so that a utility can change the number without relinking.

The dateTime stamp allows the Debugger to compare a symbol file to a run file, and to report an error if there is a difference.

The cModify field allows a count to be kept of the number of times a run file has been modified.

The qbMinCode and qbMaxCode fields pertain to the use of virtual memory. They indicate to the operating system the approximate size of the working set in bytes.

Table 2-1 Version 4 and Version 6 Run File Header Formats

Offset	Field	Size (bytes)	Description
0	wSignature	2	Run file signature
2	ver	2	Run file format version
4	cpnRes	2	Run file size
6	irleMax	2	Maximum relocation entry index
8	cparDirectory	2	Relocation directory size
10	cparMinAlloc	2	Minimum memory array size
12	cparMaxAlloc	2	Maximum memory array size
14	snStack	2	Initial stack segment (version 6)
	saStack	2	Initial stack segment (version 4)
16	raStackInit	2	Initial stack offset
18	wchksum	2	Run file checksum
20	raStart	2	Initial code offset
22	snStart	2	Initial code segment (version 6)
	saStart	2	Initial code segment (version 4)
24	rbrgrle	2	Relocation directory offset
26	iovMax	2	Maximum overlay index
28	snMainDs	2	Initial data segment, large model (version 6)
	Fs	2	Constant 0FFFFh (Version 4)
Version 6 Only:			
30	Fs	2	Constant 0FFFFh
32	maskOptions (veralt)	2	Run File Mode
34	rbIdiv	2	Idiv table offset
36	cIdiv	2	Size of idiv table

Table 2-1 **Version 4 and Version 6 Run File Header Formats** (continued)

Offset	Field	Size (bytes)	Description
38	qbMinData	4	Minimum virtual data partition size
42	qbMaxData	4	Maximum virtual data partition size
46	rbRgProtoDesc	2	Prototype descriptor table offset
48	iProtoDescMax	2	Maximum prototype descriptor index
50	rbRgRqLablE	2	Resident request fixup table offset
52	iRqLablEMax	2	Maximum resident request fixup index
54	rbMpSnSa	2	SN to SA translation table
56	iSnMax	2	Maximum SN index
58	snFirst	2	First prototype descriptor SN
60	slCode	2	First code segment selector
62	cSlCode	2	Count of code segments
64	slData	2	First data segment selector
66	cSlData	2	Count of data segments
68	slStack	2	Stack segment selector
70	cSlStack	2	Constant 1
72	lfaSbVerRun	4	File address of sbVerRun
76	dateTime	4	Time stamp
80	cModify	2	Modify count
82	qbMinCode	4	VM hint information
86	qbMaxCode	4	VM hint information

Table 2-2 **Address Structure**

Offset	Field	Size (bytes)
0	ra	2
2	sa	2

Table 2-3 **Prototype Descriptor Structure**

Offset	Field	Size (bytes)
0	limit	2
2	lfaLow	2
4	lfaHi	1
5	bAccessp	1
6	reserved	1
7	reserved	1

Using the LINK or BIND Command

The Linker combines object modules (files produced by high level language Compilers and the Assembler) to build run files (memory images of tasks linked into the BTOS Loader format).

When you use the LINK or BIND command to create a run file, the Linker performs the following operations:

- resolves references from one object module to variables and entry points of other object modules
- searches CTOS.lib and any additional libraries you specify to select the object modules necessary to satisfy unresolved external interfaces
- builds a run file the BTOS Loader can load efficiently
- computes information the BTOS Loader uses to relocate the loaded task to any memory location, and includes the relocation information in the run file
- constructs run files containing overlays for use as virtual code segments
- creates a list file that contains an entry for each segment and shows the relative address and length of the segment in the memory image

  You can direct the Linker to include public symbols and line number addresses in the list file.
- creates a symbol file

Note: Run files are limited to 1024 public symbols and 256 segments; object modules are limited to 256 publics and 256 externals.

LINK and BIND Command Forms and Parameters

When you select the Executive BIND command, the system displays the BIND command form as shown in figure 3–1. The BIND command activates the Linker to create version 6 run files, or create version 4 run files if you specifically request them.

When you use the Executive LINK command, the system displays the LINK command form as shown in figure 3–2. The LINK command creates version 4 run files, and is provided for use with older automated Submit programs that generate run files requiring the LINK command.

With either command, you must enter parameters in the **Object modules** and **Run file** fields (refer to Linking a Run File, in this section).

Both commands have default parameters for the fields that start and end with brackets (for example, **[List file]**). You can leave the fields blank to accept the defaults or enter a parameter to override the default. Refer to table 3–1 for information (including defaults) on each bracketed field.

Figure 3–1 BIND Command Form

```
Bind
  Object Modules
  Run file
  [Map file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max array, data, code]
  [Min array, data, code]
  [Run File Mode]
  [Version]
  [Libraries]
  [DS allocation]
  [Symbol file]
```

Figure 3–2 LINK Command Form

```
Link
  Object Modules
  Run file
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max memory array size]
  [Min memory array size]
  [System build?]
  [Version]
  [Libraries]
  [DS allocation?]
  [Symbol file]
```

Table 3–1 **LINK/BIND Options**

Field	Action/Explanation
[List file]	Field appears for LINK command only. The default directs the Linker to derive the map file name from the run file name. The Linker drops the .run suffix (if any) and adds a .map suffix. For example: If your run file name is Prog.run, the default map file name is Prog.map. If your run file name is [Dev]<Jones>Main, then the default map file name is [Dev]<Jones>Main.map. To specify a different map file name, enter the name.
[Map file]	Field appears for BIND command only. The default is the same as for **[List file]**. To specify a different map file name, enter the name.
[Publics?]	The default (no) directs the Linker not to include public symbols in the map file. Enter y to direct the Linker to add a list of public symbol relative addresses to the map file. The Linker sorts the publics by name (alphabetically) and address (numerically) as shown in table 3–2.
[Line numbers?]	The default (no) directs the Linker not to include a list of line numbers and addresses in the map file. If your object modules contain line numbers, enter y to direct the Linker to add a line number address list to the map file as shown in table 3–3.

Table 3–1 **LINK/BIND Options** (continued)

Field	Action/Explanation
[Stack size]	The default directs the Linker to use the Compiler or Assembler input (in the object modules) to estimate the stack size. The Compiler/Assembler input normally results in a stack size larger than the actual requirement; however, your program can contain features that cause the Linker to undercompute the required stack size. For example, Compiler/Assembler input for a program with many recursive procedures can cause the Linker to underestimate the stack size. To override the Compiler or Assembler input, enter a stack size (an even decimal number of bytes).
[Max memory array size]	Field appears for LINK command only: the default is one. To leave data space above the highest memory address, enter (in decimal) both the maximum memory array size and the minimum memory array size. Figure 3–3 shows the normal memory configuration when BTOS loads a run file; figure 3–4 shows the memory configuration when you specify the memory array size. **Note:** If the minimum size you specify leaves insufficient room for the task, an error message appears when BTOS fails to load the task. To make sure the task loads low (with a maximum data space above the task), set the minimum to 0 and the maximum to 1000000.
[Min memory array size]	Field appears for LINK command only: the default is zero. Refer to [Max memory array size].

Table 3-1 **LINK/BIND Options** (continued)

Field	Action/Explanation
[Max array, data,code] [Min array, data,code]	Fields appear for BIND command only: the default is the minimum space allocated. To override the default, separate entries with spaces. For maximum allocations, specify **0 0 0**. For maximum and minimum array, fill in the first parameter in each field to leave data space (the memory array) above the highest memory address of a task. For maximum and minimum data, specify the amount of short-lived memory the application will use. Maximum and minimum code are not implemented at this time.
[System build?]	Field appears for the LINK command only. The default directs the Linker not to make a system build. Enter y to build a custom operating system. **Note:** If you enter y and specify overlays in the object module field, the system creates the overlays but not the related data structures. For information on entering overlays, refer to Linking a Run File, in this section. Refer to the system build information in your operating system reference documentation.
[Run File Mode]	Field appears for the BIND command only. The default, real, directs the Linker to make an entry in the run-file header that specifies that the run file is real mode. To override the default, you can enter one of the following single-word options in this field: **Yes** Reserved for use by Unisys.

Table 3-1 **LINK/BIND Options** (continued)

Field	Action/Explanation
[Run File Mode] (continued)	**No** Reserved for use by Unisys **V4** V4 generates a Version 4 run file. **Protected** Protected indicates that the run file can run in protected mode and uses the local descriptor table (LDT). **HighMemProtected** HighMemProtected is meaningful only if your system contains a Mode 3 DMA device. Enter this parameter if you know that your run file is capable of running in the top 8 Mb of memory, capable of running in protected mode, and uses the local descriptor table (LDT). If you do not use this parameter, the system will only load the code portion of the run file in the top 8 Mb of memory, but only then if it was not loaded remotely over B-NET. **GDTProtected** GDTProtected indicates that the run file can run in protected mode and uses the global descriptor table (GDT). **HighMemGDTProtected** HighMemProtected is meaningful only if your system contains a Mode 3 DMA device. Enter this parameter if you know that your run file is capable of running in the top 8 Mb of memory, capable of running in protected mode, uses the global descriptor table (GDT). If you do not use this parameter, the system will only load the code portion of the run file in the top 8 Mb of memory, but only then if it was not loaded remotely over B-NET.

Table 3-1 **LINK/BIND Options** (continued)

Field	Action/Explanation
	LowDataGDTProtected LowDataGDTProtected indicates that the run file's data should be made accessible to real mode programs. The run file can run in protected mode and uses the GDT. This option is generally used only by special operating systems services that return pointers to their data, such as the bitmapped video service. **SuppressStubs** If you enter SuppressStubs and you have overlays in your object module list, the Linker does not generate the data structures for use by the virtual code segment management (for example, RgStubs). If you do not have overlays in your object module list, this option has no effect. For more information on LDT, GDT, and protected mode programs, refer your protected mode programming documentation. **CodeSharingServer** You use this option if you want one server to perform multiple tasks while executing the same code. If you choose this option, you may not then deallocate initialization code for reuse as part of short-lived memory. **HighMemCodeSharingServer** HighMemCodeSharingServer works only if your system contains a Mode 3 DMA device. Enter this parameter if you want the server to run in the top 8 Mb of memory, and to perform multiple tasks while executing the same code. If you choose this option, you may not then deallocate initialization code for reuse as part of short-lived memory.

Table 3-1 **LINK/BIND Options** (continued)

Field	Action/Explanation
	If you do not use this parameter, the system will only load the code portion of the run file in the top 8 Mb of memory, but only then if it was not loaded remotely over B-NET.
	Conditional Protected
	This option makes the decision to run in protected mode conditional upon the version of the OS. If the run file's version is older than the OS version, it will run in real mode; otherwise, it will run in protected mode. To use this option, you enter the parameter along with the version number of your OS.
[Version]	The default directs the Linker not to add a version to the run file header.
	To specify a version, enter an alphanumeric string. If the version has embedded spaces, surround your entry with single quotes.
	Note: If you are linking an operating system, you should specify a version to avoid an unresolved external error for sbVerRun.
	The Linker: - adds the prefix VER to your entry - places the version in the first run file sector - defines sbVerRun as your string preceded by a single byte containing the string length
	For example: If you enter 1.0, the run file header contains 'VER 1.0', and sbVerRun is the number 3 followed by the ASCII characters 1, ., and 0.
	You can use the Executive DUMP or VERSION commands to display the version number from the run file header.

Table 3-1 **LINK/BIND Options** (continued)

Field	Action/Explanation
[Libraries]	The default directs the Linker to search [Sys]<sys>CTOS.lib and any extensions such as CTOSToolkit.lib to satisfy unresolved external interfaces. By default, the Linker appends the versions of libraries specified in this parameter to the run file. To override this default, enter **NoReport**. To suppress all library searches, enter **None**. To direct the Linker to search library files in addition to CTOS.lib, enter the file name(s). Separate the names with single spaces. The Linker always searches [Sys]<sys>CTOS.lib last, even if you specify a different CTOS.lib. To suppress the use of [Sys]<sys>CTOS.lib only, enter **None** at the end of the library list. To link object modules from libraries into overlays, you must name the object modules in the **Object Modules** field (refer to Linking a Run File, in this section). The Linker links the object modules from the **[Libraries]** field in the resident portion of the task. If duplicate definitions appear, the Linker defaults to the first definition and creates a multiply-defined public.
[DS allocation?]	The default (yes) directs the Linker to locate DGroup at the end of a 64 Kb segment addressed by the DS register. Therefore, the last byte of DGroup is at DS:0FFFF. This enables allocation of memory at run time that is addressable using DS. This field applies only if your task uses a single value in DS during execution and includes the group DGroup, with DS equal to DGroup.

Table 3-1 **LINK/BIND Options (continued)**

Field	Action/Explanation
	Enter **n** if you want no DS allocation (for modules in most languages).
	Note: If you include a Pascal or FORTRAN module in the object list or by reference in a library, the default value should be used.
[Symbol file]	The default directs the Linker to derive the symbol file name from the run file name. The Linker drops the .run suffix (if any) and adds a .sym suffix.
	For example:
	If your run file name is Prog.run, the default symbol file name is Prog.sym.
	If your run file name is [Dev]<Jones>Main, the default symbol file name is [Dev]<Jones>Main.sym.
	To specify a file name for the run file symbol table, enter the file name.
	To direct the Linker not to create a symbol file, enter **[NUL]**.

Table 3–2 **Map File Public Symbol Lists (Sample)**

Publics by name	Address	Overlay
BSRUNFILE	076B:3630h	Res
BSVIDCLEARMARK	0593:0682h	Res
BSVIDEO	076B:36CCh	Res
BSVIDMARK	0593:0604h	Res
BSVIDTURNOFFCURSOR	0593:06E5h	Res
BSVIDTURNONCURSOR	0593:06A4h	Res
CBREC	076B:376Ch	Res

Publics by value	Address	Overlay
BSVIDMARK	0593:0604h	Res
BSVIDCLEARMARK	0593:0682h	Res
BSVIDTURNONCURSOR	0593:06A4h	Res
BSVIDTURNOFFCURSOR	0593:06E5h	Res
BSRUNFILE	076B:3630h	Res
BSVIDEO	076B:36CCh	Res
CBREC	076B:376Ch	Res

Note: In the public symbol list Address column, the h means hexadecimal; this is the standard processor segment-plus-offset addressing structure. Addresses are relative to the beginning of the file and are subject to fix-up at load time.

In the public symbol list Overlay column:

- Res means the symbol is resident
- an integer (n) means the symbol is in the nth overlay
- Abs (absolute) means the symbol has a specified place in memory

Table 3–3 **Map File Line Number List (Sample)**

105	0000:0000h	106	0000:0003h	107	0000:0023h
108	0000:00B2h	109	0000:0092h	110	0000:00AFh
111	0000:00EBh	112	0000:00EDh	113	0000:00F2h
114	0000:0114h	115	0000:011Eh	116	0000:0123h

Figure 3-3 Real Mode Normal Memory Configuration

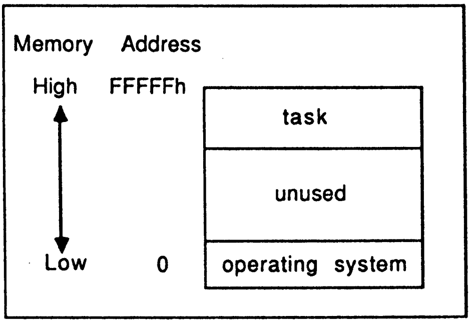
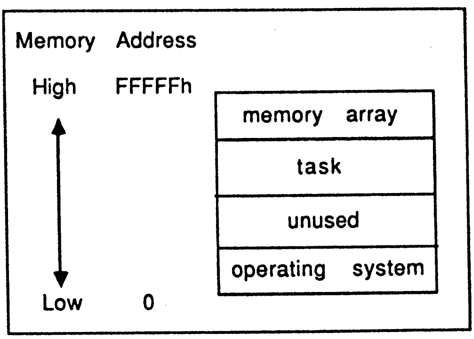


Figure 3-4 Real Mode Memory Configuration with Memory Array Size Specified



Note: In protected mode, the OS does not occupy the location shown in figures 3-3, 3-4.

Linking a Run File

To link a run file, use the following procedure:

1 At the Executive command prompt, type **LINK** or **BIND**.

2 Press **RETURN**.

The system displays the LINK or BIND command form as shown in figure 3-1 or 3-2; the highlight is on the **Object modules** field.

3 Enter the object module name(s), formatting the entries as follows:

- For individual object modules, enter the object module names, separated by single spaces.

  For example:

  a.obj b.obj 1.form

  where **a.obj** and **b.obj** are object modules; **1.form** is a form created by the Forms Designer (refer to your Forms Designer programming documentation).

- To extract object modules from a library file, enter the library file name followed by the object module names in parentheses, separated by single spaces.

  For example:

  Filename.lib (module1 module2)

  where **Filename.lib** is the library file name; **module1** and **module2** are the object module names.

  Note: Do not use a space between the opening parenthesis and the first module name.

- To use object modules as overlays for virtual code segments, append /O (a slash followed by the letter O) to the first module in each overlay. The /O is case-insensitive.

For example:

A.obj B.obj/O Z.lib(W X) D.obj/O

A.obj is the resident portion (it can include code and data); **B.obj/O Z.lib(W X) D.obj/O** is the nonresident portion consisting of two overlays: **B.obj Z.lib(W X)** and **D.obj**.

Note: List all other modules before you list the overlays.

For more information on virtual code segments, refer to your operating system reference documentation.

4 In the **Run file** field, enter a name for the run file.

   Standard BTOS software uses a **.run** suffix to assist in file management (to help identify run files). You can use this suffix, but the system does not require it.

5 Complete optional fields or accept the default values.

   For information on optional fields, refer to table 3–1.

6 Press **GO**.

   For information on error or warning messages, refer to appendix A.

   If the Linker displays the message **There were X errors detected**, you should examine the map file.

Program Memory Requirements

Determining the actual amount of memory that a run file needs is important for many reasons. For example, it allows the user to minimize the partition size that the program requires when executed under the Context Manager.

The memory requirement depends on these considerations:

- size of the data segment (for example, stack plus constants plus variables)
- size of the resident code (both the code written by the programmer and the code extracted from libraries)

- size of the overlay area, if swapping is used
- extra memory allocated at load time (the memory array) or later (by calls to AllocMemorySL or AllocMemory LL)

Run–Time Library Code

For compiled languages like Pascal, even a very small program requires the language run–time library, as well as associated support code from CTOS.lib. Consequently, almost all programs require 20 Kb to 40 Kb of space for run–time library code. The largest component usually is Sequential Access Method (SAM) code. Code from the run–time library is included in the map file.

Resident Programs

A resident program is one that is fully loaded into memory prior to execution. It contains no overlays and it stays in memory throughout execution.

You can read the memory required for resident programs directly from the map. The size is the stop address of the last segment (usually MEMORY) listed in the map. This number is the hexadecimal count in bytes from the first byte of the first segment.

Swapping Programs

A swapping program contains a resident program and overlays. BTOS loads the resident part of a swapping program into memory prior to execution, and loads the overlays during execution as they are needed.

Swapping programs should usually be sized on the stop address of the last segment of the resident portion, with the size of the required swap buffer added in.

Programs that Allocate Memory

To size a program that allocates memory, enter the maximum amount of memory that will be allocated in the appropriate field of the LINK or BIND command form. For programs that do DS allocation (for example, Pascal programs that use the New function), you add the extra amount of DS required to the allocated amount of memory. Use of the memory array is subject to the availability of a minimum amount of memory. (Refer to The Memory Array, in this section.)

Linker Map and Symbol Files

The Linker generates a map file that contains the following information for each object module or segment in the memory image:

- name
- relative address
- length
- public symbol values (if you select the Publics option)
- line numbers and addresses (if you select the Line number option)

Note: The starting addresses are offsets, not absolute addresses. The offsets are relative to the base memory address when BTOS loads the run file.

Reading the Map File (Version 4)

Table 3-4 shows a sample map file for a version 4 run file.

Addresses

The first three columns in the map show the beginning and ending addresses and the length of each segment. The starting addresses under Start are offsets, not absolute addresses. The offsets are relative to the base memory address at which the operating system loads the run file. This base address is determined at run time.

Segment Names

The fourth column of the sample map file in table 3–4 gives the name of each segment. In the case of a code segment, this name is not the module file name.

In most high-level language programs, you assign this module name at the beginning of the module. The compiler creates the code segment name by appending an underscore and a suffix to this assigned module name, and the Linker reports the resulting name here.

In Assembly language, you can directly name each segment. The Linker does not append a suffix to the segment name.

For easy reference, you can assign the same name for the module file name and for the program module name. This convention is particularly helpful when you use the map to decide what segments to place in overlays, since you enter file names (not internal module names) in the **Object modules** field of the LINK command form. However, you are not required to use this convention.

Segment Classes

The fifth column in the map gives the class of each segment. The Linker groups segments by class and uses class to assign order in the program.

Table 3–4 **Version 4 Map File (Sample)**

Linker (Version)

Start	Stop	Length	Name	Class
0000h	00020h	0021h	EXAMPLE_CODE	CODE
00022h	00022h	0000h	CONST	CONST
00022h	00087h	0066h	DATA	DATA
00090h	0009Bh	000Ch	STACK	STACK
0009Ch	0009Ch	0000h	MEMORY	MEMORY

Program entry point at 0000:0000

Reading the Map File (Version 6)

Table 3–5 shows a map file for a version 6 run file. It is similar in format to the version 4 map file, but includes another column of numbers in parentheses between **Length** and **Name**.

Note: Disregard this column if the map applies to version 6 Real Mode run files.

These numbers are 80286/80386 selectors. For each code segment, this selector is the value of the CS register while it is executing, if you are running in 80286/80386 protected mode. For a data segment, this number is the selector that you use to access data within it.

For all segments within a given group, the selector number is the same. (Refer to section 6 for a discussion of groups.)

Table 3–5 **Version 6 Map File (Sample)**

Linker (Version)

Start	Stop	Length		Name	Class
00000h	00020h	0021h	(0084h)	EXAMPLE_CODE	CODE
00030h	00030h	0000h	(008Ch)	CONST	CONST
00030h	00095h	0066h	(008Ch)	DATA	DATA
000A0h	000ABh	000Ch	(008Ch)	STACK	STACK
000B0h	000B0h	0000h	(008Ch)	MEMORY	MEMORY

Program entry point at 0000:0000 (0084:0000)

Public Symbols and Line Numbers

You can request the Linker to create a map file that lists public symbols and line numbers.

Table 3–6 shows a version 4 map file that lists the values of all public symbols and their addresses. The symbols are sorted first alphabetically and then numerically. A list of line numbers follows the public symbol lists.

You request a list of public symbols by entering **y** in the **[Publics?]** field of the LINK or BIND command form. You request a list of line numbers separately by entering **y** in the **[Line numbers?]** field.

The Address column in table 3–6 contains the notation XXXX:YYYYh; this is the public symbol hexadecimal address.

The Overlay column contains Res if the symbol is in the resident portion of your task, an integer (n) if it is in the nth overlay, and Abs if it is absolute. An absolute symbol is one with a specified place in memory (for example, an address within the operating system).

You use line numbers during debugging, which allow you to examine a known part of your program at a known address, even though there is no public symbol at that address. The addresses, however, are relative to the beginning of the run file.

Table 3–7 shows a list of public symbols, of line numbers, and addresses in a version 6 map file.

In the list of public symbols in the version 6 map, the name of the public symbol is followed by two addresses. The first is the address in real mode; the second is the address in protected mode.

In a version 6 run file, operating system absolute addresses are converted to an 80286/80386–compatible form (called global descriptor table, or GDT), but they are still denoted as absolute in this listing. Application–defined absolute addresses are not permitted in version 6 run files.

Table 3-6 **Sample Version 4 Map File with Lists of Public Symbols and Line Numbers**

Linker (Version)

Start	**Stop**	**Length**	**Name**	**Class**
00000h	00020h	0021h	EXAMPLE_CODE	CODE
00022h	00022h	0000h	CONST	CONST
00022h	00087h	0066h	DATA	DATA
00090h	0009Bh	000Ch	STACK	STACK
0009Ch	0009Ch	0000h	MEMORY	MEMORY

Publics by name	**Address**	**Overlay**
ANOTHERSAMPLEPROCEDURE	0000:000Dh	Res
MAIN	0000:0012h	Res
SAMPLEDATA	0002:0002h	Res
SAMPLETABLE	0002:0004H	Res
SAMPLEPROCEDURE	0000:0008h	Res

Publics by value	**Address**	**Overlay**
SAMPLEPROCEDURE	0000:0008h	Res
ANOTHERSAMPLEPROCEDURE	0000:000Dh	Res
MAIN	0000:0012h	Res
SAMPLEDATA	0002:0002h	Res
SAMPLETABLE	0002:0004h	Res

Line numbers for EXAMPLE_CODE

4 0000:000H	5 0000:000BH	6 0000:000DH	7 0000:000DH
8 0000:0010H			
9 0000:0012H	10 0000:0012H	11 0000:0015H	12 0000:001AH
13 0000:001FH			
14 0000:0000H	15 0000:0008H		

Program entry point at 0000:0000

Table 3–7 Sample Version 6 Map File with Lists of Public Symbols and Line Numbers

Linker (Version)

Start	**Stop**	**Length**		**Name**	**Class**
00000h	00020h	0021h	(0084h)	EXAMPLE_CODE	CODE
00030h	00030h	0000h	(008Ch)	CONST	CONS
00030h	00095h	0066h	(008Ch)	DATA	DATA
000A0h	000ABh	000Ch	(008Ch)	STACK	STACK
000B0h	000B0h	0000h	(008Ch)	MEMORY	MEMORY

Publics by name	**Address**	**Overlay**	
ANOTHERSAMPLEPROCEDURE	0000:000Dh	(0084:000Dh)	Res
MAIN	0000:0012h	(0084:0012h)	Res
SAMPLEDATA	0003:0000h	(008C:0000h)	Res
SAMPLETABLE	0003:0002h	(008C:0002h)	Res
SAMPLEPROCEDURE	0000:0008h	(0084:0008h)	Res

Publics by value	**Address**	**Overlay**	
SAMPLEPROCEDURE	0000:0008h	(0084:0008h)	Res
ANOTHERSAMPLEPROCEDURE	0000:000Dh	(0084:000Dh)	Res
MAIN	0000:0012h	(0084:0012h)	Res
SAMPLEDATA	0003:0000h	(008C:0000h)	Res
SAMPLETABLE	0003:0002h	(008C:0002h)	Res

Line numbers for EXAMPLE_CODE

```
 4 0000:0008H      5 0000:000BH      6 0000:000DH      7 0000:000DH
 8 0000:0010H
 9 0000:0012H     10 0000:0012H     11 0000:0015H     12 0000:001AH
13 0000:001FH
14 0000:0000H     15 0000:000H
```

Program entry point at 0000:0000 (0084:0000)

Allocating Memory Space

Normally, when a task is loaded in a partition, its high end is placed at the high-address end of memory. (Refer to your operating system reference documentation.)

During compilation or assembly, a program can allocate memory needed during execution. This extra memory takes up space in the program's disk file.

Sometimes it is more efficient for a program to allocate a portion of memory only at load time or during execution. Usually, if a program must allocate short-lived memory during execution, it calls AllocMemorySL or ExpandAreaSL, and the memory is allocated toward lower addresses. You address this memory with 32-bit segment-and-offset addresses.

The Linker allows you to choose two unrelated options for allocation of memory space at load or run time. These options are DS allocation and the memory array data code allocation, and you can choose one or both.

DS Allocation

DS allocation allows your program to allocate short-lived memory toward lower addresses as usual, but also allows it to address the memory efficiently with only 16-bit offset addresses. The data segment (addressed by DS) has a maximum size of 64 Kb, and your program takes up a certain amount of that.

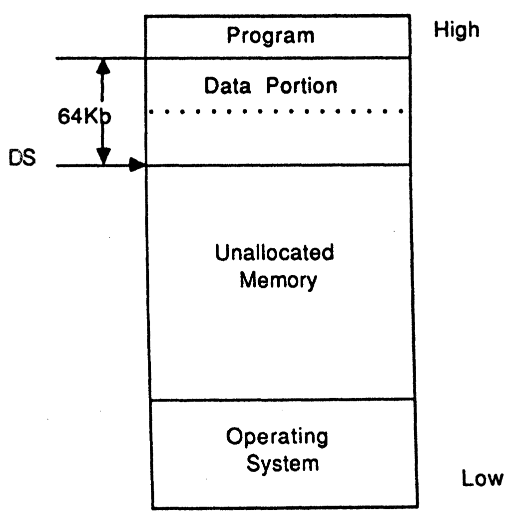
DS allocation allows you to define a maximum-size data segment, even though your program's data segment would normally be smaller. The excess space in this maximum data segment extends beyond your program toward lower memory addresses. You allocate memory in this space with AllocMemorySL or ExpandAreaSL, and you can address within this space with 16-bit offset addresses from DS.

To achieve this, you specify **yes** in the **[DS allocation?]** field of the LINK or BIND command form. The Linker then gives DS the lowest possible value that still allows the data segment to encompass your program's data (or DGroup). (See figure 3-5.)

The program must be arranged with the data segment as its first or lowest-address segment. If your compiler does not order the classes in this way, or if you are writing in Assembly language, you must specify the segment ordering in the first object module listed for linking.

DS allocation has several advantages. It allows the 16-bit DS-relative addressing discussed previously. In addition, memory allocated within this space adjoins the common pool of available memory below the program, and it can be flexibly deallocated and reallocated flexibly by the program. However, the program must make procedure calls for memory allocation, and the 16-bit addressable space is less than 64 Kb.

Figure 3-5 A Real Mode Program with DS Allocation



The Memory Array

The memory array is allocated at the high-address end of your program at load time, not through procedure calls. To use the memory array, you specify values in the **[Max memory array]** and **[Min memory array]** fields of the LINK command form, or in the first parameters of the **[Max array, data, code]** and **[Min array, data, code]** fields of the BIND command form. Figure 3-6 shows the memory array.

You do not have to know the size of your program or how much memory is available in the partition to specify a memory array. The **cParMemArray** field of the Application System Control Block structure contains the number of paragraphs of memory array actually available. If the partition cannot accommodate the minimum memory array you requested, the program is not loaded, and the operating system returns a status code and error message.

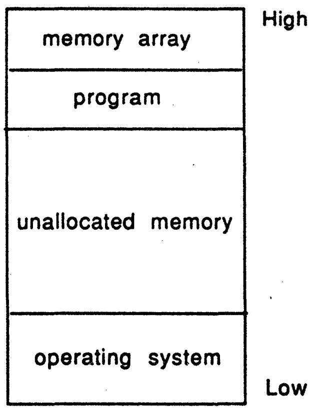
To specify that the task always loads at the lowest possible address, (i.e., with maximum memory array at the end of the task), set the minimum to 0 and the maximum to 1000000.

The memory array has several advantages:

- It is not limited to less than 64 Kb, but can occupy all available memory in a partition.
- The program does not have to make procedure calls to allocate memory during execution.
- The task is at a lower address than the memory array.
- The memory array can be referenced from DS if DGroup is placed at the end of the program.

The memory array is static, however. You cannot reclaim any of it for other uses, and it remains throughout execution. Further, in the form described here, it cannot be referenced from DS. Usually, the ES register is loaded with the lowest address of the memory array.

Figure 3-6 A Program with the Memory Array



Linking a Swapping Program

The Virtual Code Segment Management facility, referred to as the Swapper, allows an application that is larger than the memory in its partition to run, but with a performance trade-off. For this purpose, the program's code is divided into variable-length code segments. One, the resident code segment, is permanently in memory. The remaining segments, or overlays, reside on disk until needed. When you call a procedure in a nonresident overlay, the Overlay Manager of the Swapper brings it into memory.

The term code segment as used here is not the same as a Linker segment. A Swapper code segment, whether resident or in an overlay, can contain several Linker code segments. For example, an overlay can include differently named code segments originating from several different modules.

Only code (not data) is placed in overlays. Module code segments produced by high-level language compilers are pure, so a particular Swapper code segment in memory that is no longer needed can be overlaid by another Swapper code segment. When the first code segment is needed again, it is re-read from the run file. Under this system, only code segments and not data segments are swapped. Nothing is written back to disk, so there is no need for a disk swap file.

You can use the Swapper with programs written in all BTOS high-level languages, and with Assembly programs that follow certain rules. Little or no modification is needed to make an existing program swap. You must write a small amount of initialization code, and you must specify in the command form which modules will contribute code to which overlays.

In some languages, you cannot place certain modules from the run-time library in overlays. In Assembly language, you must follow call/return conventions and certain other rules for your swapping program to work.

Refer to your operating system reference documentation for more information on the Swapper. In addition, refer to the language manuals for language-specific information.

Computing Stack Size

All compilers produce information in object modules from which the Linker can compute the size of the required stack segment. For safety, this information usually specifies a stack that is larger than the actual requirements.

Reducing the Stack

If your program has a data segment that is close to the 64 Kb size limit, in many cases you can reclaim space by reducing the stack size. For example, if you link a program that uses Forms, ISAM, and Graphics, the Linker supplies extra stack space for each of these products. You can examine the size of the default stack by looking at the map file. It is often possible to reduce the amount of stack space by as much as one third without any problem.

To estimate the needed stack size more closely, run the program under the Debugger and set a breakpoint at the end of execution, or at another convenient point just after the stack reaches its largest requirement. Because the stack is initialized to zeros, you can now check to see how much of the low part of the stack is still zeros in order to find the maximum requirement. Allow another 128 bytes (64 bytes for interrupt handlers and 64 bytes for making requests) and reduce the stack size accordingly.

Correcting Stack Overflow

In rare cases, the compiler supplies information that causes the Linker to undercompute the required stack size. An example is a task with many recursive procedures.

The stack grows down from higher to lower addresses. If a program's requirements exceed the stack size, the stack can overwrite whatever precedes it in the link map, causing abnormal program behavior. In this case, you should relink the program, specifying a larger stack size in the command form.

The amount of stack needed is highly program dependent and cannot be estimated precisely. You should increase the stack to the maximum size allowed within the limitations of your data segment. If the program now runs, reduce the stack size according to the guidelines described.

Using the LIBRARIAN Command

You can perform the following operations when you use the LIBRARIAN command at the Executive level:

- build a new library by specifying a new library file name and the object module(s) to compose it
- modify a library by specifying object modules to be added or deleted
- extract one or more object modules from a library by entering the object module name(s)
- produce a sorted cross-reference list of the object modules and public symbols in the library

LIBRARIAN Command Form and Parameters

When you use the Executive LIBRARIAN command, the system displays the LIBRARIAN command form as shown in figure 4–1.

The fields that start and end with brackets (for example, **[Files to add]**) are optional; you can leave the fields blank or enter a parameter. Refer to table 4–1 for information on each bracketed field.

Figure 4–1 LIBRARIAN Command Form

```
Librarian
  Library file
  [Files to add]
  [Modules to delete]
  [Modules to extract]
  [Cross-reference file]
  [Suppress confirmation?]
```

Table 4–1 **Librarian Options**

Field	Action/Explanation
[Files to add]	To add object modules to the library, enter the file names. Separate names with single spaces.
[Modules to delete]	To delete object modules, enter the object module names (do not include the .obj suffix). Separate the names with single spaces. **Note:** If you are replacing a module with a revised version, enter the module in the **[Files to add]** field and allow the Librarian to overwrite it. Deleting the old module and adding a new module takes twice as long.
[Modules to extract]	To extract modules, enter the names in one of the following ways: - to create object module files with the same names used in the library, enter: **ModuleName** The Librarian appends the .obj suffix to the name. - to create object module files with a different name, enter: **FileName (ModuleName)** Separate the names with single spaces. Extraction does not modify the library.
[Cross-reference file]	To produce an alphabetical cross-reference list of public symbols and modules, enter a file name for the list. The Librarian enters a cross-reference map in the file.
[Suppress confirmation]	If you accept the default (no), the system prompts you to confirm the following operations: - creating new library files (when the file name you enter in the **Library file** field is unknown to the system) - replacing an existing object module (when the object module name you enter in the **Files to add** field already exists) - proceeding when a multiply-defined public symbol is encountered To deactivate the prompts, enter **y**.

Building a New Library

To build a new library, use the following procedure:

1 In the Executive command line, type **LIBRARIAN.**

2 Press **RETURN.**

The system displays the LIBRARIAN command form as shown in figure 4–1; the highlight is on the **Library file** field.

3 Enter the new library file name.

Caution: If you enter an existing library name, the Librarian acts on the existing library and can overwrite information. However, the Librarian saves the previous library contents in a file with the suffix -old.

Note: You cannot use **none** for the library name; you use the parameter **none** in the Linker command **[Libraries]** field to direct the Linker not to search libraries.

BTOS standard software uses a **.lib** suffix to assist in file management (it helps identify library files). You can use this suffix, but the system does not require it.

4 In the **[Files to add]** field, enter the object module name(s). If you enter multiple names, separate them with single spaces.

Note: When you add object modules to a library file, the Librarian drops the object module suffix (.obj), if any.

5 Complete the optional fields or accept the default values.

For information on optional fields, refer to table 4–1.

6 Press **GO.**

If you did not turn off the system confirmation prompts, the system prompts you to confirm the creation of a new library file.

Press **GO** to confirm or **FINISH** to exit the Librarian.

Modifying a Library

Caution: When you use the Librarian to add or delete a module, the system deletes the version text string that appears at the end of the library file. (The version should not remain if the library has been changed.) If this deletion causes problems for you, rename the library to preserve the original version number in the name (for example, 8.0CTOS.Lib).

To modify a library, use the following procedure:

1 In the Executive command line, type **LIBRARIAN**.

2 Press **RETURN**.

The system displays the LIBRARIAN command form as shown in figure 4–1; the highlight is on the **Library file** field.

3 Enter the library file name.

4 To add object modules, enter the names in the **[Files to add]** field. Separate the names with single spaces.

5 To delete object modules, enter the names in the **[Modules to delete]** field. Separate the names with single spaces and do not enter the **.obj** suffix.

6 Complete the optional fields or accept the default values.

For information on optional fields, refer to table 4–1.

7 Press **GO**.

The system preserves the content of the previous library file in a file with the library file name plus the suffix **-old**.

If you did not turn off the system confirmation prompts, the system prompts you to confirm the following operations:

- creation of a new library file if the file you entered in the **Library file** field does not exist
- replacement of an object module if an object module file you enter in the **[Files to add]** field has the same name as an object module in the library

  If you press **GO**, the system replaces the library file with the added file.

- a duplicate entry for a public symbol if the public symbol declared in an object module you want to add conflicts with a public symbol in the library

  If you press **GO**, the Librarian adds the object module, but removes the public symbols (both old and new) from the symbol index the Linker searches.

Extracting Object Modules from a Library

To extract object modules from a library, use the following procedure:

1. In the Executive command line, type **LIBRARIAN**.
2. Press **RETURN**.

   The system displays the LIBRARIAN command form as shown in figure 4–1; the highlight is on the **Library file** field.
3. Enter the library file name.
4. In the **[Modules to extract]** field, enter the object module names. Separate the names with single spaces and do not enter the **.obj** suffix.

   To create object module files with names of the form ModuleName.obj, enter **ModuleName**.

   To create object module files with the name FileName, enter **FileName (ModuleName)**.
5. Press **GO**. The Librarian extracts the object module.

Producing A Cross–Reference List Only

If you enter only the library file name and a cross–reference list file name, the Librarian sorts public symbols and object module names alphabetically and enters the list in the file you specify without changing the library file.

The same symbol defined within different modules in a library is called a duplicate symbol name. Such duplicate symbol names are removed from the index of symbols to be searched by the Linker, but are listed in the cross-reference file. The first duplicate symbol name encountered is followed by an asterisk, the second by two, and so on. Modules in which they occur are also listed.

Table 4–2 shows a sample cross-reference list.

Table 4–2 Cross-Reference List (Sample)

```
COMPACTDATETIME..........CMPDT   EXPANDDATETIME............EXPDT
FILLFRAME ..................VAM   POSFRAMECURSOR .............VAM
PUTFRAMEATTRS ..............VAM   PUTFRAMECHARS ..............VAM
QUERYFRAMECHAR..............VAM   RESETFRAM ..................VAM
CMPDT (Length 0177h bytes)

      COMPACTDATETIME

EXPDT (Length 014Ch bytes)

      EXPANDDATETIME

VAM (Length 09B8h bytes)
FILLFRAME              POSFRAMECURSOR         PUTFRAMEATTRS
PUTFRAMECHARS          QUERYFRAMECHAR         RESETFRAM
SCROLLFRAM
```

To produce only a cross-reference list, use the following procedure:

1 In the Executive command line, type **LIBRARIAN**.

2 Press **RETURN**.

The system displays the LIBRARIAN command form as shown in figure 4-1; the highlight is on the **Library file** field.

3 Enter the library file name.

4 In the **[Cross-reference file]** field, enter a file name.

5 Press **GO**. The Librarian produces the cross-reference list in the file you specified.

To display or print the cross-reference list, use an Executive command or the EDITOR.

Invoking the Assembler from the Executive

When you use the Executive ASSEMBLE command, the system displays the ASSEMBLE command form as shown in figure 5–1. Refer to table 5–1 for information on each field.

For information on filling out Executive command forms, refer to your Standard Software documentation.

Figure 5–1 **ASSEMBLE Command Form**

```
Assemble
  Source files
  [Errors only?]
  [GenOnly, NoGen, or Gen]
  [Object file]
  [List file]
  [Error file]
  [List on pass 1?]
  [:f1:]
  [:f0: (default [sys]<edf>)]
```

Table 5–1 **ASSEMBLE Command Fields**

Field	Description
Source files	Enter the list of source files to be assembled. This is the only required field. Separate the names with single spaces, not commas. The result is logically like assembling a single file that is the set of all the source files. See the example following this table.
[Errors only?]	For a listing only of lines with errors, enter **Yes**. The default is No (a full listing). The listing normally contains source and object code for all source lines. Assembly produces an object file and a list file, with names as described below.

Table 5-1 **ASSEMBLE Command Fields** (continued)

Field	Description
[GenOnly, NoGen, or Gen]	This field specifies the mode of listing macro expansion results. GenOnly (the default) lists the results. The NoGen mode listing contains the unexpanded macro invocations. In Gen mode, the listing contains invocations and full expansions, as well as intermediate stages of expansion. This last mode is most useful in debugging complex macros. Note that these controls affect only the listing content. The result of full expansions is always assembled to produce object code. You can also specify this setting in the source with the assembly control directives $GENONLY, $NOGEN, AND $GEN.
[Object file]	This field specifies the object file to which the object code (that results from assembly) is written. The default name is taken from the last source file as follows: The last source name is treated as a string, any final suffix is stripped off beginning with the period, and ".Obj" is added.
[List file]	This field specifies the list file to which the assembly listing is written. The default name is taken from the last source file the same way as for object files, except ".1st" is added.
[Error file]	This field specifies the file to receive the Errors only listing if you want both a full listing and a listing of only the errors. The default is no listing.
[List on pass 1?]	You use this field for diagnosing certain errors in macros. Listings are normally generated only during the second assembly pass. However, some programming errors involving macros prevent the assembly process from ever reaching its second pass. To diagnose such errors, enter **Yes** to get listings for both passes. The default is No.
[:f1:]	You use this field to redirect and use 'Include files' from local or global directories. The system uses this entry (for example, $INCLUDE (:f1: filename)) as a substitution when it assembles the program. The default is [sys]<edf>.

Table 5–1 **ASSEMBLE Command Fields** (continued)

Field	Description
[:f0: (default – [sys]<edf>)]	You use this field to redirect and use 'Include files' from local or global directories. The system uses this entry (for example, $INCLUDE (:f0: filename)) as a substitution when it assembles the program. The default is [sys]<edf>.

Sample Source Files Field Entry

To illustrate the use of the Source files field entry, assume the program is contained in Main.Asm and depends on a set of assembly time parameters. You maintain two source fragments to define the parameters, one for debugging and one for production. Then Source Files would be either:

ParamsDebugging.Asm Main.Asm

or

ParamsProduction.Asm Main.Asm

For default object file examples, assume the last source file is:

[Dev]<Jones>Main

The default object file is then named:

[Dev]<Jones>Main.Obj

If the last source file is:

Prog.Asm

the default object file is:

Prog.Obj

Programs and Segments

Assembler programs are composed of segments, which are variable-length areas of contiguous memory. Each instruction of a program and each item of data is created within a segment, which are then linked together. This section discusses how you name and combine segments with other segments to form a program. It also describes the Assembly language directives used in this process.

Segments and Memory References

At Assembly time, you can define as many segments as you wish, as long as each assembly module has at least one segment. Each instruction of the program and each item of data must lie within a segment. Code and data may be mixed in the same segment, but this is generally not done because such a segment cannot be linked with object segments produced by Pascal or FORTRAN. Programs that contain segments that mix code and data cannot be run in a protected mode.

Referencing Segments

Prior to the introduction of the 80286 and 80386, the registers that contained a segment value from which a physical address could be calculated were called segment registers. These 16-bit registers contained paragraph numbers that were multiplied by 16 and then added to a 16-bit offset to form the 20-bit physical address. This form of addressing is used by the 80286 and 80386 microprocessors when they operate in real mode.

In protected mode, the segment is indirectly pointed to by a selector. A selector is a 16-bit value in which the high 13 bits are an index to a descriptor table, the next 2 bits are the request privilege level, and the low bit indicates whether to use a local or a global descriptor table.

The descriptor table contains the actual segment address (which is no longer necessarily paragraph–aligned) to which the 16–bit offset is added to find the physical address. Since the segment register is a subset of the selector register, both are referred to as the selector register throughout this manual.

(For information about programming in protected mode, refer to your protected mode programming documentation.)

Segment Naming and Linkage

This section discusses segment naming and linkage conventions.

SEGMENT/ENDS Directives

You use the SEGMENT directive to name a segment, and the ENDS directive to indicate the end of a segment.

You name segments explicitly with the SEGMENT directive. (If you do not specify a name, the Assembler assigns the name ??SEG.) The SEGMENT directive also controls the alignment, combination, and contiguity of segments. Its format is:

```
[segname] SEGMENT [align–type] [combine–type]
['classname']
   .
   .
[segname]ENDS
```

You must specify the optional fields (in brackets) in the order given.

Alignment

A segment is located on a memory boundary specified by align–type, as follows:

- PARA (the default): The segment begins on a paragraph boundary, an address whose least significant hexadecimal digit is 0.
- BYTE: The segment can begin anywhere.
- WORD: The segment begins on a word boundary (an even address).
- PAGE: The segment begins on an address divisible by 256.

Combining Segments

The Linker combines segments with other segments as specified by the combine–type field of the SEGMENT directive. Segment combination permits segment elements from different assemblies to be overlaid or connected by the Linker. Such segment elements must have the same segname and classname and an appropriate combine–type, as follows:

- Not combinable (the default)
- PUBLIC: When linked, this segment is placed adjacent to others of the same name. The Linker controls the order of placement during linkage, according to your specifications.
- AT expression: The segment is located at the 16–bit segment base address evaluated from the given expression. The expression argument is interpreted as a paragraph number. For example, if you wish the segment to begin at paragraph 3223h (absolute memory address 32230h), specify AT 3223h.

  You can use any valid expression that evaluates to a constant and has no forward references. You can have an absolute segment in order to establish a template for memory that is accessed at run time. No assembly time data or code is automatically loaded into an absolute segment.

- STACK: The elements are overlaid such that the final bytes of each element are juxtaposed to yield a combined segment whose length is the sum of the lengths of the elements.

  Stack segments with the name STACK are a special case. When stack segments are combined, they are overlaid but their lengths are added together. After the Linker combines all stack segments, it forces the total length of the aggregate stack segment to a multiple of 16 bytes.

  Compilers construct stack segments automatically. However, if your entire program is written in assembly language, you must define an explicit stack segment. There are special rules regarding the use of the stack that you must observe when making calls to standard object module procedures. (See section 11, Accessing Standard Services from Assembly Code.)

- COMMON: The elements are overlaid such that the initial bytes of each element are juxtaposed to yield a combined segment whose length is the largest of the lengths of the elements.

Classname

You can use the optional classname field to change the ordering of segments in the memory image constructed by the Linker. (See your Standard Software documentation.)

Segment Nesting

You can code a portion of one segment, start and end another, and then continue coding the first. However, you can specify lexical nesting only (not physical), since the combination rules given above are always followed.

Lexically-nested segments must end with an ENDS directive before the enclosing SEGMENT directive is closed with its own ENDS directive.

Segment Linkage

The fundamental units of relocation and linkage are segment elements, linker segments, class names, and groups.

An object module is a sequence of segment elements, each of which has a segment name. An object module might consist of segment elements whose names are B, C, and D.

The Linker combines all segment elements with the same segment name from all object modules into a single entity called a linker segment. A linker segment forms a contiguous block of memory in the run time memory image of the task. For example, you might use the Linker to link these two object modules:

- Object Module 1 containing segment elements B, C, D
- Object Module 2 containing segment elements C, D, E

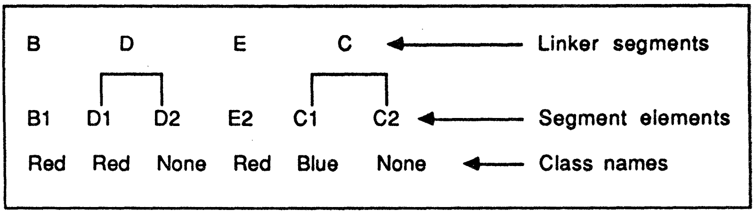
Linkage produces these four linker segments:

- Linker Segment B consisting of element B1
- Linker Segment C consisting of elements C1, C2
- Linker Segment D consisting of elements D1, D2
- Linker Segment E consisting of element E2

(The element number format xi denotes the segment element x in module i.)

Class names determine the ordering of the various linker segments. (A class name is an arbitrary symbol used to designate a class.) All the linker segments with a common class name and segment name go together in memory. For example, if B1, D1, and E2 have class names Red, while C1 has class name Blue, then the ordering of linker segments in memory is: B, D, E, C. Inside the linker segments, the segment elements are arranged as shown in figure 6–1.

Figure 6–1 **Linker Segment Elements**



If two segment elements have different class names, they are considered unrelated for purposes of these algorithms, even though they have the same segment name. Thus, segment names and class names together determine the ordering of segment elements in the final memory image.

The next step for the Linker is to establish how hardware selector registers address these segment elements at run time.

A group is a named collection of linker segments that is addressed at run time with a common selector register. To make the addressing work, all the bytes within a group must be within 64K of each other.

You can combine several linker segments into a group. For example, if you combine B and C into a group, then you can use a single selector register to address segment elements B1, C1, and C2. You can assign segment, class, and group names explicitly in assembler modules using appropriate assembler directives.

ASSUME Directive

The ASSUME directive declares how the instructions and data specified during assembly are to be addressed from the selector registers during execution. You must explicitly control the values in selector registers at run time, since the ASSUME directive does not cause loading of the selector registers referenced.

Use of the ASSUME directive permits the Assembler to verify that data and instructions will be addressable at run time.

The ASSUME directive can be written either as:

ASSUME sel–reg:seg–name [, ...]

or

ASSUME NOTHING

In this example, sel–reg is one of the selector registers.

Sel–name is one of the following:

- A segment name, as in:

  ASSUME CS:codeSeg, DS:dataSeg

- A GROUP name that has been defined earlier, as in:

  ASSUME DS:DGroup, CS:CGroup

- The expression SEG variable–name or SEG label–name, as in:

  ASSUME CS:SEG Main, DS:SEG Table

- The keyword NOTHING as in:

  ASSUME ES:NOTHING

A particular sel–reg:seg–name pair remains in force until another ASSUME assigns a different segment (or NOTHING) to the given sel–reg. To ASSUME NOTHING means to cancel any ASSUME in effect for the indicated registers. A reference to a variable whose segment is assumed automatically generates the proper object instruction. A reference to a variable whose segment is not assumed must have an explicit segment specifications.

For example:

```
Tables SEGMENT
   xTab    DW 100 DUP(10)    ;100–word array, initially
                             ;10's.
   yTab    DW 500 DUP(20)    ;500–word array, initially
                             ;20's.
Tables ENDS
ZSeg SEGMENT                 ;800–word array, initially
                             30's
   zTab    DW 800 DUP(30)
ZSeg ENDS
```

```
Sum SEGMENT

   ASSUME CS:sum,DS:Tables  ,ES:NOTHING
                            ;Sum addressable through
                            ;CS and Tables through DS.
                            No
                            ;assumption about ES (ZSeg
                            ;is not assumed).
   Start: MOV BX, xTab      ;xTab addressable by DS:
                            ;defined in Tables.
      ADD BX, yTab          ;yTab addressable by DS:
                            ;defined in Tables.
      MOV AX, SEG zTab      ;Now AX is the proper
                            ;selector value to
                            ;address reference to
                            ;zTab.
      MOV ES, AX            ;ES now holds the selector
                            ;for ZSeg.
      MOV ES:zTab, 35       ;zTab must be addressed
                            ;with explicit selector
                            ;override—the Assembler
                            ;does not know
                            ;automatically what selector
                            ;register to use.
Sum ENDS
```

In this example, the ASSUME directive:

- tells the Assembler to use CS to address the instructions in the segment Sum. This program fragment does not load CS. CS must have been set previously to point to the segment Sum. For example, CS is often initialized by a long jump or long call.
- tells the Assembler to look at DS for the symbolic reference to xTab and yTab.

Memory Addressing

This section describes the general rules for addressing code in a segment.

Loading Selector Registers

You load the CS register using a long jump (JMP), a long call (CALL), an interrupt (INT n, or external interrupt), or a hardware RESET.

In real mode, the instruction INT n loads the instruction pointer (IP) with the 16–bit value stored at location 4*n of physical memory, and loads CS with the 16–bit value stored at physical memory address (4*n)+2.

In protected mode, the INT n instruction causes the CPU to use n as an index into the IDT.

The following is an example of defining the stack and loading the stack selector register, SS:

```
STACK SEGMENT STACK       ;1000-words of stack.
   DW 1000 DUP(0)
StackStart LABEL WORD     ;Stack expands toward low
                          ;memory.

Stack ENDS
StackSetup SEGMENT        CS:StackSetup
   ASSUME                 BX,  Stack
   MOV                    SS,  BX
   MOV                    SP,  OFFSET  StackStart
   MOV                    ;start = end initially
StackSetup ENDS
```

This example illustrates an important point: You must load each of the two register pairs SS/SP and CS/IP together. The hardware has a special provision to assist in this. Loading a selector register by a POP or MOV instruction causes execution of the very next instruction (only) to be protected against all interrupts. That is why the next instruction, after the load of the stack base register, SS, must load the stack offset register, SP.

CS and its associated offset IP are loaded only by special instructions, never by normal data transfers. SS and its associated offset SP are loaded by normal data transfers but must be loaded in two successive operations.

Selector Override Prefix

If there is no ASSUME directive for a reference to a named variable, you can insert the appropriate selector reference explicitly as a selector override prefix. The format is:

sel–reg:

where sel–reg is CS, DS, ES, or SS, as in:

DS:xyz

This construct does not require an ASSUME directive for the variable reference, but its scope is limited to the instruction in which it occurs.

Thus, the following two program fragments are correct and equivalent:

```
Mycode SEGMENT
ASSUME CS:Mycode,DS:Mydata
  MOV AX, rgwAnything
  ADD AL, rgb
  MOV rgwAnythingElse, AX
Mycode ENDS

Mycode SEGMENT
ASSUME CS:Mycode
  MOV AX, DS:rgwAnything
  ADD AL, DS:rgb
  MOV DS:rgwAnythingElse, AX
Mycode ENDS
```

where Mydata would be defined by:

```
Mydata SEGMENT
  rgwAnything       DW   100 DUP (0)  ;100 words 0's
  rgb               DB   500 DUP (0)  ;500 bytes 0's
  rgwAnythingElse   DW   800 DUP (0)  ;800 words 0's
Mydata ENDS
```

Anonymous References

Memory references that do not include a variable name are called anonymous references. For example:

[BX]
[BP]

Hardware defaults determine the selector registers for anonymous references, unless there is an explicit selector prefix operator. Table 6–1 shows the hardware defaults.

Table 6–1 **Hardware Defaults**

Addressing	Default
[BX]	DS
[BX][DI]	DS
[BX][SI]	DS
[BP]	SS
[BP][DI]	SS
[BP][SI]	SS
[DI]	ES
[SI]	DS

There are a few exceptions to these defaults:

- PUSH, POP, CALL, RET, INT, AND IRET always use SS. This default cannot be overridden.
- String instructions on operands pointed to by DI always use ES. This default cannot be overridden.

It is important that you make an anonymous reference to the correct segment. Unless there is a segment prefix override, the hardware default is applied. For example:

ADD AX, [BP+5] is the same as ADD AX, SS:[BP+5]
MOV [BX+4], CX is the same as MOV DS:[BX +4], CX
SUB [BX+SI], CX is the same as SUB DS:[BX+SI], CX
AND [BP+DI], DX is the same as AND SS:[BP+DI], DX
MOV BX, [SI].one is the same as MOV BX, DS:[SI].one
AND [DI],CX is the same as AND ES:[DI],CX

The following examples require explicit overrides because they differ from the default usage:

ADD AX, DS:[BP+5]
MOV AX, ES:[BX+2]
XOR SS:[BX+SI], CX
AND DS:[BP+DI], CX
MOV BX, ES:[DI].one
AND ES:[SI+4], DX

Memory References in String Instructions

Table 6–2 shows the mnemonics of the string instructions. These include those that you can code with operands (such as MOVS), and those that you can code without operands (like MOVSB, MOVSW).

Each string instruction has type–specific forms (for example, LODSB, LODSW) and a generic form (like LODS). The assembled machine instruction is always type–specific. If you code the generic form, you must provide arguments that serve only to declare the type and addressability of the arguments.

Table 6-2 **String Instruction Mnemonics**

Operation	Mnemonic for Byte Operands	Mnemonic for Word Operands	Mnemonics for Symbolic Operands*
Move	MOVSB	MOVSW	MOVS
Compare	CMPSB	CMPSW	MPS
Load AL/AX	LODSB	LODSW	LODS
Store from AL/AX	STOSB	STOSW	STOS
Compare to AL/AX	SCASB	SCASW	SCAS

* The Assembler checks the addressability of symbolic operands. The opcode generated is determined by the type (BYTE or WORD) of the operands.

A string instruction must be preceded by a load of the offset of the source string into SI, and preceeded by a load of the offset of the destination string into DI.

The string operation mnemonic may be preceded by a "repeat prefix" (REP, REPZ, REPE, REPNE, or REPNZ), as in REPZ SCASB. This specifies that the string operation is to be repeated the number of times contained in CX (repeat, decrementing CX each iteration until CX=0).

String operations without operands (such as MOVSB, MOVSW) use the hardware defaults, which are SI offset from DS, and DI offset from ES. Thus MOVSB is equivalent to:

MOVS ES:BYTE PTR[DI],[SI]

If you do not intend to use the hardware defaults, both segment and type overriding are required for anonymous references, as in:

MOVS ES:BYTE PTR[DI], SS:[SI]

Refer to section 8 for a discussion of PTR.

String instructions cannot use [BX] or [BP] addressing.

Note: You should not use repeat and segment override together if interrupts are enabled, since the hardware defaults are assumed upon return from the interrupt.

GROUP Directive

The GROUP directive specifies that certain segments lie within the same 64 Kb of memory. The format is:

name GROUP segname [, ...]

In this case, name is a unique identifier used in referring to the group. segname can be the name field of a SEGMENT directive, an expression of the form SEG variable–name, or an expression of the form SEG label–name. (For a definition of the SEG operator, see Value–Returning Operators in section 8.) The field [, ...] is an optional list of segnames. Each segname in the list is preceded by a comma.

This directive defines a group consisting of the specified segments. You use the group–name much like a segname (except that a group–name must not appear in another GROUP statement as a segname.)

The GROUP directive has three important uses:

- as an immediate value, loaded first into a general register, and then into a selector register, as in:

  MOV CX,DGroup

  MOV ES,CX

  The Linker computes the value of DGroup as the paragraph address of the lowest (first) segment in DGroup.

- as an ASSUME statement, to indicate that the selector register addresses all segments of the group, as in:

  Assume CS:CGroup

- as an operand prefix, to specify the use of the group base value or offset (instead of the default segment base value or offset), as in:

  MOV CX,OFFSET DGroup:xTab

  (For additional information about OFFSET, refer to Value–Returning Operators in section 8.)

You do not know during assembly whether all segments named in a GROUP directive will fit into 64K; the Linker checks and issues a message if they do not fit. Note that the GROUP directive is declarative only, not imperative. It asserts that segments fit in 64K, but does not alter segment ordering to make this happen. For example:

DGroup GROUP dSeg, sSeg

An associated ASSUME directive that might be used with this group is:

ASSUME CS:codel, DS:DGroup, SS:DGroup

You cannot use forward references to GROUPs.

Procedures

This section discusses how you use Assembly language procedures.

PROC/ENDP Directives

The Assembly language defines a procedure as a block of code and data delimited by PROC and ENDP statements. Although procedures can be executed by in–line "fall–through" of control, or jumped to, the standard and most useful method of invocation is the CALL.

The formats of the PROC and ENDP directives are as follows:

```
name        PROC        NEAR or FAR
             .
             .
            RET
             .
             .
name                    ENDP
```

"name" is specified as NEAR or FAR, and defaults to NEAR.

If you call the procedure by instructions assembled under the same ASSUME CS value, you can specify NEAR. A RET (return) instruction in a NEAR procedure pops a single word of offset from the stack, returning to a location in the same segment.

If you call the procedure by instructions assembled under another ASSUME CS value, then you must specify FAR. A RET in a FAR procedure pops two words, (a new selector base as well as offset), and thus can return to a different segment.

You can nest procedures as shown below, but they must not overlap:

```
WriteFile PROC
    .
    .
    RET
    WriteLine PROC
        .
        .
        RET
        .
        .
    WriteLine ENDP
        .
        .
WriteFile ENDP
```

Calling a Procedure

The CALL instruction assembles into one of two forms, depending on whether the destination procedure is NEAR or FAR.

When you call a NEAR procedure the instruction pointer (IP, the address of the next sequential instruction) is pushed onto the stack. Control then transfers to the first instruction in the procedure.

When you call a FAR procedure, first the content of the CS register is pushed onto the stack, followed by the IP. Control then transfers to the first instruction of the procedure.

You can have multiple entry points to a procedure. All entry points to a procedure should be declared as NEAR or FAR, depending on whether the procedure is NEAR or FAR.

All returns from a procedure are assembled according to the procedure type (NEAR or FAR).

See figure 6–2 for the procedure CALL/RET control flow.

Recursive Procedures and Nesting on the Stack

When procedures call other procedures, the rules are the same for declaration, calling, and returning.

A recursive procedure is one which calls itself, or which calls another procedure which then calls the first. The following additional rules apply to recursive procedures:

- A recursive procedure must be reentrant. This means that it must put local variables on the stack and refer to them with [BP] addressing modes.
- A recursive procedure must remove local variables from the stack before returning, by appropriate manipulation of SP.

The number of calls that can be nested (the nesting limit) depends on the size of the stack segment. Two words on the stack are taken up by FAR calls, and one word by NEAR calls. Of course, parameters passed on the stack and any variables stored on the stack take additional space.

Returning From a Procedure

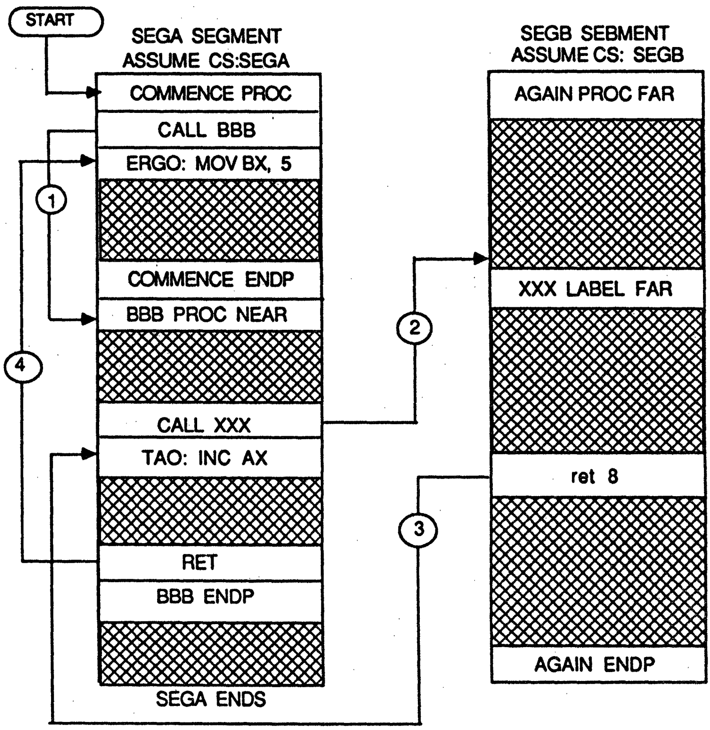
The RET instruction returns from a procedure. It reloads IP from the stack if the procedure is NEAR; it reloads both IP and CS from the stack if the procedure is FAR. IRET is used to return from an interrupt handler and it reloads the flags, as well as CS and IP.

A procedure can contain more than one RET or IRET instruction, and the instruction does not necessarily have to come last in the procedure.

Other Directives

The remainder of this section discusses the use of the location counter ($), and the ORG, EVEN, and program linkage directives (NAME/END, PUBLIC, and EXTRN).

Figure 6–2 Call/Ret Control Flow



KEY:

START	1	2	3	4
Comes from any of: o hardware reset o external interrupt o INT N o CALL BX o NEAR/FAR o JUMP/CALL Whatever the START, CS ← SEGA IP ← OFFSET COMMENCE	SP ← SP-2 (SP) ← IP IP ← OFFSET BBB	SP ← SP-2 (SP) ← CS CS ← SEGB SP ← SP-2 (SP) ← IP IP ← OFFSET XXX	IP ← (SP) SP ← SP+2 CS ← (SP) SP ← SP+2 AND SP ← SP+8 (for RET 8)	IP ← (SP) SP ← SP+2

Location Counter ($) and ORG Directive

The assembly–time counterpart of the instruction pointer is the location counter. The value contained in the location counter is symbolically represented by the dollar sign ($). The value is the offset from the current segment at which the next instruction or data item will be assembled. This value is initialized to 0 for each segment. If a segment is ended by an ENDS directive, and then reopened by a SEGMENT directive, the location counter resumes the value it had at the ENDS.

You use the ORG directive to set the location counter to a non–negative number. The format is:

ORG expression

The expression is evaluated modulo 65536 and must not contain any forward references. The expression can contain $ (the current value of the location counter), as in:

ORG OFFSET $+1000

which moves the location counter forward 1000 bytes.

An ORG directive cannot have a label.

The use of the location counter and ORG is related to the use of the THIS directive. Refer to section 8 for information on the THIS directive.

EVEN Directive

The EVEN directive ensures that an item of code or data is aligned on a word boundary. For example, a disk sector buffer for use by the operating system must be word aligned. For example:

BUFFER	EVEN DW	256	DUP(0)

The Assembler implements the EVEN directive in aligning code by inserting before the code, where necessary, a 1–byte NOP (no operation) instruction (90h).

You can use the EVEN directive only in a segment whose alignment type (as specified in the SEGMENT directive) is WORD, PARA, or PAGE. You cannot use it in a segment having an alignment type of BYTE.

Program Linkage Directives (NAME/END, PUBLIC and EXTRN)

The Linker combines several different assembly modules into a single load module for execution. The assembly module can use three program linkage directives to identify symbolic references between modules. The linkage directives cannot be labeled. They are:

- NAME, which assigns a name to the object module generated by the assembly. For example:

  NAME SortRoutines

  If there is no explicit NAME directive, the module name is derived from the source file name. Thus, the source file [Volname]<Dirname>Sort.Asm has the default module name Sort.

- PUBLIC, which specifies those symbols defined within the assembly module whose attributes are made available to other modules at linkage. For example:

  PUBLIC SortExtended, Merge

  If a symbol is declared PUBLIC in module, the module must contain a definition of the symbol.

- EXTRN, which specifies symbols that are defined as PUBLIC in other modules and referred to in the current module. The format of the EXTRN directive is:

  EXTRN name:type [, name:type...]

  In this format, name is the symbol defined PUBLIC elsewhere and type must be consistent with the declaration of name in its defining module. The type is one of:

- BYTE, WORD, DWORD, structure name, or record name (for variables)
- NEAR or FAR (for labels or procedures)
- ABS (for pure numbers; the implicit SIZE is WORD)

If you know the name of the segment in which an external symbol is declared as PUBLIC, you should place the corresponding EXTRN directive inside a set of SEGMENT/ENDS directives that use this segment name. You may then access the external symbol in the same way as if the uses were in the same module as the definition.

If you do not know the name of the segment in which an external symbol is declared as PUBLIC, you should place the corresponding EXTRN directive at the top of the module outside all SEGMENT/ENDS pairs. To address an external symbol declared in this way, you must:

- Use the SEG operator to load the selector register. For example:

  ```
  MOV AX, SEG Var        ;Load selector value for VAR
  MOV ES, AX             ;into AX and then to ES.
  ```

- Refer to the variable under control of corresponding ASSUME (such as ASSUME ES:SEG var), or use a segment override prefix.

END Directive

The end of the source program is identified by the END directive. This terminates assembly and has the format:

```
END [expression]
```

The expression should be included only in your main program and indicates the starting execution address of the program. For example:

```
END Initialize
```

You must specify the expression as NEAR or FAR.

Data Definitions

The names of data items, segments, procedures, etc., are called identifiers. An identifier is a combination of letters, digits, and three special characters: question mark (?), the at sign (@), and underscore (_). An identifier cannot begin with a digit.

The Assembler accepts three basic kinds of data items: constants, variables, and labels.

- Constants are names associated with pure numbers, i.e., values with no attributes. For example:

  Seven EQU 7 ;Seven represents the constant 7

  Although a value is defined for Seven, no location or intended use is indicated. You can assemble this constant as a byte (eight bits), a word (two bytes), or a doubleword (four bytes).

- Variables are identifiers for data items, forming the operands of MOV, ADD, AND, MUL, etc. You define variables as residing at a certain OFFSET within a specific SEGMENT. They are declared to reserve a fixed memory-cell TYPE, which is a byte, a word, a doubleword, or the number of bytes specified in a structure definition. For example:

  Desk DW 8EH ;Declare Desk a WORD of initial value 008EH

- Labels are identifiers for executable code, forming the operands of CALL, JMP, and the conditional jumps. You define them as residing at a certain OFFSET within a specific SEGMENT. The label can be declared to have a DISTANCE attribute of NEAR if it is referred to only from within the segment in which it is defined. You usually introduce a label as follows:

  label:instruction

  which yields a NEAR label. See PROC (under Procedures in section 6) and LABEL under Labels and the LABEL Directive, which can introduce NEAR or FAR labels.

Constants

There are five types of constants: binary, octal, decimal, hexadecimal, and string. Table 7–1 specifies their syntax.

An instruction can contain 8– or 16–bit immediate values. For example:

```
MOV CH, 53H          ;Word immediate value
MOV CX, 3257H        ;Byte immediate value
```

Constants can be values assigned to symbols with the EQU directive, as shown in these examples:

```
Seven EQU 7          ;7 used wherever Seven
                     ;referenced
MOV AH, Seven        ;Same as MOV AH,7.
```

Refer to section 8 for the complete definition of EQU. The format is:

symbol EQU expression

In this case, expression can be any Assembly language item or expression. For example:

xyz EQU [BP+7]

Table 7–1 **Constants**

Constant Type	Rules for Formation	Examples
Binary (Base 2)	Sequence of 0's and 1's plus letter B	10B B11001011B
Octal (Base 8)	Sequence of digits 0 through 7 plus either letter O or letter Q	7654O 7777Q 77777Q
Decimal (Base 10)	Sequence of digits 0 through 9, plus optional letter D	9903 9903D

Table 7–1 **Constants** (continued)

Constant Type	Rules for Formation	Examples
Hexadecimal (Base 16)	Sequence of digits 0 through 9 and/or letters A through F plus letter h. (If the first digit is a letter, it must be preceded by 0.)	77h 1Fh 0CEACh 0DFh
STRING	Any character string within single quotes. (Strings having more than two characters must use DB.)	'A', 'B' 'ABC' 'Rowrff' 'DN.TWN'

Variable and Label Attributes

Attributes are the distinguishing characteristics of variables and labels that influence the particular machine instructions generated by the Assembler.

Attributes tell where the variable or label is defined. Because of the nature of the processor, it is necessary to know in which SEGMENT a variable or label is defined, and it is necessary to know the OFFSET of the variable or label within that segment.

Attributes also specify how the variable or label is used. The TYPE attribute declares the size, in bytes, of a variable. The DISTANCE attribute declares whether a label can be referred to under a different ASSUMED CS than that of the definition.

Attribute Summary

The following list summarizes of the attributes of data items:

- SEGMENT

  SEGMENT is the segment base address that defines the variable or label. To ensure that variable and labels are addressable at run-time, the Assembler correlates ASSUME CS, DS, ES, and SS (and selector prefix) information with variable and label references. You can apply the SEG operator to a data item to compute the corresponding segment base address (see Value-Returning Operators in section 8).

- OFFSET

  OFFSET is the 16-bit displacement of a variable or label from the base of the containing segment. Depending on the alignment and combine-type of the segment, the run-time value here can be different from the assembly time value (see the SEGMENT Directive in section 6). You can use the OFFSET operator to compute this value.

- TYPE (for Data)

BYTE:	1 byte
WORD:	2 bytes
DWORD:	4 bytes
RECORD:	1 or 2 bytes (according to record definition)
STRUC:	n bytes (according to structure definition)

- DISTANCE (for Code)

  NEAR: Reference only in same segment as definition; definition with LABEL, PROC, or id

  FAR: Reference in segment rather than definition; definition with LABEL or PROC

Variable Definition (DB, DW, DD Directives)

To define variables and initialize memory or both, you use the DB, DW, and DD directives. These directives allocate and initialize memory in units of BYTES (8 bits), words (2 bytes), and DWORDS (doublewords, 4 bytes), respectively. The attributes of the variable defined by DB, DW, or DD are as follows:

- The SEGMENT attribute is the segment containing the definition.
- The OFFSET attribute is the current offset within that segment.
- The TYPE is BYTE (1) for DB, WORD (2) for DW, and DWORD (4) for DD.

The general form for DB, DW, and DD is either:

```
[variable-name] (DB | DW | DD) [exp , ...]
[variable-name] (DB | DW | DD) dup-count DUP (init[, ...])
```

where variable-name is an identifier and either DB, DW, or DD must be chosen.

You can define and initialize arrays of bytes, words, doublewords, structures, and records with, respectively, the DB, DW, DD, structure-name, and record-name directives, as shown in these examples:

```
rgb   DB 50 DUP(66)    ;Allocate 50 bytes named rgb.
                       ;Initialize each to 66.
rgw   DW 100 DUP(0)    ;Allocate 100 words named
                       ;rgw. Initialize each to 0.
rgdd  DD 20 DUP(?)     ;Allocate 20 doublewords named
                       ;rgdd. Do not initialize them.
```

When you refer to array elements, note that the origin of an array is 0. This means that the first byte of the array rgb is rgb(0), not rgb(1). Its nth byte is rgb[n-1].

Also note that indexes are the number of bytes, words, or doublewords.

You can use the DB, DW, and DD directives in the following ways:

- constant initialization
- indeterminate initialization (the reserved symbol "?")
- address initialization (DW and DD only)
- string initialization
- enumerated initialization
- DUP initialization

Constant Initialization

One, two, or four bytes are allocated. The expression is evaluated to a 17–bit constant using twos complement arithmetic. For bytes, the least significant byte of the result is used. For words, the two least significant bytes are used with the least significant byte the lower–addressed byte, and the most significant byte the higher–addressed byte. (As an example, 0AAFFh is stored with the 0FFh byte first and the 0AAh byte second.)

For double words, the same two bytes are used as for words, followed by an additional two bytes of zeros. For example:

```
number           DW 1F3Eh      ;3Eh at number, 1Fh at
                               ;number + 1
                 DB 100        ;Unnamed byte
inches_per_yard  DW 3*12       ;Assembler performs
                               ;arithmetic
```

Indeterminate Initialization

To leave initialization of memory unspecified, use the reserved symbol "?", as shown in the following examples:

```
x                DW   ?          ;Define and allocate
                                 ;a word, contents
                                 ;indeterminate
buffer           DB   1000       ;1000 uninitialized
                 DUP(?)          bytes
```

(See Dup Initialization in this section for information on the DUP clause.)

Address Initialization (DW and DD only)

[variable-name] (DW | DD) init-addr

An address expression is computed with four bytes of precision: two bytes of selector and two bytes of offset. All four bytes are used with DD (with the offset at the lower addresses), but only the offset is used with DW. You can combine address expressions to form more complex expressions as follows:

- A relocatable expression plus or minus an absolute expression is a relocatable expression with the same segment attribute.
- A relocatable expression minus a relocatable expression is an absolute expression, but it is permitted only if both components have the same segment attribute.
- You can combine absolute expressions freely with each other.
- All other combinations are forbidden.

The following are examples of initializing using address expressions:

```
pRequest    DD Request      ;offset and selector
                            ;of Request (32
                            ;bits).
pErc        DD Request+5    ;Offset and selector
                            ;of sixth byte in
                            ;Request.
oRequest    DW Request      ;offset of Request
                            ;(16 bits).
```

String Initialization

You can initialize variables with constant strings as well as with constant numeric expressions. With DD and DW, strings of one or two characters are permitted. The arrangement in memory is tailored to the processor architecture as follows: DW 'XY' allocates two bytes of memory containing, in ascending addresses, 'Y', 'X'. DD 'XY' allocates four bytes of memory containing, in ascending addresses, 'Y', 'X', 0, 0.

With DB, strings of up to 255 characters are permitted. Characters, from left to right, are stored in ascending memory locations. For example, 'ABC' is stored as 41h, 42h, 43h ('A', 'B', 'C').

You must enclose strings in single quotes ('). A single quote or apostrophe is included in a string as two consecutive single quotes, as follows:

```
Date           DB     '08/08/80'
Apostrophe     DB     'I''m so happy!'
Single_Quote   DB     '''NOW IS THE TIME FOR ALL
                        GOOD MEN...'
Run Header     DW     'WG'
```

Enumerated Initialization

[variable-name] (DB | DW | DD) init [, ...]

This directive initializes bytes, words, or doublewords in consecutive memory. You can specify an unlimited number of items as shown below:

```
Squares        DW     0,1,4,9,16,25,36
Digit_Codes    DB     30h,316,32h,33h,34h,
                      36h,37h,38h,39h
Message        DB     'HELLO, FRIEND.',0AH
                      ;14-byte text plus new line code
```

DUP Initialization

To repeat init (or list of init) a specified number of times, use the DUP operator in this format:

dup-count DUP (init)

The duplication count is expressed by dup-count (which must be a positive number). "init" can be a numeric expression, an address (if used with DW, SD, or DD), a question mark, a list of items, or a nested DUP expression.

In the DB, DW, and DD directives, the name of the variable being defined is not followed by a colon. (This differs from many other assembly languages.) For example:

```
Name       DW      100       ;okay
Name:      DW      100       ;wrong
```

Labels and LABEL Directive

Labels identify locations within executable code to be used as operands of jump and call instructions. A NEAR label is declared by any of the following:

```
Start     LABEL              ;NEAR is the default
Start     LABEL NEAR         ;NEAR can be explicit
                             ;Followed by code
Start:
Start     EQU $
Start     EQU THIS NEAR
Start     PROC               ;NEAR is the default
Start     PROC NEAR          ;NEAR can be explicit
```

A FAR label is declared by any of the following:

```
Start2    EQU THIS FAR
Start2    LABEL FAR
Start     PROC FAR
```

LABEL Directive

To create a name for data or instructions, use the LABEL directive, in the format:

name LABEL type

"name" is given segment, offset, and type attributes. The label is given a segment attribute specifying the current segment, an offset attribute specifying the offset within this segment, and a type as explicitly coded (NEAR, FAR, BYTE, WORD, DWORD, structure–name, or record–name).

When the LABEL directive is followed by executable code, type is usually NEAR or FAR. The label is used for jumps or calls, but not MOVs or other instructions that manipulate data. You cannot index NEAR and FAR labels.

When the LABEL directive is followed by data, type is one of the other five classifications. You can index an identifier declared using the LABEL directive if it is assigned a data type such as BYTE, WORD, etc. The name is then valid in MOVs, ADDs, and so on, but not in direct jumps or calls. (Refer to section 8 for information on indirect jumps or calls.)

The main uses of the LABEL directive are:

- accessing variables by an "alternate type"
- defining FAR labels
- accessing code by an "alternate distance" (for example, defining a FAR label with the same segment and offset values as an existing NEAR label)

Label with Variables

The Assembler uses the type of a variable in determining the instruction assembled for manipulating it. You can cause an instruction normally generated for a different type to be assembled by using LABEL to associate an alternative name and type with a location. For example, you can treat the same area of memory sometimes as a byte array, and sometimes as a word array with the following definitions:

```
rgw          LABEL        WORD
rgb          DB           200 DUP(0)
```

You can refer to the data for this array in two ways:

```
ADD AL, rgb[50]          ;Add fifty-first byte to AL
                         ;(rgb[0] is the first byte)
ADD AX, rgw[20]          ;Add tenth word from RGW
                         to AX
                         ;(2 bytes per word)
```

Label with Code

You can use a label definition to define a name as type NEAR or FAR. This is only permitted when a CS assumption is in effect; the CS assumption (not the segment being assembled) is used to determine the SEG and OFFSET for the defined name.

For example:

```
Place        LABEL FAR
SamePlace:   MUL CX,[BP]
```

introduces Place as a FAR label, which is otherwise equivalent to the NEAR label SamePlace.

Label Addressability

The addressability of a label is determined by:

- its declaration as NEAR or FAR
- its use under the same or different ASSUME:CS directive as its declaration

Table 7–2 shows the four coding possibilities for each.

A NEAR jump or call is assembled with a 1–WORD displacement using modulo 64K arithmetic. 64 Kb of the current segment can be addressed as NEAR.

A FAR jump or call is assembled with a 4–byte address. The address consists of a 16–bit offset and 16–bit selector. The entire memory can be addressed as FAR.

Table 7–2 Target Label Addressability

Near Label	Far Label	
Same ASSUME CS:	NEAR Jump/Call	NEAR Jump FAR Call
Different ASSUME CS:	Not allowed	FAR Jump FAR Call

Forward References

The instruction set of the processor often provides several ways of achieving the same end. For example, if a jump is within 128 bytes of its target, the control transfer can be a SHORT jump (two bytes), a NEAR jump (three bytes), or a FAR jump (four bytes). If the Assembler "knows" which case applies, it generates the optimal object code.

However, for the convenience of the programmer, the Assembly language allows, in many cases, the use of a variable or label prior to its definition. When the Assembler encounters such a forward reference, it must reserve space for the reference, although it does not yet know whether the label (for example) will turn out to be SHORT, NEAR, or FAR. If necessary, the Assembler estimates the memory required, and then proceeds on the basis of that estimate.

The Assembler makes two successive passes over the source program, and can always tell during the second pass whether an estimate made during the first pass was correct. If the estimate is too generous, the Assembler corrects the problem during the second pass. For example, it may insert an extra no-op instruction after an offending jump, and still produce valid output. If the estimate is too conservative, however, no such remedy is available. The Assembler then flags the forward reference as an error during the second pass.

You can generally repair this kind of error by a small change to the source text and a reassembly. For example, the insertion of an attribute coercion such as BYTE PTR or FAR PTR is often a sufficient correction. However, the safest course is to follow programming practices that make it unnecessary for the Assembler to guess. This can be done as follows:

- Put EQU directives early in programs.
- Put EXTRN directives early in programs.
- Within a multisegment source file, try to position the data segments (and hence the variable definitions) before the code segments.

Operands and Expressions

The instruction set of the processor makes it possible to refer to operands in a variety of ways, using combinations of base registers, index registers, displacement, and direct offset.

Either memory or a register can serve as the first operand (destination) in most two-operand instructions; the second operand (source) can be memory, a register, or a constant within the instruction. The format of a two-operand instruction is:

MOV Destination, Source

The source operand can be an immediate value (a constant that is part of the instruction itself, such as the "7" in MOV CX, 7), a register, or a memory reference. If the source is an immediate value, then the destination operand can be either a register or a memory reference.

Source and destination operands cannot both be memory-to-memory operations.

You can use a 16-bit offset address to directly address operands in memory. To indirectly address operands in memory, you use base registers (BX or BP) or index registers (SI or DI) or both, plus an optional 8- or 16-bit displacement constant.

A memory reference is direct when a data item is addressed without the use of a register, as in:

```
MOV prod, DX        ;prod is addressed by 16-bit direct
                    ;offset.
MOV CL, jones.bar   ;Offset of jones plus bar is 16-bit
                    ;direct offset.
```

A reference is indirect when a register is specified within brackets, as in:

```
MOV prod[BX], DX  ;Destination address is base
                  ;register plus 16-bit displacement.
MOV CX, [BP][SI]  ;Source address is sum of base
                  ;register and index register.
```

Either memory or a register receives the result of a two-operand operation. You can use any register or memory operand (but not a constant operand) in single-operand operations. You can specify either 8- or 16-bit operands for almost all operations.

Immediate Operands

An immediate value expression can be the source operand of two-operand instructions, except for multiply, divide, and the string operations. The formats are:

[label:] mnemonic memory-reference, expression

[label:] mnemonic register, expression

In this case, [label] is an optional identifier and mnemonic is any two-operand mnemonic (for example, MOV, ADD, and XOR). (See Memory Operands in this section for the definition of memory-reference.) In summary, it has a direct 16-bit offset address, and is indirect through BX or BP, SI or DI, or through BX or BP plus SI or DI, all with an optional 8- or 16-bit displacement.

In the second format, register is any general-purpose (not selector) register. See table 7-1 in section 7 for rules on formation of constants.

The Assembler uses the following steps to develop an instruction containing an immediate operand:

1. determines if the destination is of the type BYTE or WORD
2. evaluates the expression with 17-bit arithmetic
3. If the destination operand can accommodate the result, it encodes the value of the expression using twos complement arithmetic, as an 8- or 16-bit field (depending on the type, BYTE or WORD, of the destination operand) in the instruction being assembled.

In processor instruction formats, as in data words, the least significant byte of a word is at the lower memory address, as shown in the following examples:

```
MOV   CH,hs1p5               ;8-bit immediate value to
                             ;register
ADD   DX,3000H               ;16-bit immediate value to
                             ;register
AND   Table[BX], 0FF00h      ;16-bit immediate value
                             ;(where Table is a WORD
                             ;through BX, 16-bit
                             ;displacement)
XOR   Table[BX+DI+100],7     ;16-bit immediate value
                             ;through BX+DI(Table+100)
```

Register Operands

The following types of registers are used by the Unisys Assembler:

- 16-bit selector (CS, DS, SS, ES)
- 16-bit general (AX, BX, CX, DX, SP, BP, SI, DI)
- 8-bit general (AH, AL, BH, BL, CH, CL, DH, DL)
- 16-bit base and index (BX, BP, SI, DI)
- 1-bit flag (AF, CF, DF, IF, OF, PF, SF, TF, ZF)

Selector registers contain segment base addresses which must be initialized at run time. (This initialization is automatic if you use Assembly language only to implement subroutines for a main program written in a high-level language.)

You can use each of the 16-bit general, 8-bit general, base and index registers in arithmetic and logical operations. Frequently, the AX is called the accumulator, but the processor actually has eight 16-bit accumulators (AX, BX, CX, DX, SP, BP, SI, DI), and has eight 8-bit accumulators (AH, AL, BH, BL, CH, CL, DH, DL). Each of the 8-bit accumulators is either the high-order (H) or the low-order (L) byte of AX, BX, CX, or DX.

One-bit flag registers are accessible to the programmer in the 16-bit FL register. In addition, certain flag registers are set and tested by specific operators.

After each instruction, the flags are updated to reflect conditions detected in the processor or any accumulator. (Refer to appendix D for the flags affected by each instruction.)

The flag-register mnemonics are:

AF: Auxillary Carry
CF: Carry
DF: Direction
IF: Interrupt-enable
OF: Overflow
PF: Parity
SF: Sign
TF: Trap
ZF: Zero

Explicit Register Operands

The two-operand instructions that explicitly specify registers are:

- register to register

  [label:] mnemonic reg, reg

  Example:

  ADD BX, DI ;BX=BX+DI

- immediate to register

  [label:] mnemonic reg, imm

  Example:

  ADD BX, 30H ;BX=BX+30H

- memory to register

  [label:] mnemonic reg, mem

  Example:

  ADD BX, Table[DI] ;BX=BX+DI'th entry in Table

- register to memory

  [label:] mnemonic mem, reg

  Example:

  ADD Table[DI], BX ;Increment DI'th entry in Table by BX

(The "i'th entry" means "entry at i'th byte.)

Implicit Register Operands

Table 8–1 shows the instructions that use registers implicitly.

Table 8–1 **Implicit Register Operands**

Instruction	Implicit Uses
AAA, AAD, AAM, AAS	AL, AH
CBW, CWD	AL, AX or AX:DX
DAA, DAS	AL
MUL, IMUL, DIV, IDIV	AL, AX or AX:DX
LAHF, SAHF	AH
LES	ES
LDS	DS
Shifts, Rotates	CL
String	DS:SI, ES:DI
REP, LOOP	CX
XLAT	AL, BX

The format of instructions with a single register operand is as follows:

[label:] mnemonic reg

Example:

INC DI ;DI=DI+1

Selector Registers

See section 6 for information on selector registers.

General Registers

When a 16–bit general register or base register is one of the operands of a two–operand instruction, the other operand must be immediate, a WORD reference to memory, or a WORD register.

When an 8–bit general register (AH, AL, BH, BL, CH, CL, DH, DL) is one of the operands of a two–operand instruction, the other operand must be an 8–bit immediate quantity, a BYTE reference to memory, or a BYTE register.

Flags

Instructions never indirectly specify the 1–bit flags as operands; flag instructions (such as STC, CLC, CMC) manipulate a specific flag, and other instructions affect one or more flags implicitly (such as INC, DEC, ADD, MUL, and DIV).

Refer to section 9 for flag operations, and appendix D for information on how each instruction affects the flags.

Memory Operands

Memory can be the first or second destination of an operand, but not both.

Memory Operands to JMP and CALL

The JMP and CALL instructions take a simple operand. There are a number of different cases, which are determined by the operand. The control transfer can be direct (with the operand specifying the target address), or indirect (with the operand specifying a word or doubleword containing the target address). The transfer can be NEAR (in which case only IP changes), or FAR (both IP and CS change).

Table 8–2 lists JMP and CALL memory references.

Table 8-2 JMP and CALL Memory References

Operand to JMP/Call	Direct/ Indirect	NEAR/FAR	Target Address
NextIteration	Direct	NEAR[1]	NextIteration
FltMul	Direct	FAR[2]	FltMul
DX	Indirect	NEAR	CS:DX
LabelsNear[DI]	Indirect	NEAR[3]	Contained in word at LabelsNear[DI]
LabelsFar[DI]	Indirect	FAR[4]	Contained in dword at LabelsFar[DI]
DWORD PTR [BX]	Indirect	FAR	Contained in dword at [BX]
WORD PTR [BX]	Indirect	NEAR	Contained in word at [BX]

[1] Assuming NextIteration is a NEAR label in the same segment or group as the next jump or call.

[2] Assuming FltMul is a FAR label--a label to which control can be transferred from outside the segment containing the label.

[3] Assuming LabelsNear is an array of words.

[4] Assuming LabelsFar is an array of dwords.

CALL differs from JMP only in that a return address is pushed onto the stack. The return address is a word for a NEAR call and a dword for a FAR call.

If the Assembler determines that the target of a JMP or CALL is addressable by a 1-byte displacement from the instruction, it uses a special short jump or call instruction. The following examples illustrate the use of JMP and CALL:

```
Again:  SUB     BX,1
        JNZ     Again   ;Short jump will be used
        JMP     Last    ;Not short because Last is a
                        ;forward reference.
Last:   ...
        JMP     $+17    ;Short jump since
                        ;displacement is in
                        ;the range -128 to 127.
                        ;BEWARE: Variable length
                        ;instructions make it easy to
                        ;get this wrong
                        ;it's safer to
                        ;use a label.
        JMP     SHORT   ;Forces assembly of a short
                Last    ;transfer; it will yield an
                        ;error if the target is not
                        ;addressable with a 1-byte
                        displacement.
```

Do not confuse the concepts of PUBLIC and EXTRN with NEAR and FAR. PUBLICs and EXTRNs are used at assembly and link time only; they are not run time concepts. NEAR and FAR, in contrast, control the instructions to be executed at run-time. It is entirely possible for an EXTRN to be NEAR.

Variables

This section covers the use of simple, indexed, and structured variables as operands. If you are unfamiliar with defining and initializing variables, review section 7.

Simple Variables

An unmodified identifier used the same way it is declared is a simple variable, as shown in the following example:

```
wData DW    'AB'
      .
      .
      MOV   BX,   wData
```

Indexed Variables

A simple variable followed by a square-bracketed expression is an indexed variable. The expression in square brackets can be one of the following:

- a constant or constant expression
- a base register (BX or BP)
- an index register (SI or DI)
- a base or index register plus or minus a constant expression (in any order)
- a base register plus an index register plus or minus a constant or constant expression (in any order).

When you use indexed variables, note that the indexing is 0–origin (the first byte is numbered 0), the index is always a number of bytes, and the type is the type of the simple variable to which the index is applied. For example, if the table Primes is defined by:

```
Primes DW    250 DUP (?)
```

and register BX contains the value 12, then the instruction

```
NOV Primes[BX], 17
```

sets the twelfth and thirteenth bytes of Primes (which are the bytes of the seventh word in Primes) to 17.

Double-Indexed Variables

Double-indexed variables use a sum of two displacements to address memory, as shown in the following example:

Primes[BX][SI+5]

You can write most forms of double indexing with a more complex single index expression. For example, these two forms are equivalent:

Var[Disp1][Disp2]
Var[Disp1+Disp2]

The displacement can be constants or expressions that evaluate to constants, base or index registers (BX, BP, SI, or DI), or base or index registers plus or minus a constant offset. The only restriction is that BX and BP cannot both appear, and SI and DI cannot both appear in the same double indexed variable.

These three expressions are *invalid*:

Primes[BX+BP]
Primes[SI][2*DI]
Primes[BX][BP]

Attribute Operators

In addition to indexing, arithmetic, and logical operators, operands can contain a class of operators called attribute operators. You use attribute operators to override an operand's attributes, to compute the values of operand attributes, and to extract record fields.

PTR, The Type Overriding Operator

PTR is an infix operator. It has two operands and is written between the operands in the following format:

type PTR addr-expr

where type is BYTE, WORD, DWORD, NEAR, or FAR, and addr-expr is a variable, label, or number.

PTR sets or overrides the type of its operand without affecting the other attributes of the operand, such as SEGMENT and OFFSET. In the following examples of its use with data, assume rgb and rgw are declared by:

```
rgb  DB  100 DUP(?)
rgw  DW  100 DUP(?)
```

Then, byte-increment and word-increment instructions are generated, respectively, by:

```
INC  rgb[SI]
INC  rgw[SI]
```

Types can be overridden with:

```
INC  WORD PTR rgb[SI]   ;word increment
INC  BYTE PTR rgw[SI]   ;byte increment
```

Sometimes, a variable is not named in an instruction; instead, the instruction uses an "anonymous" variable. In such cases, the PTR operator must always be used, as in:

```
INC  WORD PTR [BX]      ;word increment
INC  BYTE PTR [BX]      ;byte increment
INC  [BX]               ;INVALID because
                        ;the operand [BX]
                        ;is "anonymous."
```

Selector Override Operator

The selector override operator is denoted by the colon (:) and takes three forms:

- sel–reg:addr–expr
- selector–name:addr–expr
- group–name:addr–expr

The SEGMENT attribute of a label, variable, or address–expression is overridden by the selector override operator. The other attributes are unaffected. The first two forms do a direct override; the third recalculates the offset from the GROUP base.

See section 6 for more information on the selector override operator.

Short Operator

The single argument of the SHORT operator is an offset that you can address through the CS selector register. When the target code is within a 1-byte signed (two complement) self-relative displacement, you can use SHORT in conditional jumps, jumps, and calls. This means that the target must lie within a range no more than 128 bytes behind the beginning of the jump or call instruction, and no more than 127 bytes in front of it.

This Operator

The single argument of the THIS operator is a type (BYTE, WORD, DWORD) or distance (NEAR, FAR) attribute. A data item with the specified type or attribute is defined at the current assembly location. The formats are:

```
THIS type
THIS distance
```

The segment and offset attributes of the defined data item are, respectively, the current segment and the current offset. The type or distance attributes are as specified. Thus, the two statements:

```
byteA       LABEL       BYTE
byteA       EQU         THIS BYTE
```

have the same effect. Similarly, $ is equivalent to THIS NEAR.

In the example

```
E1          EQU         THIS FAR
E2:         REPNZ       SCASW
```

the two addresses, E1 and E2, differ in that E1 is FAR and E2 is NEAR.

Value–Returning Operators

The value–returning operators are:

- TYPE accepts one argument, either a variable or a label. For variables, TYPE returns the following:

  1 for type BYTE

  2 for type WORD

  4 for type DWORD, and the number of bytes for a variable declared with a structure type.

  For labels, TYPE returns either –1 or –2 (representing NEAR or FAR, respectively).

- LENGTH accepts one argument, a variable. It returns the number of units allocated for that variable. (The number returned is not necessarily bytes.) For example:

  One DB 250(?) ;LENGTH One=250

  TWO DW 350(?) ;LENGTH TWO=350

- SIZE returns the total number of bytes allocated for a variable. SIZE is the product of LENGTH and TYPE.

- SEG computes the selector value of a variable or a label. Use it in ASSUME directives or to initialize selector registers.

- OFFSET returns the offset of a variable or label. When the final alignment of the segment is frozen at link time the value is resolved. If a segment is combined with pieces of the same segment defined in other assembly modules, or is not aligned on a paragraph boundary, the assembly–time offsets shown in the assembly listing cannot be valid at run–time. The offsets are properly calculated by the Linker if you use the OFFSET operator.

  The only attribute of a variable in many assembly languages is its offset. A reference to the variable name is a reference also to its offset. Three attributes are defined by this assembly language for a variable; therefore, to isolate the offset value, the OFFSET operator is needed. In a DW directive, however, the OFFSET operator is implicit. For example:

  oVar1 DW Var1

is the same as

oVarl DB MOV oVarl, OFFSET Varl

The variables in address expressions that appear in DW and DD directives have an implicit OFFSET.

When used with the GROUP directive, the OFFSET operator does not yield the offset of a variable within the group; instead, it returns the offset of the variable within its segment.

Use the GROUP override operator to get the offset of the variable within the group. For example:

```
DGroup   GROUP     Data,??SEG
data     SEGMENT
           .
           .
xyz      DB        0
           .
           .
         DW        xyz ;Offset within segment
         DW        DGroup:xyz ;Offset within group
data       ENDS
                   ASSUME CS:??SEG,DS:DGroup
         MOV       BX,OFFSET:xyz ;Loads seg offset of
                   xyz
         MOV       CX,OFFSET DGroup:xyz; Loads
                   group
                   offset of xyz
         LEA       CX, xyz ;Also loads group offset
                   of xyz
```

You cannot use forward references to group–names.

Operator Precedence in Expressions

The Assembler evaluates expressions from left to right. It evaluates operators with higher precedence before other operators that come directly before or after. To override the normal order of precedence, use parentheses.

In order of decreasing precedence, the operator classes are:

1. Expressions within parentheses, expressions within angle brackets (records), expressions within square brackets, the structure "dot" operator, ".", and the LENGTH, SIZE, WIDTH, and MASK operators
2. PTR, OFFSET, SEG, TYPE, THIS, and "REGISTER:" (selector override)
3. Multiplication and division: *, /, MOD, SHL, SHR
4. Addition and subtraction: +, –
5. Relational operators: EQ, NE, LT, LE, GT, GE
6. Logical NOT
7. Logical AND
8. Logical OR and XOR
9. SHORT

EQU Directive

You use EQU to assign an assembly–time value to a symbol. The format is:

name EQU expression

The following examples illustrate the use of EQU:

```
y     EQU  z               ;y is made a synonym for z.
xx    EQU  [BX+DI-3]       ;xx is a synonym for an
                           ;indexed reference--note
                           that
                           ;the right side is evaluated
                           ;at use, not at definition.
x     EQU  ES:Bar[BP+2]    ;Selector overrides are also
                           ;allowed.
xy    EQU  (Type y)*5      ;Random expressions are
                           ;allowed.
RAX   EQU                  ;Synonyms for registers are
                           ;allowed.
```

PURGE Directive

You use the PURGE directive to delete the definition of a specified symbol. After a PURGE, the symbol can be redefined. The symbol's new definition is used by all occurrences of the symbol after the redefinition. You cannot purge register names, reserved words, or a symbol appearing in a PUBLIC directive.

Flags

Flags denote or distinguish certain results of data manipulations. In particular, most arithmetic operations set or clear six flag registers. ("Set" means set to 1, and "clear" means clear to 0.) The flags that are affected by data manipulations are AF, CF, OF, PF, SF, and ZF.

Flag Operations

The processor provides the four basic mathematical operations (addition, subtraction, multiplication and division). Both 8–bit and 16–bit operations are available, as are signed and unsigned arithmetic. The addition and subtraction operations serve as both signed and unsigned operations. The two possibilities are distinguished by the flag settings.

You can perform arithmetic directly on unpacked decimal digits, or on packed decimal representations.

Some operations indicate these results only by setting flags. For example, the processor implements the compare instruction as a special subtract which does not change either operand, but it does set flags to indicate a zero, positive, or negative result.

By using one of the conditional jump instructions, a program can test the setting of five of the flags (carry, sign, zero, overflow, and parity). The flow of program execution can be altered based on the outcome of a previous operation.

ASCII and decimal–adjust instructions use one more flag, the Auxiliary Carry flag.

It is important to understand which instructions set which flags. Suppose you wish to load a value into AX, and then test whether the value is 0. The MOV instruction does not set ZF; therefore, the following does not work:

```
MOV     AX,
        wData
JZ      Zero
```

Since ADD sets ZF, the following does work:

```
MOV     AX,     wData
ADD     AX,
        0
JZ      Zero
```

You can set a flag but not test it over the duration of several instructions. However, this is generally a dangerous programming practice. In such cases, the intervening instructions must be carefully checked to ascertain that they do not affect the flag in question. (Refer to appendix D for the flags set by each instruction.)

Auxiliary Carry Flag (AF)

If an operation results in a carry out of, or a borrow into, the low-order four bits of the result, AF is set; otherwise it is cleared. A program cannot test this flag directly; it is used solely by the decimal adjust functions.

Carry Flag (CF)

If an operation results in a carry out of (from addition), or a borrow into (from subtraction), the high-order bit of the result, CF is set; otherwise, it is cleared.

This flag usually indicates whether an addition causes a "carry" into the next higher order digit, or whether a subtraction causes a "borrow." CF is not, however, affected by increment (INC) and decrement (DEC) instructions. CF is set by an addition that causes a carry out of the high-order bit of the destination, and it is cleared by an addition that does not cause a carry. CF is also affected by the logical AND, OR, and XOR instructions.

The contents of an operand are moved one or more positions to the left or right by the rotate and shift instructions. The Carry Flag is treated as if it were an extra bit of the operand by RCL and RCR, which preserve the original value in CF. The value does not, in these cases, remain in CF. The value is replaced with the next bit rotated out of the source.

If an RCL is used, the value of CF is replaced by the high–order bit and goes into the low–order bit. If an RCR is used, the value in CF is replaced by the low–order bit and goes into the high–order bit. (This is useful in multiple–word arithmetic operations.) In other rotates and shifts, the value in CF is lost.

Overflow Flag (OF)

If a signed operation results in an overflow, OF is set; otherwise it is cleared. (That is, an operation results in a carry into the high–order bit of the result, but does not result in a carry out of the high–order bit).

Parity Flag (PF)

If the modulo 2 sum of the low–order eight bits of an operation is 0 (even parity), PF is set; otherwise, it is cleared (odd parity).

Following certain instructions, the number of one bits in the destination is counted and the Parity Flag set if the number is even; it is cleared if the number is odd.

Sign Flag (SF)

If the high–order bit of the result is set, SF is set; otherwise, it is cleared.

Following an operation, the high–order bit of its target can be interpreted as a sign. The SF flag is set equal to this high–order bit by instructions that affect SF. Bit 7 is the high–order bit of a byte and bit 15 is the high–order bit of a word.

Zero Flag (ZF)

If the result of an operation is 0, ZF is set; otherwise, it is cleared.

Following certain operations, if the destination is zero, the Zero Flag is set, and if the destination is not zero, the Zero Flag is cleared. Both ZF and CF are set by a result that has a carry and a zero. For example:

```
 00110101   Carry Flag = 1
+11001011
Zero Flag = 1
 00000000
```

The Macro Assembler

The Assembler supports the definition and invocation of macros, which are expressions that are evaluated during assembly to produce text. The text that results is then processed by the Assembler as source code, just as if it had been literally present in the input to the Assembler. For example, consider the following program fragment:

```
%*DEFINE  (Call2(subr,arg1,arg2))(
        PUSH    %arg1
        PUSH    %arg2
        CALL    %subr
)
        %CALL2  (Input,p1,p2)
```

This fragment defines a macro of three arguments (Call2) and then invokes it. The invocation is expanded to the form:

```
PUSH    p1
PUSH    p2
CALL    Input
```

The character "%" is called the metacharacter, which activates all macro processing facilities: macro invocations are preceded by "%"; macro definitions are preceded by "%*".

The simplest kind of macro definition takes the form:

%*DEFINE (MacroName ParameterList) (Body)

where MacroName is an identifier, ParameterList is a list of parameter names enclosed in parentheses, and Body is the text of the macro.

When parameter names appear in the Body, they are preceded by the "%" character. A simple macro invocation takes the form:

%MacroName (Arglist)

This expands to the corresponding macro Body, with parameter names of the macro definition replaced by arguments from the macro invocation.

Local Declaration

Macros permit the definition of a pattern--the body of the macro--that is to be recreated at each invocation of the macro. Thus, two invocations of a macro normally expand to source text that differs only as the parameters of invocation differ.

However, consider the definition:

```
%*DEFINE        (callNTimes(n,subr))(
                  MOV    AX,%n
                  INC    AX
Again:            SUB    AX,1
                  JZ     Done
                  PUSH   AX
                  CALL   %subr
                  POP    AX
                  JMP    Again)
Done:
```

An invocation such as %CallNTimes(5,FlashScreen) expands to:

```
                MOV    AX,5
                INC    AX
Again:          SUB    AX,1
                JZ     Done
                PUSH   AX
                CALL   Flashscreen
                POP    AX
                JMP    Again
Done:
```

A second invocation of this macro produces an error because it doubly defines the labels Again and Done. The problem in this case is that you want a new, unique pair of labels created for each invocation. You can do this in a macro definition using the LOCAL declaration, which declares a variable within a procedure, as follows:

```
%*DEFINE(CallNTimes(n,subr)) LOCAL Again Done (
                MOV    AX,%n
                INC    AX
%Again:         SUB    AX,1
                JZ     %Done
                PUSH   AX
                CALL   %subr
                POP    AX
                JMP    %Again
%Done:)
```

Conditional Assembly

In a manner carefully integrated with macro processing, the Assembler also supports assembly time expression evaluation and supports string manipulation facilities. These include the EVAL, LEN, EQS, GTS, LTS, NES, GES, LES, and SUBSTR functions.

The examples in table 10–1 illustrate the possibilities of conditional assembly.

Table 10–1 Conditional Assembly Examples

Function	Example	Evaluation of Example	Description
EVAL	%EVAL(3*(8/5))	3h	Evaluate expression
LEN	%LEN(First)	5h	Length of string
EQS	%EQS(AA,AA)	0FFFFh	String equality
GTS	%GTS(y,x)	0FFFFh	String greater
LTS	%LTS(y,x)	0h	String less
NES	%NES(AA,AB)	0FFFFh	String not equal
GES	%GES(y,y)	0FFFFh	String greater or equal
LES	%LES(z,y)	0h	String less or equal
SUBSTR	%SUBSTR (abcde,2,3)	bcd	Substring

Note: EQ, GT, LT, GE, LE, and NE are the numeric equivalents to the string compare operations.

These functions evaluate to hexadecimal numbers, and the relational functions (EQS, etc.) evaluate to 0FFFFh if the relation holds, and to 0h if it does not. The EVAL parameter must evaluate to a number.

You can give the result of a numeric computation performed during macro processing a symbolic name with the SET function, which is invoked in the form:

%SET (name, value)

For example:

%SET (xyz, 7+5)

sets the macro variable xyz to value 0Ch. After the use of SET, %xyz is equivalent to 0Ch.

Similarly, the invocation:

%SET (xyz, %xyz–1)

decrements the value of the macro variable xyz.

Note: If you use the %SET macro in conjunction with the location counter ($, this byte, etc.), the %SET macro should follow a blank line.

The macro facility also supports conditional and repetitive assembly with the control functions IF, REPEAT, and WHILE.

IF has two versions:

%IF (param1) THEN (param2) ELSE (param3) FI

%IF (param1) THEN (param2) FI

The first parameter is treated as a truth value: odd numbers are true and even numbers are false. If the first parameter is true, the IF expression is equivalent to the value of its second parameter.

If the first parameter is false, the IF expression is equivalent to the value of its third parameter (or to the null string if the third parameter is omitted).

For example:

%IF (1) THEN (aa) ELSE (bb) FI

is equivalent to aa, and:

%IF (2) THEN (aa) FI

is equivalent to the null string.

You can use the IF function in conjunction with macro variables to perform a conditional assembly. Suppose a program contains a table that is to be searched for a value at run time. If the table is small, a simple linear search is best. If the table is large, a binary search is preferable, as shown in the following code:

```
%IF (%sTable GT 10)
  THEN(
 ;binary search version here
)ELSE(
 ;linear search here
)
```

You have to define the macro variable %sTable with some numeric value or the expansion of the IF function yields an error.

Sometimes it is convenient to control a conditional assembly based on whether or not a symbol has been defined. Usually, the symbol is not defined and one alternative is selected, but if a definition for the symbol is found, a different alternative is selected.

The macro processor supports this capability with the ISDEF function. ISDEF has two forms: one tests whether a run time symbol (for example, a label) has been defined, and the other tests whether a macro time symbol has been defined. In both cases, the result is 0FFFFH if the symbol is defined, and 0 if the symbol is not defined. The two forms are ,%ISDEF (symbol), which checks a run time symbol, and %*ISDEF (%symbol), which checks a macro time symbol.

Repetitive Assembly

The REPEAT function is used to assemble one of its parameters a specified number of times. The form is:

```
%REPEAT (param1) (param2)
```

For example:

```
%REPEAT (4)
(     DW  0
)
```

is equivalent to:

```
DW  0
DW  0
DW  0
DW  0
```

(Note that in this, and in most examples involving the macro facility, the parentheses are the delimiters of textual parameters, which makes placement critical.)

You use the WHILE function to assemble one of its parameters a variable number of times, depending on the result of an assembly time computation that is performed before each repetition. The form is:

```
%WHILE (param1) (param2)
```

For example, suppose %nWords has the value 3h. Then the result of:

```
%WHILE (%nWords GT 0) (%REPEAT (%nWords)
(      DW    %nWords
)      %SET  (nWords, %nWords-1)
```

is:

```
DW  3h
DW  3h
DW  3h
DW  2h
DW  2h
DW  1h
```

When you use the control functions REPEAT and WHILE, you may want to explicitly terminate expansion. This can be done with the EXIT function. The invocation of EXIT stops the expansion of the enclosing REPEAT, WHILE, or macro. For example, if %n is initially 5, then the expression:

```
%WHILE(%n GT 0)
     (%REPEAT (%n) (%IF (%n) THEN (%EXIT) FI DW %n
)%SET (n, %n-1)
```

expands to:

```
DW 4
DW 4
DW 4
DW 4
DW 2
DW 2
```

Interactive Assembly

The macro capability supports interactive assembly, based on the two functions IN and OUT. You use these functions, respectively, to read input from the keyboard during assembly, and to display information on the video display during assembly. When using IN and OUT, it is important to understand the two-pass nature of the Assembler.

Since the Assembler makes two passes over the text, it expands all macros and macro time functions twice. You must ensure that:

- expressions involving macro-time variables generate the same code or data in both passes
- IN and OUT are not expanded twice

You can can control these effects using the specially defined macro variables PASS1 and PASS2, whose values are shown in table 10-2.

Table 10–2 PASS1 and PASS2 Macro Variable Values

	During First Pass	During Second Pass
PASS1	-1	0
PASS2	0	-1

As an example, suppose you want to prompt the user for a number at the beginning of an assembly, then use this (input) string later. You can do this by inserting the following code near the beginning of the source:

```
%IF (%PASS1 EQ -1)
THEN (%OUT (Enter table size in bytes)
%SET (sTable, %IN)) FI
```

OUT and IN execute during the first pass only, and your input becomes the value of the macro variable sTable; later, you can refer to this by %sTable.

Comments

You can write macro time comments in either of the following formats:

%'text–not–containing–RETURN–or–apostrophe'

or

%'text–not–containing RETURN–or–apostrophe–RETURN

(In this case, RETURN designates the character generated by the RETURN key, code 0Ah.) Since the characters of the embedded text of a comment are ignored, you can use comments to insert extra returns for readability in macro definitions.

MATCH Operation

The special macro function MATCH is particularly useful for parsing strings during macro processing. MATCH permits its parameters to be divided into two parts: a head and a tail. A simple form of this function is:

%MATCH (varl, var2) (text)

For example, following the expansion of

%MATCH (var1, var2) (a, b, c, d)

the macro variable var1 has the value "a" and var2 the value "b, c, d". You can use this facility together with LEN and WHILE. Consider the expression:

```
%WHILE (%LEN(arg) GT 0)(%MATCH (head, arg)(%arg)
     DW %head
))
```

If %arg is initially the text 10, 20, 30, 40, then the expansion is:

```
DW  10
DW  20
DW  30
DW  40
```

Advanced Macro Features

The form of MATCH just described, as well as the form of macro definition and call described earlier, are actually special cases. In fact, the separator between the parameters of MATCH or of a macro can be a (user–specified) separator other than comma.

The remainder of this section explains this and a number of related advanced features of the macro facility.

Macro Identifiers, Delimiters, and Parameters

The entities manipulated during macro processing are macro identifiers, macro delimiters, and macro parameters.

A macro identifier is any string of alphanumeric characters and underscores that begins with an alphabetic character.

A macro delimiter is a text string used as punctuation between macro parameters. There are three kinds of macro delimiters:

- an identifier delimiter is the character "@" followed by an identifier
- an implicit blank delimiter is any text string made up of the "white space" characters space, RETURN, or TAB
- a literal delimiter is any other delimiter. Thus, all the preceding examples have used the comma as a literal delimiter.

A macro parameter is any text string in which parentheses are balanced. The following are valid parameters:

```
xyz
(xyz)
((xyz)()(()))
```

whereas the following are not:

```
(
(()
xy)(
```

The parentheses are considered balanced if the number of left and right parentheses is the same and, in reading from left to right, there is no intermediate point at which more right than left parentheses have been encountered.

The most general form of macro definition is:

%*DEFINE (ident pattern) <locals> (body)

where:

- The "*" is optional
- ident is a macro identifier
- pattern and body are any parenthetically-balanced strings
- <locals> is optional and, if present, consists of the reserved word LOCAL and a list of macro identifiers separated by spaces

In all of the macro definitions illustrated above, the pattern has the form:

(id1, id2, ..., idn)

and all invocations are of the form:

%ident (param1, param2, ..., paramn)

The following example illustrates the use of a user–defined delimiter. The definition:

```
%*DEFINE (DWDW A @AND B)(DW %A
      DW %B)
```

requires an invocation such as:

```
%DWDW 1 and 2
```

which expands to:

```
DW 1
DW 2
```

In this case, the delimiter preceding the formal parameter A and following the formal parameter B is an implicit space. The delimiter between the A and the B is the identifier delimiter @AND.

Bracket and Escape

The macro processor has two special functions, bracket and escape, which you can use to define invocation patterns and parameters.

Bracket

The bracket function prevents further expansion of the bracketed text (macro invocation), and has the form:

```
%(text)
```

where text is parenthetically balanced. The text within the brackets is treated literally. For example, given the definition:

```
%*DEFINE (F(A)) (%(%F(2)))
```

the invocation:

```
%F(1)
```

expands to:

```
%F(2)
```

since the %F(2) is embedded within a bracket function and therefore is not treated as another macro call. If it were not, when invoked it would invoke itself to the limits of the Assembler.

Similarly, the definition:

```
%*DEFINE (DWDW A AND B) (DW %A
        DW %B)
```

declares three formal parameters A, AND, and B (with implicit blank delimiters), whereas the definition:

```
%*DEFINE (DWDW A %(AND) B)(DW %A
        DW %B)
```

treats the AND as a literal delimiter, so that the invocation:

```
%DWDW 1AND2
```

yields the expanded form:

```
DW 1
DW 2
```

Note that the carriage return is required after (DW %A, since macro input is expanded to strings, and DW's must be on separate lines.

Escape

The escape function is useful in bypassing requirements for balanced text or to use special characters like "%" or "*" as regular characters.

The form is:

```
%ntext
```

where n is a digit, 0 to 9, and text is a string exactly n characters long. For example, you might define:

```
%DEFINE (Concat(A,B))(%A%B)
```

and invoke this macro by:

```
%Concat (DW ,%1(3+4%1))
```

which yields the expansion:

```
DW (3+4)
```

The parentheses following the %1 are treated as text by the Assembler.

MATCH Calling Patterns

Generalized calling patterns are applicable to MATCH just as they are to macro definition and invocation. The general form is:

%MATCH(ident1 macrodelimiter ident2)(balancedtext)

MATCH scans text until macrodelimiter is found, then it puts the text up to macrodelimiter in ident1 and the remaining text (less macrodelimiter) in ident2.

For example, if "arg" is initially:

10 xyz 20 xyz 30

then:

```
%WHILE (%LEN(%arg) GT 0)(%MATCH(head @xyz arg)(%arg)
        DW %head
)
```

expands to:

```
DW 10
DW 20
DW 30
```

Processing Macro Invocations

In processing macro invocations, the Assembler expands inner invocations as they are encountered. For example, in the invocation:

%F(%G(1))

the argument to be passed to F is the result of expanding %G(1). You can suppress the expansion of inner invocations using the bracket and escape functions. Thus, in the invocations:

```
%F(%(%G(1)))
%F(%5%G(1))
```

it is the literal text %G(1), not the expansion of that text, that is the actual parameter of F.

Expanded and Unexpanded Modes

All macro processor functions can be evaluated in one of two modes: expanded and unexpanded. When the function, invocation, or definition is preceded by "%", the expanded mode is used. If preceded by "%*", the unexpanded mode is used. In either case, actual parameters are expanded and substituted for formal parameters within the body of invoked macros.

In unexpanded mode, there is no further expansion. In expanded mode, macro processing specified in the body of a macro is also performed. For example, if the macros F and G are defined by:

```
%*DEFINE(F(X))(%G(%X))
%*DEFINE(G(Y))(%Y+%Y)
```

then the invocation:

```
%*F(1)
```

expands to:

```
%G(1)
```

whereas the invocation:

```
%F(1)
```

expands to:

```
1+1
```

Nested Macro Expansion

When macro expansion is nested, inner expansions are performed according to the mode they specify. On completion of inner expansions, processing continues in the mode of the outer expansion. Another way of saying this is that the parameters of user-defined macros are always processed in expanded mode. The bodies are processed in expanded mode when a "%" invocation is used, and in unexpanded mode when a "%*" invocation is used.

The complete list of macro functions is as follows:

DEFINE (p–arg)(b–arg)
EQS (p–arg)
EVAL (p–arg)
GE (p–arg)
GES (p–arg)
GT (p–arg)
GTS (p–arg)
IF (p–arg) THEN (b–arg) ELSE (b–arg)
ISDEF (b–arg)
LEN (b–arg)
LE (p–arg)
LES (p–arg)
LT (p–arg)
LTS (p–arg)
MATCH (p–arg)(b–arg)
METACHAR (p–arg)
NE (p–arg)
NES (p–arg)
OUT (b–arg)
REPEAT (p–arg)(b–arg)
SUBSTR (b–arg)(p–arg, p–arg)
WHILE (p–arg)(b–arg)

where p–arg denotes parameter–like arguments and b–arg denotes body–like arguments.

Assembly control directives (explained in appendix F), begin with a "$" after a RETURN. If a control is encountered in expanded mode, it is obeyed; otherwise, the control is simply treated as text.

Changing the Metacharacter

You can substitute a different character for the built–in metacharacter "%" by calling the function METACHAR, in the form:

%METACHAR (newmetacharacter)

The metacharacter should not be a left or right parenthesis, an asterisk, an alphanumeric character, or a "white space" character.

Accessing Standard Services from Assembly Code

You can access all system services from modules written in Assembly language. To do so, you must follow certain standard calling conventions, register conventions, and segment/group conventions. If you also wish to use the system's virtual code management services, you must follow additional virtual code conventions.

Calling Conventions

This discussion explains how to invoke operating system services and standard object module procedures from programs written in Assembly language. The following example of a call to the standard object module procedure ReadBsRecord is helpful in understanding this subject.

The calling pattern of this procedure is:

ReadBsRecord (pBSWA, pBufferRet, sBufferMax
psDataRet): ErcType

For a detailed description of this procedure, refer to your system procedural interface documentation.

The operating system and the standard object modules deal with quantities of many different sizes, ranging from single–byte quantities, such as Boolean flags, to multibyte quantities, such as request block and Byte Stream Work Areas. Three of these sizes are special: one byte, two bytes, and four bytes. Only quantities of these sizes are passed as parameters on the stack or returned as results in the registers.

Pointers

When it is necessary to pass a larger quantity as a parameter or to return a larger quantity as result, a pointer to the larger quantity is used in place of the quantity itself. A pointer is always a 4–byte logical memory address consisting of an offset and selector base address.

For example, ReadBsRecord takes as parameters a pointer to a Byte Stream Work Area (pBSWA), a pointer to a buffer (pBufferRet), a maximum buffer size (sBufferMax), and a pointer to a word containing the size of some data (psDataRet). ReadBsRecord returns an error status of type ErcType. The pointers are all 4-byte quantities, the size is a 2-byte quantity, and the error status is a 2-byte quantity.

Suppose that data is allocated by the declarations:

```
sBSWA      EQU    130
sBuffer    EQU    80

bswa       DB     sBSWA      DUP(?)
buffer     DB     sBuffer    DUP(?)
sData      DW     ?
```

To call ReadBsRecord, you must first push the following onto the stack, in order: a pointer to bswa, a pointer to buffer, the size of buffer (the constant sBuffer), and a pointer to sData. If DS contains the selector for the segment containing bswa, buffer, and sData, you accomplish this with the following code:

```
PUSH DS                 ;Push the selector
                        ;for bswa
LEA  AX, bswa           ;Set Ax to th  offset of bswa
PUSH AX                 ;Push the offset of bswa
PUSH DS                 ;Ditto for the buffer
LEA  AX, BUFFER
PUSH AX
PUSH sBuffer            ;Push sBuffer onto the stack
PUSH DS                 ;Push the selector
LEA  AX, sData
PUSH AX                 ;and then the offset of sData
CALL ReadBsRecord       ;Do the call
```

Pointers are arranged in memory with the low-order part (the offset), at the lower memory address, and the high-order part (the selector), at the higher memory address. However, the processor architecture is such that stacks expand from high memory addresses toward low memory addresses. Therefore, the high-order part of a pointer is pushed before the low-order part.

This sample code actually computes the various pointers at run time. It is also possible to precompute the pointers by adding the following declaration to the program:

```
pBSWA     DD    bswa
pBuffer   DD    buffer
psData    DD    sData
```

If this is done, the appropriate calling sequence is:

```
LES       BX, pBSWA
PUSH      ES
PUSH      BX
LES       BX, pBuffer
PUSH      ES
PUSH      BX
PUSH      sBuffer
LES       BX, psData
PUSH      ES
PUSH      BX
CALL      ReadBsRecord
```

The LES instruction loads the offset part of the pointer into BX and the selector part into ES in a single instruction.

Object module and system common procedures as well as procedural references to system services must be declared EXTRN and FAR. These declarations may not be embedded in a SEGMENT/ENDS declaration. (In appendix G, see line 6 of figure G–3.)

The result returned by ReadBsRecord is a 2–byte quantity, which, according to Unisys calling conventions, is returned in AX. If the result were a 4–byte pointer, the selector part would be returned in ES and the offset part in BX. If the result were a 4–byte datum (not a pointer), the high word would be in DX and the low word would be in AX.

Other Conventions

All of the 4–byte quantities described in this example are pointers. There are many cases in which the operating system and standard object module procedures deal with 4–byte quantities other than pointers, such as logical file addresses (lfa).

It is important to understand that you should not use selector registers as data registers. Loading a selector register with an invalid selector in protected mode causes a protection fault. For more information about programming in the protected mode environment, refer to your protected mode programming documentation.

There is an additional case that is not illustrated by the example of ReadBsRecord. When a parameter is a single byte, such as a boolean flag, two bytes are pushed onto the stack, although the high–order byte of these two bytes is not used. Therefore, the instruction

```
PUSH BYTE PTR[BX]
```

adds two bytes to the stack. One of these bytes is specified by the operand of the PUSH instruction; the other is not set and no reference should be made to it. Similarly, when the result of a function is a single byte, that byte is returned in AL and no reference should be made to the contents of AH.

Register Usage Conventions

When writing an Assembly language call to a standard object module procedure or to the operating system, you must be aware of the Unisys standard register conventions. The contents of the CS, DS, SS, SP and BP registers are preserved across calls; they are the same on the return as they were just prior to the pushing of the first argument.

It is assumed that SS and SP point, respectively, to the base of the stack and to the top of the stack. It is also assumed that this stack will, in general, be used by the called service. (You should not put temporary variables in the stack area below SS:SP. Refer to Interrupts and the Stack in this section for details.)

These conventions place no particular requirement on the contents of BP unless you are using virtual code segment management. (Refer to Virtual Code Segment Management and Assembly Code in this section for details of BP usage with virtual code.)

However, the Debugger cannot trace the stack of a procedure being debugged if BP is not used as shown in the your system interface reference documentation. The other registers and the flags are not automatically preserved across calls to Unisys procedures, so any registers that the caller needs to preserve must be explicitly saved by the caller in a particular application.

Although there is no absolute requirement that these register usage conventions be followed in parts of an application that do not call standard Unisys services, failing to do so is not recommended in the Unisys programming environment.

Segment and Group Conventions

This section discusses segment and group conventions.

Main Program

A main program module written in Assembly language must declare its stack segment and starting address in a special way. This method is illustrated in the sample assembler module in figure G–2. In particular:

- The stack segment must have the combine type Stack. (See line 24.)
- The starting address must be specified in the END statement. (See line 29.)

When the program is run, the operating system performs the following steps:

1. It loads the program.
2. It initializes SS to the segment base address of the program's stack.
3. It initializes SP to the top of the stack.
4. It transfers control to the starting address with interrupts enabled.

Use of SS and DS When Calling Object Module Procedures

If a program calls Unisys object module procedures, there are additional requirements. Refer to the program in figure G–3, which illustrates the following points:

- The stack segment must have segment name Stack, combine type Stack, and classname 'Stack'. (See line 43.)
- Although not required, it is standard practice that user code be contiguous in memory with Unisys code and that code be at the front of the memory image. You can achieve this if all code segments have classname 'Code' and this class is mentioned before any other in the module. (See lines 9 through 13).

- You should avoid forward references to constants. It is also standard, though not required, to make user constants contiguous with Unisys constants in the memory image, and to locate constants directly after code. You can achieve both goals by giving all constant segments the classname 'Const' and by mentioning this classname before any other except 'Code'. (See lines 15–23.)
- You should avoid forward references to data. It is also standard, though not required, to make user data contiguous with Unisys data in the memory image, and to locate data directly after constants. You can achieve both goals by giving all data segments the classname 'Data' and by mentioning this classname before any others except 'Code' and 'Const'. (See lines 25–36.) The EXTRN declarations for data declared in object module procedures must be embedded in the data SEGMENT/ENDS declarations.
- Any time that a call is made to an object module procedure, DS and SS must contain the segment base address of a special group named DGroup. This group contains the Data, Const, and Stack segments, and is declared as shown in line 51.

  In addition, at the time of a call to an object module procedure, SP must address the top of a stack area to be used by the called procedure. A correct initialization of SS, SP, and DS is illustrated in lines 60–67.

  These values need not be maintained constantly, but if they change, you should restore them (using the appropriate top of stack value in SP if it has changed) for any call to an object module procedure. Note that the operating system's interrupt handlers save the user registers by pushing them onto the stack defined by SS:SP. Therefore, a valid stack must be defined whenever interrupts are enabled.

Interrupts and the Stack

If interrupts are enabled, interrupt routines use the stack as defined by SS and SP. Therefore, you should never, even temporarily, put data in the stack segment at a memory address less than SS:SP.

Use of Macros

As discussed above, the instructions to set up parameters on the stack before a call and to examine the result on return are complex. The instructions that must be executed differ slightly according to whether a parameter is in a register, a static variable, an immediate constant, a word, or a doubleword.

If you are programming a less complex assembly module, it may be preferable to program the required calling sequences just once, include them in your program as macro definitions, and invoke them using the Assembler's macro expansion capability.

For example, the procedural interface to the Write operation is given in your system procedural interface documentation as:

Write (fh, pBuffer, sBuffer, lfa, psDataRet): ErcType

where fh and sBuffer are 2-byte quantities and pBuffer, lfa, and psDataRet are 4-byte quantities. The corresponding external declaration and macro definition would be:

```
EXTRN      Write:  FAR
%*DEFINE  (Write(fh pBuffer sBuffer lfa psDataRet))
                   (PUSH %fh
                   PUSH  WORD PTR %pBuffer[2]
                   PUSH  WORD PTR %pBuffer[0]
                   PUSH  %sBuffer
                   PUSH  WORD PTR %lfa[2]
                   PUSH  WORD PTR %lfa[0]
                   PUSH  WORD PTR %psDataRet[2]
                   PUSH  WORD PTR %psDataRet[0]
                   CALL  Write)
```

Note that the 4-byte quantities are treated slightly differently from the 2-byte quantities, requiring first a PUSH of the high-order word, then a PUSH of the low-order word.

The following example illustrates the use of this macro with "static" actual parameters:

```
fh          DW    ?
            EVEN
buffer      DB    512 DUP(?)
sBuf        DW    SIZE buffer
pBuf        DD    buffer
lfa         DD    ?
sDataRet    DW    ?
psDataRet   DD    sDataRet
            .
            .

;code to initialize fh, buffer, and lfa
      .
      .
   %Write(fhpBuffer sBuffer lfa psDataRet)
```

You might, instead, want to invoke this macro with actual parameters on the stack. Suppose that the quantities rbfh, rbsBuf, rbpBuf, rblfa, and rbpsData are on the stack and that the top of the stack pointer is in register BP. A sample invocation is as follows:

```
rbfh        EQU  -6
rbsBuf      EQU  -8
rbpBuf      EQU  -10
rblfa       EQU  -14
rbpsDat     EQU  -18
            %Write([BP+rbfh] [BP+rbpBuf]
                   [BP+rbsBuf] [BP+rblfa]
                   [BP+rbpsData]
```

Virtual Code Segment Management and Assembly Code

The virtual code segment management services of the Unisys Information Processing System allow you to configure a program (written in Assembly language, in any of the Unisys compiled languages, or in a mixture of these) into overlays. Although data cannot be overlaid with these services, code can be overlaid.

Moreover, the run time operations whereby code overlays are read into memory and discarded from memory are entirely automatic. When linking the program, you only have to specify which modules are to be overlaid. You do not have to make any changes to the program other than inserting a single procedural call at the beginning that initializes virtual code segment management services. (Refer to your operating system reference documentatin for details.)

Operational Rules for the Assembly Programmer

The correct automatic operation of the virtual code facility makes certain assumptions about stack formats and register usage in the run time environment. These assumptions are automatically satisfied by the compiled languages of the Unisys System. However, you must follow some simple rules if you use virtual code segment management.

If a program contains no calls to overlaid modules from Assembly language code, then the presence of Assembly language code in the program has no effect on the operation of virtual code segment management services. In this case, there are no additional rules.

An overlay fault is defined as a call to, or return to, an overlaid module that is not in memory. An overlay fault automatically invokes virtual code segment management services to read the required overlay into memory and possibly to discard one or more other overlays from memory. The virtual code segment management services do this, in part, by examining the run time stack.

Therefore, if there are control paths in a program such that the stack may contain entries created by Assembly language code when an overlay fault occurs, you must observe the following additional rules:

- You must follow the register usage conventions discussed earlier. The intervention of the virtual code segment management service preserves the registers SS, SP, DS, and BP, and, if an overlay fault occurs during the return from a function, it preserves registers AX, BX, DX, and ES where results may be returned.

  Other registers are not, in general, preserved and therefore cannot be used to contain parameters or return results. All Assembly language modules which are linked into a run file that uses overlays must begin with a PUSH BP and end with a RET.
- The stack segment must be named STACK and must be part of DGroup. This happens automatically if a program is a mixture of Assembly language code and compiled code, and if all code shares the same stack. If a main program is written in Assembly language, it must be done explicitly.
- You must declare all directives using the PROC and ENDP directives. Procedure bodies may not be defined within other procedure bodies. For instance, the following pattern is not permitted:

  ```
  Outer  PROC   FAR  ;Code of Outer
  Inner  PROC   FAR  ;Code of Inner
  Inner  ENDP        ;More code of Outer
  Outer  ENDP
  ```

  The following pattern is correct:

  ```
  Outer  PROC   FAR  ;Code of Outer, More code of Outer
  Outer  ENDP
  Inner  PROC   FAR  ;Code of Inner
  Inner  ENDP
  ```

  This is only a restriction on syntactic nesting. There is no restriction on nested calls, and Outer can, in any case, contain calls to Inner.

- When control enters an Assembly language procedure, the most recent entry on the stack is the return address, if all of the conventions above are followed. In addition to preserving the value of BP, the procedure must push this value onto the stack before it makes any nested calls. No values may be pushed onto the stack between the return address and the pushed BP.

  This convention enables the virtual code segment management services to scan the stack during an overlay fault. Its violation is not detected as an error, but causes the overlaid program to fail in unpredictable ways. Naturally, the pushed BP must be popped during the procedure's exit sequence.
- You must place all code in a class named CODE.
- Do not use the SEG operator on an operand in class CODE, nor in any segment that is part of an overlay. In particular, the following instruction is not permitted:

```
MOV        AX, SEG Procedure
```

- If you want to construct a procedural value (a value that points to a procedure) it must be done in a class other than CODE by either of these two methods:

```
pProc   DD   Procedure
pProc   DW   Procedure
        DW   SEG Procedure
```

  Such procedural values do not point directly at the procedure (since the procedure may be in an overlay), but at a special resident transfer vector created by the Linker. Such a procedural value may be invoked by the code:

```
CALL    DWORD PTR pProc
```

- If a procedure is resident and you wish to address the procedure code directly (and not its entry in the resident transfer vector), use the operators RSEG and ROFFSET in place of SEG and OFFSET. If you apply RSEG or ROFFSET to a value in an overlay, an error is detected during linking.

System Programming Notes

The rest of this section describes some of the algorithms and data structures that make up the virtual code segment management facility. An understanding of these details is not needed by the user of the virtual code segment management facility. They are included for the system programmer who is interested in a model of the internal workings of the virtual code segment management facility.

Statics Segment and Stubs

If you specify the use of overlays when you invoke the Linker, it creates in the run file a special segment in the resident part of the program called the statics segment. This segment contains a transfer vector which is an array of 5–byte entries called stubs, with one stub for each public procedure in the program.

A stub consists of one byte containing an operation code, either JMP or CALL, and four bytes containing a long address. The Linker notes each call to a public procedure in an overlaid program and transforms it to an intersegment indirect call through the address part of the corresponding stub.

The contents of the address part of a stub for a procedure which is in memory (either resident or overlaid but currently swapped in) is the actual starting address of the procedure. Consequently, the call to such a procedure is slower than it would be in a non–overlaid program by only one memory reference.

The contents of the address part of a stub for a procedure not in memory is the address of a procedure in the virtual code segment management facility. Thus, a call to such a procedure actually transfers to the virtual code segment management facility. This kind of call to the virtual code segment management facility is a "call fault." When a call fault occurs, the virtual code segment management facility reads the needed overlay into the swap buffer.

Before control transfers to the called procedure, two other steps are taken:

1 The address in all stubs for procedures in the overlay is changed to the swapped-in address of the procedure.

2 If some overlays had to be deleted from the swap buffer to make room for the new overlay, the stubs for their procedures are reset to the address of the procedure in the virtual code segment management facility that deals with call faults.

  It is possible for an overlay to be deleted from memory even though control is nested within it—that is, even though a return into it is pushed onto the stack. This situation is handled properly; all such stacked return addresses are changed to the address of a procedure in the virtual code segment management facility that subsequently swaps the overlay back into memory when a "return fault" occurs.

In the preceding discussion, no reference is made to the first byte of a stub, the operation code. This byte is used only for calls of procedural values. For an overlay in memory, the virtual code segment management facility arranges that the operation code is a jump instruction. Thus, an invocation of a procedural argument for such a procedure results in a call to a jump instruction which then transfers control to the procedure.

For an overlay not in memory, the virtual code segment management facility arranges that the operation code is a call. Since the address part of such a stub is the address of the virtual code segment management facility, the invocation of such a procedure activates the virtual code segment management facility.

Linker and Librarian Messages

Linker and Librarian messages are similar because the structure and functions of the two programs are related. Throughout this appendix, references to Linker messages and solutions are also applicable to the Librarian unless an exception is noted.

If an error occurs during linking, the following message appears:

There were x errors detected.

The map file includes descriptions of the errors.

Levels of Linker Errors

The Linker can encounter three levels of problems:

- violation of a Linker convention that still allows the Linker to produce a valid run file (program results can be affected)
- violation of a Linker convention that produces a run file that you cannot run (the system crashes if you try to run the file)
- fatal errors that cause the Linker to abort the linking process (the Linker does not produce a run file)

The Linker cannot always provide a complete diagnosis for each problem because it may not have enough information. For some of the complex problems, you must examine your program, using clues from the Linker messages.

Linker Compatibility

The Linker is compatible with only certain versions of CTOS.lib, Compilers and the Assembler. If you use an incompatible Compiler, Assembler, or CTOS.lib, errors can occur.

Causes of Linker Errors

Linker messages result from:

- LINK or BIND command input problems, such as erroneous file names or a missing entry from a required field

  These problems prevent the Linker from producing a run file.

- capacity limitations, such as too many public symbols or not enough memory

  These limitations prevent the Linker from producing a run file.

  Note:

  If the problem is a lack of memory, try running the program in a larger partition or on a workstation with more memory.

- relocation or overlay problems

  If you have a relocation error, you should try rearranging the input modules listed in the LINK or BIND command form.

  If the error persists, you must determine the program's segment size requirement and reduce it. You can use the Linker list file (filename.map) to determine segment lengths. You can allocate large buffers to decrease the data segment memory requirements.

- I/O problems, such as an inability to create, read, write, or perform other operations on disk files

  These problems prevent the Linker from producing a run file.

  A BTOS error code accompanies most I/O problems; refer to table A–2, or to your status codes documentation.

- Compiler/Assembler problems, such as using the latest version of the Linker on object modules produced by earlier versions of a Compiler or the Assembler

Linker/Librarian Error Messages

This appendix contains two tables of Linker/Librarian messages:

- Table A–1 is an alphabetical list of messages that do not have status code identification. The table provides an explanation/action for each message.
- Table A–2 is a numerical list of messages that have status code identification. Some of these messages also appear in your status codes documentation.

  Numeric status codes for the Linker are within the range 4400 through 4423. The status codes from this range that do not appear on the list in table A–2 are part of internal Linker error checking; if you see an unlisted status code displayed, you should report it to Unisys because it results from a Linker or compiler error.

Table A–1 **Linker Messages**

Message	Explanation/Action
Bad max parameter	You entered a minimum higher than a maximum for the array size in the LINK or BIND command form fields.
Bad numeric parameter	You entered a nondecimal character in a LINK or BIND command form field that requires a decimal number.
Bad yes/no parameter	You entered something other than yes or no in a LINK or BIND command form field that requires a yes/no response.
IDIV instruction in overlay	When a Pascal or FORTRAN program contains code that results in an IDIV (integer division) instruction within an overlay, this error results. It indicates a real problem only if you plan to run the resulting run file on one of the affected systems (one which uses an early production 80186 processor chip). Move the code containing IDIV into the resident or ensure that all integer–division operands are positive. The alternative is to avoid using the DIV operator in Pascal, or an I/J construction in FORTRAN (where I and J are integers), unless you are sure that all operands are positive.

Table A–1 **Linker Messages** (continued)

Message	Explanation/Action
Illegal segment address reference type 1	The Linker has not created a stub in the data structure for a procedure you called in an overlay (normally this is an Assembly program problem). If you are trying to link an Assembly program: - If the message **Warning: proc near xxxxx in xxxxx** doesn't follow CALL/RET conventions appeared during the link, examine that location in your Assembly program. - If the message did not appear, examine your entire Assembly program for call/return violations. The location cited with the message indicates where the call occurred. You can use this location to refer to a compilation listing to see what was called. **Note:** Some run time library modules in noncurrent versions of high level language Compilers generate code that violates the Linker call/return conventions. Either place such modules and the calls to them in the resident portion of your code or upgrade your Compiler to the current level.
Illegal segment address reference type 2	Parts of a procedure address have been separated. In a swapping program, it is illegal to use only one part of a two–part procedure address. In PL/M you can generate this error by using the construction p–@ProcedureName, which generates the statement MOV AX, SEG ProcedureName. To find the overlay address of a PL/M procedure name, you must define the procedure as a static constant in a DECLARE statement.
Illegal segment address reference type 3	Parts of a procedure address have been separated. This error occurs when you use an earlier version of the Assembler to produce the object module. Use the current Assembler to produce a new object module.
Illegal segment address reference type 4	Parts of a procedure address have been separated. This error occurs when you use an earlier version of a Compiler to produce the object module. Use the current Compiler to produce a new object module.

Table A–1 **Linker Messages** (continued)

Message	Explanation/Action
Illegal segment address reference type 5	Your Assembly program uses segment and offset in other than the two allowed ways: - a long CALL instruction - a DD instruction Examine your Assembly code. This error usually results from using a far JMP. This is illegal in an overlay program.
Input file read error, bad object module	You specified an input file that is either corrupt, not a valid object module, or not a library file. Check your file name entry. Make sure your Compiler or Assembler is current.
Module compiled with Publics is not resident	This error message is applicable only for programs generated by the BASIC Compiler. You cannot locate BASIC modules that contain public symbols in overlays. Move the module to the resident segment, or remove the data definitions from the module.
Multiply–defined symbol	The same public symbol is defined in two or more modules; the Linker uses the first definition it encounters and issues this error. You can determine which symbol the Linker encounters first; proceed as follows: **1** List the location of each multiply–defined symbol (use the Librarian). **2** List the object modules in the LINK or BIND command form such that the Linker encounters the symbol first.
Non "CODE" class loaded into overlay	An overlay cannot contain a segment with a class other than CODE. Segments in overlays can contain only executable instructions. The program may run if the affected overlay is not used as an overlay.
Non–contiguous GROUPS not pMode compatible (Selectors nnn and mmm)	This error message is printed when the protected mode requirement that all code segments on all data segments be contiguous is violated. For example, binding modules in which the original order of groups has not been preserved. This message often occurs when binding assembler modules with various compiler–generated modules.

Table A-1 **Linker Messages** (continued)

Message	Explanation/Action
No "OverlayFault" procedure loaded	In a program with overlays, no call to InitOverlays or InitLargeOverlays exists, so the Overlay Handler is not loaded. Add the call to your program.
No run file	You must specify a run file name in the LINK or BIND command form.
No STACK segment	You must provide a stack segment for Assembly language programs. The Linker creates a run file, but the system crashes when you run it.
Odd length STACK	This is a Compiler error; make sure you have the current version. All stack lengths must be an even number of bytes. The Linker adds one byte to the length of any stack that is odd. The run file should execute correctly.
Odd size stack requested; rounded up	You requested an odd-length stack in the stack size parameter of the Linker or Assembler. The Linker adds one byte to the length of any stack that is odd; the run file should execute correctly.
Proc near xxxxx in xxxxx doesn't follow CALL/RET conventions	The Linker call/return conventions have been violated. If the message Illegal segment address reference of type x appears, a fatal error has occurred. Refer to the Explanation/Action for the Illegal segment address reference of type x message. This violation can result from the use of a noncurrent Compiler, from placing a noncurrent run time library module in an overlay, or from an Assembly program with a call/ret problem.
Program size exceeds Linker capacity	Insufficient memory is available to the Linker. There is no fixed limit on the size of the program to be linked, but certain tables built by the Linker must be resident in memory. If these tables cannot be built, this error results.

Table A-1 **Linker Messages** (continued)

Message	Explanation/Action
Relocation offset from group is too large	Your program contains too much data, causing the sum of the data, constant, and stack segments to exceed 64 Kb. This problem can occur: - when you port a large data declaration program from another system - because a Compiler inserts another kind of area between two of these segments - if the memory segment is at the end of a series of segments (although the segment is empty at link time, the Linker checks for this error) The Linker displays the message Segment size exceeds 65520, status code 4405, if any one segment exceeds 64 Kb. The Linker produces an invalid run file. If excessive length causes the problem, dynamically allocate short-lived memory (use AllocMemorySL or, in FORTRAN, reduce data segment lengths by moving variables into common blocks). If the error is caused by non-contiguous segments, use an Assembly program to declare the class names of the segments in a different order and place this module first in the Linker object modules field. This first module serves as a template; the Linker orders segments from the following modules in the same way.
Relocation offset is too large	Refer to the explanation and action for the message Relocation offset from group is too large.

Table A-1 **Linker Messages** (continued)

Message	Explanation/Action
Relocation offset of near reference is too large	The procedure call or data reference uses a 16–bit address, but the target object is too far away to be reached using only 16 bits. A near call requires that the called address be less than 64 Kb from the caller's address and that a 16–bit address be used. The run file produced is invalid. You can make your program smaller, or reorder the object modules to bring references and addresses closer together. If the message identifies a public symbol, you can use it to identify the call. If the message identifies a hexadecimal address, you can examine a compilation list to identify the call. If the caller and called address are from a high level language, this error probably results from a data segment variable reference. If the caller or the called address are in Assembly, change the near call to a far call. If you cannot do this, make sure both addresses are in the same group.
Requested stack size exceeds 64 Kb	You requested a stack size that exceeds 64 Kb. You must reduce your stack requirements.
Segment of absolute or unknown type	All segments must be relocatable. This message can result from using a non–supported Compiler. The run file the Linker produced may be invalid.
Symbol file hash table overflow	The program requires more table space than is currently available to the Linker. The upper limit on the symbol table is 512 sectors or 256 Kb. This message can also appear if you have many long names for public symbols. You must reduce the number of public symbols, or the name length, before the Linker can produce a run file.
Symbol table capacity exceeded	The number of symbols, symbol string lengths, and use of overlays determine the symbol table size. Overlays nearly double the symbol table space required. The symbol table capacity is 512 Kb. You must reduce the number of public symbols, or the name length, before the Linker can produce a run file.

Table A-1 **Linker Messages** (continued)

Message	Explanation/Action
Too many public symbols	Insufficient memory is available. There is no fixed limit on the size of the program to be linked, but certain tables built by the Linker must be resident in memory. If these tables cannot be built, this error results. If you are using the Linker, increase the Linker's available memory or link the files on a workstation with more memory. If you are using the Librarian, divide your library into two libraries. In a library where there are many multiply defined symbols, the symbol table may be of adequate size if you choose to add, delete, or extract modules, but it may be exceeded if you request a listing. To list the symbols, the Librarian must expand the single statement of a multiply defined symbol, creating separate symbols with varying numbers of asterisks. In this process, the symbol table can be exceeded.
Unresolved externals	Your program contains references to external names that do not have public definitions in any other module. Your program contains more than one public definition for a reference and the Linker doesn't know which one to choose. The map file contains an undefined symbol list. The Linker produces a run file. For direct calls, the Linker modifies the call to reference the Debugger. You can run the program; however, the program response is questionable. The system may crash. You should add the definitions to an existing module or provide a new module containing the definitions. **Note:** If you do not specify a version when you are linking the operating system, or any system that uses a version number, this error results. The unresolved external's name will be SBVERRUN in this case.

Table A-2 **Linker Status Codes**

Code	Message	Explanation/Action
200–299	**Cannot open temporary file** **VM read error** **Write error in temporary file** **Write error on list file** **Write error on run file** **Write error on symbol file**	A file system error has occurred; the Linker passes the message from the operating system. The Write error on xxxxx file messages usually result from a full disk. The other messages result from a problem with the temporary file directory ($ directory). Either delete files from the disk to create room or investigate the status of the $ directory to resolve the file system problem.
	Error during legalese	The Linker could not find or could not read the legalese file you specified to append to the run file. The Linker produces a valid run file, missing the legalese portion. **Note:** These error messages, from 200–299, are only samples; the actions/explainations in these samples do not correspond exactly with the messages. For a complete listing, refer to your status codes documentation.
400	**Not enough memory available**	The Linker does not have enough memory available to link the file. To link the file: - If you are running the Linker under the Context Manager, reconfigure the partition size. - Link the run file on a system with more memory.
1380–1390	**Heap errors**	An internal memory management error has occurred. Such an error usually causes the system to stop all activity or to exit to the Executive. If you observe such an error, report it to your Unisys representative.

Table A–2 **Linker Status Codes** (continued)

Code	Message	Explanation/Action
4400	**Attempt to access data outside of segment bounds, possibly bad object module**	If you did not use a segment directive in your Assembly program, or if you declare code or data outside any segment, the Assembler supplies a segment named ??SEG. The resulting object module is invalid and the Linker cannot produce a run file. In Assembly programs, make sure you include a segment directive. This error can also result from a Compiler error.
4402	**Fatal error**	An internal failure has occurred. Report the failure to your Unisys representative.
4403 and 4404	**Too many segment or class names** **Too many segments**	You cannot declare more than 255 segments or different segment names in one module; however, the program can contain more than 255 segments. The Linker does not produce a run file. If necessary, divide the module.
4405	**Segment size exceeds 65520**	Each segment cannot be larger than 65,520 bytes. This error pertains only to a single segment, not to a group or sum of segments (for example, DATA, CONST, and STACK). The link is aborted. The Linker does not produce a run file. If you are writing in Assembly language or Pascal, reduce the size of the segment to less than 65,520.
4406	**Too many groups**	Each module can contain a maximum of 10 groups, and the program can contain a maximum of 256 groups.

Table A-2 **Linker Status Codes** (continued)

Code	Message	Explanation/Action
4407 and 4408	**Too many public symbols in one module** **Too many external symbols in one module**	The Linker does not have sufficient memory to link these modules. To link the file: - If you are running the Linker under the Context Manager, reconfigure the partition size. - Link the run file on a system with more memory.
4409	**Invalid object module**	A file you specified as an object module is not in object module format. This could result from: - Compiler error - damage to the file - specification of a text file (such as the source file) instead of an object module
4411	**Too many common symbols in one module**	The Linker does not have sufficient memory to link the run file. To link the file: - If you are running the Linker under the Context Manager, reconfigure the partition size. - Link the run file on a system with more memory.
4413	**Bad object module, segment, or group index out of range**	You included an invalid object module. Usually this is the result of a Compiler error.
4414	**Too many public procedures in resident overlay**	The resident portion and any single overlay can have a maximum of 4,096 procedures. Divide the code into more overlays.
4418–4420	**Too many segments** **Too many areas**	The Linker does not have sufficient memory to link the run file. To link the file: - If you are running the Linker under the Context Manager, reconfigure the partition size. - Link the run file on a system with more memory.
4422–4423	**Bad object module, external index out of range**	You included an invalid object module. Usually this is the result of a Compiler error.

Software Installation

After you install the Language Development software, you can run the Linker, Librarian, Assembler, Math Server, or Mouse Server by entering commands at the Executive level.

You install the Language Development software from the software diskettes. The diskettes are write-protected; you should not write-enable them or use them as a working copy.

You use the Executive SOFTWARE INSTALLATION command to install the software. You do this by entering the command name in the Executive command line and pressing **GO**. The system then directs the software installation, prompting you when it requires a decision. Before you begin this process, you should review the library file and software installation decision information in this section.

Optional Library Files

The Language Development software includes several library files as listed in table B–1. The files contain object modules necessary for some Linker operations.

You can copy the files to your system as part of the software installation, but you can also use the Executive COPY command to copy the files from the diskette at any time.

You should review the library files before you install the software and decide which ones to copy as part of the software installation.

Software Installation Decisions

Table B–2 lists the commands and libraries that you must decide to add or not add to your system during Language Development software installation.

Table B-1 Language Development Library Files

File Name	Contains
CTOS.lib	operating system run time support
SortMerge.lib	object modules containing external-key and key-in-record sort procedures
Mouse.lib	object modules containing request and procedural interfaces for the 2- and 3-button mouse, cursor control, and tracking

Table B-2 Language Development Software Installation Features and Selections

Item	Executive Command or Library Name	Size
Assembler and SAMGEN	ASSEMBLE	255 sectors
Linker and Librarian	BIND	333 sectors
	LINK	
	LIBRARIAN	
	WRAP	119 sectors
Math Server	INSTALL MATH SERVER	66 sectors
Libraries	Mouse.lib	17 sectors
	CTOS.lib	644 sectors
	SortMerge.lib	155 sectors

Assembler Instruction Format

This appendix describes the instruction format of the processor, and provides a detailed analysis of a sample Assembly language instruction.

The MOD–R/M Byte

The instruction format of the processor uses up to three fields to specify the location of an operand in a register or in memory. The Assembler sets all three fields automatically when it generates code. When used, these fields make up the second byte of an instruction, which is called the MOD–R/M byte.

The two most significant bits of the MOD–R/M byte are the MOD field, which specifies how to interpret the R/M field.

The next three bits are occupied by the REG field, which specifies an 8– or 16– bit register as an operand. Instead of specifying a register, the REG field can, in some instructions, refine the instruction code given in the first byte of an instruction.

The next three bits are occupied by the R/M field, which can specify either a particular register operand, or the addressing mode, to select a memory operand. This occurs in combination with the MOD field.

The MOD and R/M fields determine the effective address (EA) of the memory operand, and the interpretation of successive bytes of the instruction, as follows:

MOD	Interpretation
00	DISP = 0 (disp–low and disp–high are absent)
01	DISP = disp–low sign–extended to 16 bits (disp–high is absent)
10	DISP = disp–high, disp–low
11	There is no DISP (both disp–low and disp–high are absent) and R/M is interpreted as a register

If MOD ≠ 11, R/M is interpreted as follows:

R/M	Interpretation
000	[BX]+[SI]+DISP
001	[BX]+[DI]+DISP
010	[BP]+[SI]+DISP
011	[BP]+[DI]+DISP
100	[SI]+DISP
101	[DI]+DISP
110	[BP]+DISP if MOD ≠ 0
	DISP if MOD = 0
111	[BX]+DISP

If MOD = 11, the effective address is a register designed by R/M. In word instructions, the interpretation is:

R/M	Register
000	AX
001	CX
010	DX
011	BX
100	SP
101	BP
110	SI
111	DI

In byte instructions (W = 0), the interpretation is:

R/M	Register
000	AL
001	CL
010	DL
011	BL
100	AH
101	CH
110	DH
111	BH

Analysis of a Sample Instruction

The Unisys Assembly language makes it possible to convey much information in a single, easy–to–code instruction. The remainder of this appendix provides a detailed description of the following sample instruction:

SUB [BP][SI].field4,CH

The contents of the 8–bit register CH are subtracted from a memory operand; registers BP and SI are used to calculate the address of the memory operand; the identifier field4 and the dot operator(.) are used to designate symbolically an offset within the structure pointed to by BP and SI.

The register BP points within the offset of the run time stack and is used, as is the case in this example, when the operand is on the stack. (The selector register for the stack segment is SS, so the 16–bit contents of SS are automatically used together with BP in addressing the memory operand.)

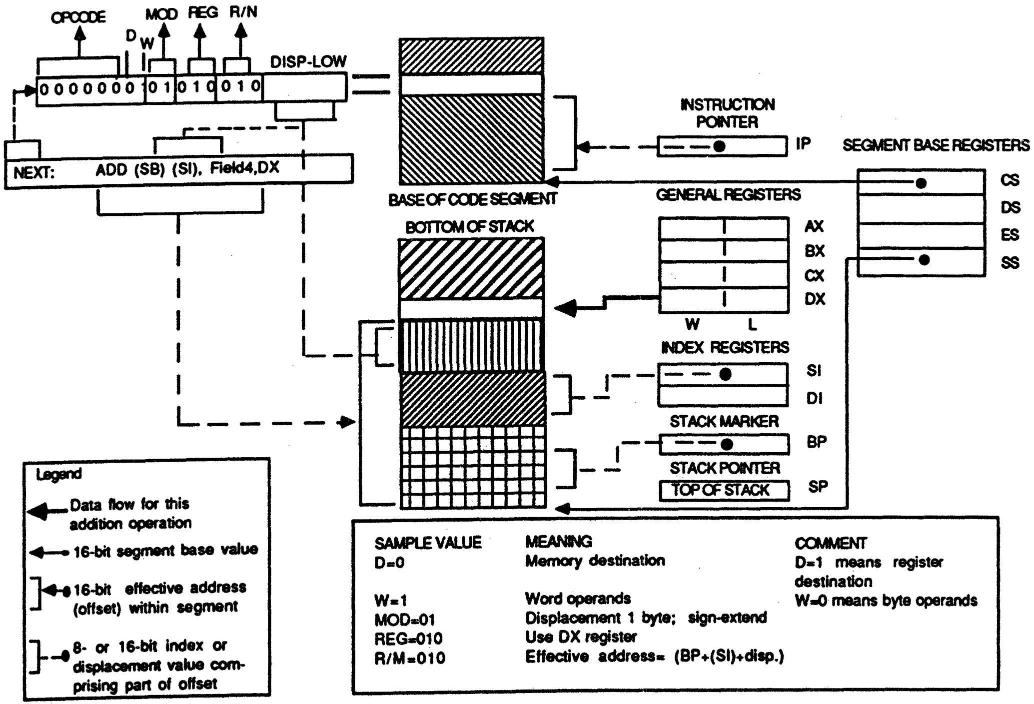
The 16–bit contents of register SI are the data from the top of the stack: the contents of BP and SI are added in the effective address calculation.

In this context, the dot operator (.) refers to a structure. (Refer to section 6 for a description of structure definitions.) The identifier that follows, field4, identifies a structure field. Its value gives the relative distance, in bytes, from the beginning of the structure to field4. (Offset values for each field of the structure relative to the beginning of the structure are generated by the Assembler. In this way the structure can be used as a pattern of relative offset values, a "storage template.")

This instruction combines the contents of the stack segment register SS, the stack base, the index register SI, and the offset of field4, to form an absolute machine address. The contents of the 8–bit register CH are subtracted from the byte thus addressed. This instruction includes opcode, base register, index register, structure displacement and relative offset, type information, direction (register to memory), and source register. The instruction assembles into only three bytes.

Figure C–1 shows a diagram of a sample Assembly language instruction.

Figure C–1 Diagram of a Sample Instruction



SAMPLE VALUE	MEANING	COMMENT
D=0	Memory destination	D=1 means register destination
W=1	Word operands	W=0 means byte operands
MOD=01	Displacement 1 byte; sign-extend	
REG=010	Use DX register	
R/M=010	Effective address= (BP+(SI)+disp.)	

Assembler Instruction Set

This appendix contains four tables:

- Table D–1 lists effective address calculation times.
- Table D–2 lists alternative mnemonics.
- Table D–3 lists the instruction set in numeric order of instruction code.
- Table D–4 lists the instruction set in alphabetical order of instruction mnemonic.

Legend

Tables D–3 and D–4 contain the following seven columns:

- The Op Cd column which is the operand code.
- The Memory Organization column which is explained in appendix C.
- The Instruction column which is the instruction mnemonic.
- The Operand column which contains the operand, if there is one, acted upon by the instruction.
- The Summary column which contains a brief summary of each instruction. Parentheses surrounding an item mean "the contents of." For example, "(EA)" means "the contents of memory location EA," and "(SS)" means "the contents of register SS." The infix operators (+, –, OR, XOR, etc.) denote the standard arithmetic or logical operation. CMP denotes a subtraction in which the result is discarded and only the values of the flags are changed. "TEST" denotes a logical "AND" in which the result is discarded and only the values of the flags are changed.

- The clocks column which is the clock time for each instruction (refer to table D–1). Where two clock times are given in the conditional instructions, the first is the time if the jump (or loop) is performed, and the second if it is not. In all instructions with memory (EA) as one of the operands, a second clock time is given in parentheses. This is because memory may be replaced by a register in all these instructions. In such cases, the faster clock time applies. Where repetitions are possible, a second clock time is also given in parentheses, in the form "x+y/rep", where "x" is the base clock time, "y" is the clock time to be added for each repetition, and "rep" is the number of repetitions.
- The flags column which enumerates the flag conditions, according to this code.

  S = set (to 1)

  C = cleared (to 0)

  X = altered to reflect operation result

  U = undefined (code should not rely on these values)

  R = replaced from memory (e.g., POPF)

  blank = unaffected

  The flags are:

  O = Overflow flag

  D = Direction flag

  I = Interrupt–enable flag

  T = Trap flag

  S = Sign flag

  Z = Zero flag

  A = Auxiliary carry flag

  P = Parity flag

  C = Carry flag

The following symbols are used in the tables:

Symbol	Interpretation
bAddr	16–bit offset within a segment of a word (addressed without use of base or indexing)
bData	byte immediate constant
bEA	effective address of a byte
bREG	8–bit register (AH, AL, BH, CH, CL, DH or DL)
CF	value (0 or 1) of the carry flag
Ext(*b*)	word obtained by sign extending byte *b*
FLAGS	values of the various flags
off	16–bit offset within a selector
Sign(w)	word of all 0's if w is positive, all 1's if w is negative
sba	segment base address
SR	selector register (CS, DS, ES, or SS)
wAddr	16–bit offset within a segment of a word (addressed without use of base or indexing)
wData	word immediate constant
wEA	effective address of a word
wREG	16–bit register (AX, BX, CX, DX, SP, BP, SI, or DI)

Table D–1 Effective Address Calculation Time

EA Components		Clocks
Displacement only		6
Base or index only	(BX, BP, SI, DI)	5
Displacement + Base or Index	(BX, BP, SI, DI)	9
Base + Index	[BP+DI],[BX+SI]	7
	[BP+SI],[BX+DI]	8
Displacement + Base + Index	[BP+DI]+DISP [BX+SI]+DISP	11
	[BP+SI]+DISP [BX+DI]+DISP	12

*Add two clocks for selector override. Add four clocks for each 16–bit word transfer with an odd address.

Alternative Mnemonics

These instructions have synonymous alternative mnemonics as listed in table D–2.

Table D–2 Alternative Mnemonics

Instruction	Synonym	Description
JA	JNBE	Jump if not below or equal
JAE	JNB	Jump if not below
JAE	JNC	Jump if not carry
JB	JNAE	Jump if not above or equal
JB	JC	Jump if carry
JBE	JNA	Jump if not above
JG	JNLE	Jump if not less or equal
JGE	JNL	Jump if not less
JL	JNGE	Jump if not greater or equal
JLE	JNG	Jump if not greater
JNZ	JNE	Jump if not equal
JPE	JP	Jump if parity
JPO	JNP	Jump if no parity
JZ	JE	Jump if equal
LOOPNZ	LOOPNE	Loop (CX) times while not equal
LOOPZ	LOOPE	Loop (CX) times while equal
REPZ	REP	Repeat string operation
REPZ	REPE	Repeat string operation while equal
REPNZ	REPNE	Repeat while (CX) $\neq$ 0 and (ZF) = 1
SHL	SAL	Byte shift EA left 1 bit

Table D-3 Instruction Set in Numeric Order of Instruction Code

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
00	MOD REGR/M	ADD	bEA,REG	(bEA)=(bEA)+(bREG)	16+EA(3)	X XXXXX
01	MOD REGR/M	ADD	wEA,REG	(wEA)=(wEA)+(wREG)	16+EA(3)	X XXXXX
02	MOD REGR/M	ADD	REG,bEA	(bREG)=(bREG)+(bEA)	9+EA(3)	X XXXXX
03	MOD REGR/M	ADD	REG,wEA	(wREG)=(wREG)+(wEA)	9+EA(3)	X XXXXX
04		ADD	AL,bData	(AL)=(AL)+bData	4	X XXXXX
05		ADD	AX,wData	(AX)=(AX)+wData	4	X XXXXX
06		PUSH	ES	Push (ES) onto stack	10	
07		POP	ES	Pop stack to ES	8	
08	MOD REGR/M	OR	bEA,REG	(bEA)=(bEA) OR (bREG)	16+EA(3)	C XXUXC
09	MOD REGR/M	OR	wEA,REG	(wEA)=(wEA) OR (wREG)	16+EA(3)	C XXUXC
0A	MOD REGR/M	OR	REG,bEA	(bREG)=(bREG) OR (bEA)	9+EA(3)	C XXUXC
0B	MOD REGR/M	OR	REG,wEA	(wREG)=(wREG) OR (wEA)	9+EA(3)	C XXUXC
0C		OR	AL,bData	(AL)=(AL) OR bData	4	C XXUXC
0D		OR	AX,wData	(AX)=(AX) OR wData	4	C XXUXC
0E		PUSH	CS	Push (CS) onto stack	11	
0F		(not used)				
10	MOD REGR/M	ADC	EA,REG	(bEA)=(bEA)+(bREG)+CF	16+EA(3)	X XXXXX
11	MOD REGR/M	ADC	EA,REG	(wEA)=(wEA)+(wREG)+CF	16+EA(3)	X XXXXX
12	MOD REGR/M	ADC	REG,EA	(bREG)=(bREG)+(bEA)+CF	9+EA(3)	X XXXXX
13	MOD REGR/M	ADC	REG,EA	(wREG)=(wREG)+(wEA)+CF	9+EA(3)	X XXXXX
14		ADC	AL,bData	(AL)=(AL)+bData+CF	4	X XXXXX
15		ADC	AX,wData	(AX)=(AX)+wData+CF	4	X XXXXX
16		PUSH	SS	Push (SS) onto stack	11	X XXXXX
17		POP	SS	Pop stack to SS	8	
18	MOD REGR/M	SBB	bEA,REG	(bEA)=(bEA)-(bREG)-CF	16+EA(3)	X XXXXX
19	MOD REGR/M	SBB	wEA,REG	(wEA)=(wEA)-(wREG)-CF	16+EA(3)	X XXXXX
1A	MOD REGR/M	SBB	REG,bEA	(bREG)=(bREG)-(bEA)-CF	9+EA(3)	X XXXXX
1B	MOD REGR/M	SBB	REG,wEA	(wREG)=(wREG)-(wEA)-CF	9+EA(3)	X XXXXX
1C		SBB	AL,bData	(AL)=(AL)-bData-CF	4	X XXXXX
1D		SBB	AL,wData	(AX)=(AX)-wData-CF	4	X XXXXX
1E		PUSH	DS	Push (DS) onto stack	10	
1F		POP	DS	Pop stack to DS	8	
20	MOD REGR/M	AND	bEA,REG	(bEA)=(bEA) AND (bREG)	16+EA(3)	C XXUXC
21	MOD REGR/M	AND	wEA,REG	(wEA)=(wEA) AND (wREG)	16+EA(3)	C XXUXC
22	MOD REGR/M	AND	REG,bEA	(bREG)=(bREG) AND (bEA)	9+EA(3)	C XXUXC
23	MOD REGR/M	AND	REG,wEA	(wREG)=(wREG) AND (wEA)	9+EA(3)	C XXUXC
24		AND	AL,bData	(AL)=(AL) AND bData	4	C XXUXC
25		AND	AX,wData	(AX)=(AX) AND wData	4	C XXUXC
26		ES:		ES segment override	2	
27		DAA		Decimal adjust for ADD	4	X XXXXX
28	MOD REGR/M	SUB	bEA,REG	(bEA)=(bEA)-(bREG)	16+EA(3)	X XXXXX
29	MOD REGR/M	SUB	wEA,REG	(wEA)=(wEA)-(wREG)	16+EA(3)	X XXXXX
2A	MOD REGR/M	SUB	REG,bEA	(bREG)=(bREG)-(bEA)	9+EA(3)	X XXXXX
2B	MOD REGR/M	SUB	REG,wEA	(wREG)=(wREG)-(wEA)	9+EA(3)	X XXXXX
2C		SUB	AL,bData	(AL)=(AL)-bData	4	X XXXXX
2D		SUB	AX,wData	(AX)=(AX)-wData	4	X XXXXX
2E		CS:		CS segment override	2	
2F		DAS		Decimal adjust for subtract	4	U XXXXX
30	MOD REGR/M	XOR	bEA,REG	(bEA)=(bEA) XOR (bREG)	16+EA(3)	C XXUXC
31	MOD REGR/M	XOR	wEA,REG	(wEA)=(wEA) XOR (wREG)	16+EA(3)	C XXUXC
32	MOD REGR/M	XOR	REG,bEA	(bREG)=(bREG) XOR (bEA)	9+EA(3)	C XXUXC
33	MOD REGR/M	XOR	REG,wEA	(wREG)=(wREG) XOR (wEA)	9+EA(3)	C XXUXC
34		XOR	AL,bData	(AL)=(AL) XOR bData	4	C XXUXC
35		XOR	AX,wData	(AX)=(AX) XOR wData	4	C XXUXC
36		SS:		SS segment override	2	
37		AAA		ASCII adjust for add	4	U UUXUX
38	MOD REGR/M	CMP	bEA,bREG	FLAGS=(bEA) CMP (bREG)	9+EA	X XXXXX
39	MOD REGR/M	CMP	wEA,wREG	FLAGS=(wEA) CMP (wREG)	9+EA	X XXXXX
3A	MOD REGR/M	CMP	bREG,bEA	FLAGS=(bREG) CMP (bEA)	9+EA	X XXXXX

Table D–3 Instruction Set in Numeric Order of Instruction Code (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
3B	MOD REGR/M	CMP	wREG,wEA	FLAGS=(wREG) CMP (wEA)	9+EA	X XXXXX
3C		CMP	AL,bData	FLAGS=(AL) CMP (bData)	4	X XXXXX
3D		CMP	AX,wData	FLAGS=(AX) CMP (wData)	4	X XXXXX
3E		DS:		DS segment override	2	
3F		AAS		ASCII adjust for subtract	4	U UUXUX
40		INC	AX	(AX)=(AX)+1	2	X XXXX
41		INC	CX	(CX)=(CX)+1	2	X XXXX
42		INC	DX	(DX)=(DX)+1	2	X XXXX
43		INC	BX	(BX)=(BX)+1	2	X XXXX
44		INC	SP	(SP)=(SP)+1	2	X XXXX
45		INC	BP	(BP)=(BP)+1	2	X XXXX
46		INC	SI	(SI)=(SI)+1	2	X XXXX
47		INC	DI	(DI)=(DI)+1	2	X XXXX
48		DEC	AX	(AX)=(AX)-1	2	X XXXX
49		DEC	CX	(CX)=(CX)-1	2	X XXXX
4A		DEC	DX	(DX)=(DX)-1	2	X XXXX
4B		DEC	BX	(BX)=(BX)-1	2	X XXXX
4C		DEC	SP	(SP)=(SP)-1	2	X XXXX
4D		DEC	BP	(BP)=(BP)-1	2	X XXXX
4E		DEC	SI	(SI)=(SI)-1	2	X XXXX
4F		DEC	DI	(DI)=(DI)-1	2	X XXXX
50		PUSH	AX	Push (AX) onto stack	11	
51		PUSH	CX	Push (CX) onto stack	11	
52		PUSH	DX	Push (DX) onto stack	11	
53		PUSH	BX	Push (BX) onto stack	11	
54		PUSH	SP	Push (SP) onto stack	11	
55		PUSH	BP	Push (BP) onto stack	11	
56		PUSH	SI	Push (SI) onto stack	11	
57		PUSH	DI	Push (DI) onto stack	11	
58		POP	AX	Pop stack to AX	8	
59		POP	CX	Pop stack to CX	8	
5A		POP	DX	Pop stack to DX	8	
5B		POP	BX	Pop stack to BX	8	
5C		POP	SP	Pop stack to SP	8	
5D		POP	BP	Pop stack to BP	8	
5E		POP	SI	Pop stack to SI	8	
5F		POP	DI	Pop stack to DI	8	
60		(not used)				
61		(not used)				
62		(not used)				
63		(not used)				
64		(not used)				
65		(not used)				
66		(not used)				
67		(not used)				
68		(not used)				
69		(not used)				
6A		(not used)				
6B		(not used)				
6C		(not used)				
6D		(not used)				
6E		(not used)				
6F		(not used)				
70		JO	bDISP	Jump if overflow	16 or 4	
71		JNO	bDISP	Jump if no overflow	16 or 4	
72		JB	bDISP	Jump if below	16 or 4	
73		JAE	bDISP	Jump if above or equal	16 or 4	
74		JZ	bDISP	Jump if zero	16 or 4	
75		JNZ	bDISP	Jump if not zero	16 or 4	

Table D-3 Instruction Set in Numeric Order of Instruction Code (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
76		JBE	bDISP	Jump if below or equal	16 or 4	
77		JA	bDISP	Jump if above	16 or 4	
78		JS	bDISP	Jump if sign	16 or 4	
79		JNS	bDISP	Jump if no sign	16 or 4	
7A		JPE	bDISP	Jump if parity even	16 or 4	
7B		JPO	bDISP	Jump if parity odd	16 or 4	
7C		JL	bDISP	Jump if less	16 or 4	
7D		JGE	bDISP	Jump if greater or equal	16 or 4	
7E		JLE	bDISP	Jump if less or equal	16 or 4	
7F		JG	bDISP	Jump if greater	16 or 4	
80	MOD 000 R/M	ADD	bEA,bData	(bEA)=(bEA)+bData	17+EA	X XXXXX
80	MOD 001 R/M	OR	bEA,bData	(bEA)=(bEA) OR bData	17+EA	0 XXUX0
80	MOD 010 R/M	ADC	bEA,bData	(bEA)=(bEA)+bData+CF	17+EA	X XXXXX
80	MOD 011 R/M	SBB	bEA,bData	(bEA)=(bEA)-bData-CF	17+EA	X XXXXX
80	MOD 100 R/M	AND	bEA,bData	(bEA)=(bEA) AND bData	17+EA	0 XXUX0
80	MOD 101 R/M	SUB	bEA,bData	(bEA)=(bEA)-bData	17+EA	X XXXXX
80	MOD 110 R/M	XOR	bEA,bData	(bEA)=(bEA) XOR bData	17+EA	0 XXUX0
80	MOD 111 R/M	CMP	bEA,bData	FLAGS=(bEA) CMP bData	10+EA	X XXXXX
81	MOD 000 R/M	ADD	wEA,wData	(wEA)=(wEA)+wData	17+EA	X XXXXX
81	MOD 001 R/M	OR	wEA,wData	(wEA)=(wEA) OR wData	17+EA	0 XXUX0
81	MOD 010 R/M	ADC	wEA,wData	(wEA)=(wEA)+wData+CF	17+EA	X XXXXX
81	MOD 011 R/M	SBB	wEA,wData	(wEA)=(wEA)-wData-CF	17+EA	X XXXXX
81	MOD 100 R/M	AND	wEA,wData	(wEA)=(wEA) AND wData	17+EA	0 XXUX0
81	MOD 101 R/M	SUB	wEA,wData	(wEA)=(wEA)-wData	17+EA	X XXXXX
81	MOD 110 R/M	XOR	wEA,wData	(wEA)=(wEA) XOR wData	17+EA	0 XXUX0
81	MOD 111 R/M	CMP	wEA,wData	FLAGS=(wEA) XOR wData	10+EA	X XXXXX
82	MOD 000 R/M	ADD	bEA,bData	(bEA)=(bEA)+bData	17+EA	X XXXXX
82	MOD 001 R/M		(not used)			
82	MOD 010 R/M	ADC	bEA,bData	(bEA)=(bEA)+bData+CF	17+EA	X XXXXX
82	MOD 011 R/M	SBB	bEA,bData	(bEA)=(bEA)-bData-CF	17+EA	X XXXXX
82	MOD 100 R/M		(not used)			
82	MOD 101 R/M	SUB	bEA,bData	(bEA)=(bEA)-bData	17+EA	X XXXXX
82	MOD 110 R/M		(not used)			
82	MOD 111 R/M	CMP	bEA,bData	FLAGS=(bEA) CMP bData	10+EA	X XXXXX
83	MOD 000 R/M	ADD	wEA,bData	FLAGS=(wEA)+Ext(bData)	17+EA	X XXXXX
83	MOD 001 R/M		(not used)			
83	MOD 010 R/M	ADC	wEA,bData	(wEA)=(wEA)+Ext(bData)+CF	17+EA	X XXXXX
83	MOD 011 R/M	SBB	wEA,bData	(wEA)=(wEA)-Ext(bData)-CF	17+EA	X XXXXX
83	MOD 100 R/M		(not used)			
83	MOD 101 R/M	SUB	wEA,bData	(wEA)=(wEA)-Ext(bData)	17+EA	X XXXXX
83	MOD 110 R/M		(not used)			
83	MOD 111 R/M	CMP	wEA,bData	FLAGS=(wEA) CMP Ext(bData)	10+EA	X XXXXX
84	MOD REGR/M	TEST	bEA,bREG	FLAGS=(bEA) TEST (bREG)	9+EA(3)	0 XXUX0
85	MOD REGR/M	TEST	wEA,wREG	FLAGS=(wEA) TEST (wREG)	9+EA(3)	0 XXUX0
86	MOD REGR/M	XCHG	bREG,bEA	Exchange bREG, bEA	17+EA(4)	
87	MOD REGR/M	XCHG	wREG,wEA	Exchange wREG, wEA	17+EA(4)	
88	MOD REGR/M	MOV	bEA,bREG	(bEA)=(bREG)	9+EA(2)	
89	MOD REGR/M	MOV	wEA,wREG	(wEA)=(wREG)	9+EA(2)	
8A	MOD REGR/M	MOV	bREG,bEA	(bREG)=(bEA)	8+EA(2)	
8B	MOD REGR/M	MOV	wREG,wEA	(wREG)=(wEA)	8+EA(2)	
8C	MOD 0SR R/M	MOV	wEA,SR	(wEA)=(SR)	9+EA(2)	
8C	MOD 1— R/M	(not used)				
8D	MOD REGR/M	LEA	REG,EA	(REG)=effective address	2+EA(2)	
8E	MOD 0SR R/M	MOV	SR,wEA	(SR)=(wEA)	8+EA(2)	
8E	MOD — R/M	(not used)				
8F	MOD 000 R/M	POP	EA	Pop stack to EA	17+EA	
8F	MOD 001 R/M	(not used)				
8F	MOD 010 R/M	(not used)				
8F	MOD 011 R/M	(not used)				

Table D–3 Instruction Set in Numeric Order of Instruction Code (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
8F	MOD 100 R/M	(not used)				
8F	MOD 101 R/M	(not used)				
8F	MOD 110 R/M	(not used)				
8F	MOD 111 R/M	(not used)				
90		XCHG	AX,AX	NOP	3	
91		XCHG	AX,CX	Exchange (AX), (CX)	3	
92		XCHG	AX,DX	Exchange (AX), (DX)	3	
93		XCHG	AX,BX	Exchange (AX), (BX)	3	
94		XCHG	AX,SP	Exchange (AX), (SP)	3	
95		XCHG	AX,BP	Exchange (AX), (BP)	3	
96		XCHG	AX,SI	Exchange (AX), (SI)	3	
97		XCHG	AX,DI	Exchange (AX), (DI)	3	
98		CBW		(AX)=Ext(AL)	2	
99		CWD		(DX)=Sign(AX)	5	
9A		CALL	off:seg	Direct FAR call	28	
9B		WAIT		Wait for TEST signal	3+WAITX	
9C		PUSHF		Push FLAGS onto stack	10	
9D		POPF		Pop stack to FLAGS	8	RRRRRRRRR
9E		SAHF		(FLAGS)=(AH)	4	RRRRRRRRR
9F		LAHF		(AH)=(FLAGS)	4	
A0		MOV	AL,bAddr	(AL)=(bAddr)	10	
A1		MOV	AX,wAddr	(AX)=(wAddr)	10	
A2		MOV	bAddr,AL	(bAddr)=(AL)	10	
A3		MOV	wAddr,AX	(wAddr)=(AX)	10	
A4		MOVSB		Move byte string	18 (9+17/rep)	
A5		MOVSW		Move word string	18 (9+17/rep)	
A6		CMPSB		Compare byte string	22 (9+22/rep)	X XXXXX
A7		CMPSW		Compare word string	22 (9+22/rep)	X XXXXX
A8		TEST	AL,bData	FLAGS=(AL) TEST (bData)	4	X XXUXC
A9		TEST	AX,bData	FLAGS=(AX) TEST (wData)	4	X XXUXC
AA		STOSB		Store byte string	11 (9+10/rep)	
AB		STOSW		Store word string	11 (9+10/rep)	
AC		LODSB		Load byte string	12 (9+13/rep)	
AD		LODSW		Load word string	12 (9+13/rep)	
AE		SCASB		Scan byte string	15 (9+15/rep)	X XXXXX
AF		SCASW		Scan word string	15 (9+15/rep)	X XXXXX
B0		MOV	AL,bData	(AL)←bData	4	
B1		MOV	CL,bData	(CL)←bData	4	
B2		MOV	DL,bData	(DL)←bData	4	
B3		MOV	BL,bData	(BL)←bData	4	
B4		MOV	AH,bData	(AH)←bData	4	
B5		MOV	CH,bData	(CH)←bData	4	
B6		MOV	DH,bData	(DH)←bData	4	
B7		MOV	BH,bData	(BH)←bData	4	
B8		MOV	AX,wData	(AX)←wData	4	
B9		MOV	CX,wData	(CX)←wData	4	
BA		MOV	DX,wData	(DX)←wData	4	
BB		MOV	BX,wData	(BX)←wData	4	
BC		MOV	SP,wData	(SP)←wData	4	

Table D-3 Instruction Set in Numeric Order of Instruction Code (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
BD		MOV	BP,wData	(BP)←wData	4	
BE		MOV	SI,wData	(SI)←wData	4	
BF		MOV	DI,wData	(DI)←wData	4	
C0		(not used)				
C1		(not used)				
C2		RET	wData	NEAR return; (SP)=(SP)+ wData	12	
C3		RET		NEAR return	8	
C4	MOD REG R/M	LES	REG,EA	ES:REG←(wEA+2):(wEA)	16+EA	
C5	MOD REG R/M	LDS	REG,EA	DS:REG←(wEA+2):(wEA)	16+EA	
C6	MOD 000 R/M	MOV	bEA,bData	(bEA)←(bData)	10+EA	
C6	MOD 001 R/M	(not used)				
C6	MOD 010 R/M	(not used)				
C6	MOD 011 R/M	(not used)				
C6	MOD 100 R/M	(not used)				
C6	MOD 101 R/M	(not used)				
C6	MOD 110 R/M	(not used)				
C6	MOD 111 R/M	(not used)				
C7	MOD 000 R/M	MOV	EA,wData	(wEA)←wData	10+EA	
C7	MOD 001 R/M	(not used)				
C7	MOD 010 R/M	(not used)				
C7	MOD 011 R/M	(not used)				
C7	MOD 100 R/M	(not used)				
C7	MOD 101 R/M	(not used)				
C7	MOD 110 R/M	(not used)				
C7	MOD 111 R/M	(not used)				
C8		(not used)				
C9		(not used)				
CA		RET	wData	FAR return, ADD data to REG SP	17	
CB		RET		FAR return	18	
CC		INT	3	Type 3 interrupt	52	CC
CD		INT	bData	Typed interrupt	51	CC
CE		INTO		Interrupt if overflow	53 or 4	CC
(Simple execution of the instruction takes 4 clocks, and actual interrupt, 53.)						
CF		IRET		Return from interrupt	24	RRRRRRRRR
D0	MOD 000 R/M	ROL	bEA,1	Rotate bEA left 1 bit	15+EA	X X
D0	MOD 001 R/M	ROR	bEA,1	Rotate bEA right 1 bit	15+EA	X X
D0	MOD 010 R/M	RCL	bEA,1	Rotate bEA left through carry 1 bit	15+EA	X X
D0	MOD 011 R/M	RCR	bEA,1	Rotate bEA right through carry 1 bit	15+EA	X X
D0	MOD 100 R/M	SHL	bEA,1	Shift bEA left 1 bit	15+EA	X X
D0	MOD 101 R/M	SHR	bEA,1	Shift bEA right 1 bit	15+EA	X X
D0	MOD 110 R/M	(not used)				
D0	MOD 111 R/M	SAR	bEA,1	Shift signed bEA right 1 bit	15+EA	X XXUXX
D1	MOD 000 R/M	ROL	wEA,1	Rotate wEA left 1 bit	15+EA	X X
D1	MOD 001 R/M	ROR	wEA,1	Rotate wEA right 1 bit	15+EA	X X
D1	MOD 010 R/M	RCL	wEA,1	Rotate wEA left through carry 1 bit	15+EA	X X
D1	MOD 011 R/M	RCR	wEA,1	Rotate wEA right through carry 1 bit	15+EA	X X
D1	MOD 100 R/M	SHL	wEA,1	Shift wEA left 1 bit	15+EA	X X
D1	MOD 101 R/M	SHR	wEA,1	Shift wEA right 1 bit	15+EA	X X
D1	MOD 110 R/M	(not used)				
D1	MOD 111 R/M	SAR	wEA,1	Shift signed wEA right 1 bit	15+EA	X XXUXX

Table D–3 Instruction Set in Numeric Order of Instruction Code (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	O	D	I	T	S	Z	A	P	C
D2	MOD 000 R/M	ROL	bEA,CL	Rotate bEA left (CL) bits	20+EA +4/bit	X								X
D2	MOD 001 R/M	ROR	bEA,CL	Rotate bEA right (CL) bits	20+EA +4/bit	X								X
D2	MOD 010 R/M	RCL	bEA,CL	Rotate bEA left through carry (CL) bits	20+EA +4/bit	X								X
D2	MOD 011 R/M	RCR	bEA,CL	Rotate bEA right through carry (CL) bits	20+EA +4/bit	X								X
D2	MOD 100 R/M	SHL	bEA,CL	Shift bEA left (CL) bits	20+EA +4/bit	X								X
D2	MOD 101 R/M	SHR	bEA,CL	Shift bEA right (CL) bits	20+EA +4/bit	X								X
D2	MOD 110 R/M	(not used)												
D2	MOD 111 R/M	SAR	bEA,CL	Shift signed bEA right (CL) bits	20+EA +4/bit	X				X	X	U	X	X
D3	MOD 000 R/M	ROL	wEA,CL	Rotate wEA left (CL) bits	20+EA +4/bit	X								X
D3	MOD 001 R/M	ROR	wEA,CL	Rotate wEA right (CL) bits	20+EA +4/bit	X								X
D3	MOD 010 R/M	RCL	wEA,CL	Rotate wEA left through carry (CL) bits	20+EA +4/bit	X								X
D3	MOD 011 R/M	RCR	wEA,CL	Rotate wEA right through carry (CL) bits	20+EA +4/bit	X								X
D3	MOD 100 R/M	SHL	wEA,CL	Shift wEA left (CL) bits	20+EA +4/bit	X								X
D3	MOD 101 R/M	SHR	wEA,CL	Shift wEA right (CL) bits	20+EA +4/bit	X								X
D3	MOD 110 R/M	(not used)												
D3	MOD 111 R/M	SAR	wEA,CL	Shift signed wEA right (CL) bits	20+EA +4/bit	X				X	X	U	X	X
D4	00001010	AAM		ASCII adjust for multiply	83	U				X	X	U	X	U
D5	00001010	AAD		ASCII adjust for divide	60	U				X	X	U	X	U
D6		(not used)												
D7		XLAT	TABLE	Translate using (BX)	11									
D8	MOD --- R/M	ESC	EA	Escape to external device	8+EA									
E0		LOOPNZ	bDISP	Loop (CX) times while not zero	19 or 5									
E1		LOOPZ	bDISP	Loop (CX) times while zero	18 or 6									
E2		LOOP	bDISP	Loop (CX) times	17 or 5									
E3		JCXZ	bDISP	Jump if (CX)=0	18 or 6									
E4		IN	AL,bPort	Input from bPort to AL	10									
E5		IN	AX,wPort	Input from wPort to AX	10									
E6		OUT	bPort,AL	Output (AL) to bPort	10									
E7		OUT	wPort,AX	Output (AX) to wPort	10									
E8		CALL	wDISP	Direct near call	11									
E9		JMP	wDISP	Direct near jump	7									
EA		JMP	wDISP, wSEG	Direct far jump	7									
EB		JMP	bDISP	Direct near jump	7									
EC		IN	AL,DX	Byte input from port (DX) to REG AL	8									
ED		IN	AX,DX	Word input from port (DX) to REG AX	8									
EE		OUT	DX,AL	Byte output (AL) to port (DX)	8									
EF		OUT	DX,AX	Word output (AX) to port (DX)	8									
F0		LOCK		Bus lock prefix	2									
F1		(not used)												

Table D–3 **Instruction Set in Numeric Order of Instruction Code** (continued)

Op Cd	Memory Organization	Instruction	Operand	Summary	Clocks	Flags ODITSZAPC
F2		REPNZ		Repeat while (CX)≠0 AND (ZF)=0	2	
F3		REPZ		Repeat while (CX)≠0 AND (ZF)=1	2	
F4		HLT		Halt	2	
F5		CMC		Complement carry flag	2	X
F6	MOD 000 R/M	TEST	bEA,bData	FLAGS=(bEA) TEST bData	10+EA	C XXUXC
F6	MOD 001 R/M	(not used)				
F6	MOD 010 R/M	NOT	bEA	Byte invert bEA	16+EA	
F6	MOD 011 R/M	NEG	bEA	Byte negate bEA	16+EA	X XXXXS
;Note: Carry Flag is C if destination is 0.)						
F6	MOD 100 R/M	MUL	bEA	Unsigned multiply by (bEA)	71	X UUUUX
F6	MOD 101 R/M	IMUL	bEA	Signed multiply by (bEA)	90	X UUUUX
F6	MOD 110 R/M	DIV	bEA	Unsigned divide by (bEA)	90	U UUUUU
F6	MOD 111 R/M	IDIV	bEA	Signed divide by (bEA)	112	U UUUUU
F7	MOD 000 R/M	TEST	wEA,wData	FLAGS=(wEA) TEST wData	10+EA	C XXUXC
F7	MOD 001 R/M	(not used)				
F7	MOD 010 R/M	NOT	wEA	Invert wEA	16+EA	
F7	MOD 011 R/M	NEG	wEA	Negate wEA	16+EA	X XXXXS
;Note: Carry Flag is C if destination is 0.)						
F7	MOD 100 R/M	MUL	wEA	Unsigned multiply by (wEA)	124	X UUUUX
F7	MOD 101 R/M	IMUL	wEA	Signed multiply by (wEA)	144	X UUUUX
F7	MOD 110 R/M	DIV	wEA	Unsigned divide by (wEA)	155	U UUUUU
F7	MOD 111 R/M	IDIV	wEA	Signed divide by (wEA)	177	U UUUUU
F8		CLC		Clear carry flag	2	C
F9		STC		Set carry flag	2	S
FA		CLI		Clear interrupt flag	2	C
FB		STI		Set interrupt flag	2	S
FC		CLD		Clear direction flag	2	C
FD		STD		Set direction flag	2	C
FE	MOD 000 R/M	INC	bEA	(bEA)=(bEA)+1	15+EA	X XXXX
FE	MOD 001 R/M	DEC	bEA	(bEA)=(bEA)-1	15+EA	X XXXX
FE	MOD 010 R/M	(not used)				
FE	MOD 011 R/M	(not used)				
FE	MOD 100 R/M	(not used)				
FE	MOD 101 R/M	(not used)				
FE	MOD 110 R/M	(not used)				
FE	MOD 111 R/M	(not used)				
FF	MOD 000 R/M	INC	wEA	(wEA)=(wEA)+1	15+EA	X XXXX
FF	MOD 001 R/M	DEC	wEA	(wEA)=(wEA)-1	15+EA	X XXXX
FF	MOD 010 R/M	CALL	EA	Indirect NEAR call	13+EA	
FF	MOD 011 R/M	CALL	EA	Indirect FAR call	29+EA	
FF	MOD 100 R/M	JMP	EA	Indirect NEAR jump	7+EA	
FF	MOD 101 R/M	JMP	EA	Indirect FAR jump	16+EA	
FF	MOD 110 R/M	PUSH	EA	Push (EA) onto stack	16+EA	
FF	MOD 111 R/M	(not used)				

Table D-4 Instruction Set in Alphabetic Order of Instruction Mnemonic

Instruction	Operand	Summary	Op Code	Memory Organization	Clocks	Flags ODITSZAPC
AAA		ASCII adjust for add	37		4	U UUXUX
AAD		ASCII adjust for divide	D5	00001010	60	U XXUXU
AAM		ASCII adjust for multiply	D4	00001010	83	U XXUXU
AAS		ASCII adjust for subtract	3F		4	U UUXUX
ADC	AL,bData	(AL)=(AL)+bData+CF	14		4	X XXXXX
ADC	AX,wData	(AX)=(AX)+wData+CF	15		4	X XXXXX
ADC	bEA,bData	(bEA)=(bEA)+bData+CF	80	MOD 010 R/M	17+EA	X XXXXX
ADC	wEA,wData	(wEA)=(wEA)+wData+CF	81	MOD 010 R/M	17+EA	X XXXXX
ADC	bEA,bData	(bEA)=(bEA)+bData+CF	82	MOD 010 R/M	17+EA	X XXXXX
ADC	wEA,bData	(wEA)=(wEA)+Ext(bData)+CF	83	MOD 010 R/M	17+EA	X XXXXX
ADC	bEA,REG	(bEA)=(bEA)+(bREG)+CF	10	MOD REG R/M	16+EA(3)	X XXXXX
ADC	wEA,REG	(wEA)=(wEA)+(wREG)+CF	11	MOD REG R/M	16+EA(3)	X XXXXX
ADC	REG,bEA	(bREG)=(bREG)+(bEA)+CF	12	MOD REG R/M	9+EA(3)	X XXXXX
ADC	REG,wEA	(wREG)=(wREG)+(wEA)+CF	13	MOD REG R/M	9+EA(3)	X XXXXX
ADD	AL,bData	(AL)=(AL)+bData	04		4	X XXXXX
ADD	AX,wData	(AX)=(AX)+wData	05		4	X XXXXX
ADD	bEA,REG	(bEA)=(bEA)+(bREG)	00	MOD REG R/M	16+EA(3)	X XXXXX
ADD	wEA,REG	(wEA)=(wEA)+(wREG)	01	MOD REG R/M	16+EA(3)	X XXXXX
ADD	REG,bEA	(bREG)=(bREG)+(bEA)	02	MOD REG R/M	9+EA(3)	X XXXXX
ADD	REG,wEA	(wREG)=(wREG)+(wEA)	03	MOD REG R/M	9+EA(3)	X XXXXX
ADD	bEA,bData	(bEA)=(bEA)+bData	80	MOD 000 R/M	17+EA	X XXXXX
ADD	wEA,wData	(wEA)=(wEA)+wData	81	MOD 000 R/M	17+EA	X XXXXX
ADD	bEA,bData	(bEA)=(bEA)+bData	82	MOD 000 R/M	17+EA	X XXXXX
ADD	wEA,bData	FLAGS=(wEA)+Ext(bData)	83	MOD 000 R/M	17+EA	X XXXXX
AND	AL,bData	(AL)=(AL) AND bData	24		4	C XXUXC
AND	AX,wData	(AX)=(AX) AND wData	25		4	C XXUXC
AND	bEA,REG	(bEA)=(bEA) AND (bREG)	20	MOD REG R/M	16+EA(3)	C XXUXC
AND	wEA,REG	(wEA)=(wEA) AND (wREG)	21	MOD REG R/M	16+EA(3)	C XXUXC
AND	REG,bEA	(bREG)=(bREG) AND (bEA)	22	MOD REG R/M	9+EA(3)	C XXUXC
AND	REG,wEA	(wREG)=(wREG) AND (wEA)	23	MOD REG R/M	9+EA(3)	C XXUXC
AND	bEA,bData	(bEA)=(bEA) AND bData	80	MOD 100 R/M	17+EA	C XXUXC
AND	wEA,wData	(wEA)=(wEA) AND wData	81	MOD 100 R/M	17+EA	C XXUXC
CALL	off:seg	Direct FAR call	9A		28	
CALL	wDISP	Direct NEAR call	E8		11	
CALL	EA	Indirect NEAR call	FF	MOD 010 R/M	13+EA	
CALL	EA	Indirect FAR call	FF	MOD 011 R/M	79+EA	
CBW		(AX)=Ext(AL)	98		2	
CLC		Clear carry flag	F8		2	C (C)
CLD		Clear direction flag	FC		2	C (D)
CLI		Clear interrupt flag	FA		2	C (I)
CMC		Complement carry flag	F5		2	X (C)
CMP	AL,bData	FLAGS=(AL) CMP (bData)	3C		4	X XXXXX
CMP	AX,wData	FLAGS=(AX) CMP (wData)	3D		4	X XXXXX
CMP	bEA,bREG	FLAGS=(bEA) CMP (bREG)	38	MOD REG R/M	9+EA	X XXXXX
CMP	wEA,wREG	FLAGS=(wEA) CMP (wREG)	39	MOD REG R/M	9+EA	X XXXXX
CMP	bREG,bEA	FLAGS=(bREG) CMP (bEA)	3A	MOD REG R/M	9+EA	X XXXXX
CMP	wREG,wEA	FLAGS=(wREG) CMP (wEA)	3B	MOD REG R/M	9+EA	X XXXXX
CMP	bEA,bData	FLAGS=(bEA) CMP bData	80	MOD 111 R/M	10+EA	X XXXXX
CMP	bEA,bData	FLAGS=(bEA) CMP bData	82	MOD 111 R/M	10+EA	X XXXXX
CMP	wEA,wData	FLAGS=(wEA) CMP wData	81	MOD 111 R/M	10+EA	X XXXXX
CMP	wEA,bData	FLAGS=(wEA) CMP Ext(bData)	83	MOD 111 R/M	10+EA	X XXXXX
CMPSB		Compare byte string	A6		22 (9+22/rep)	X XXXXX
CMPSW		Compare word string	A7		22 (9+22/rep)	X XXXXX
CS:		CS segment override	2E		2	
CWD		(DX)=Sign(AX)	99		5	
DAA		Decimal adjust for ADD	27		4	X XXXXX

Table D–4 **Instruction Set in Alphabetic Order of Instruction Mnemonic (continued)**

Instruction	Operand	Summary	Op Code	Memory Organization	Clocks	Flags ODITSZAPC
DAS		Decimal adjust for subtract	2F		4	U XXXXX
DEC	AX	(AX)=(AX)-1	48		2	X XXXX
DEC	BP	(BP)=(BP)-1	4D		2	X XXXX
DEC	BX	(BX)=(BX)-1	4B		2	X XXXX
DEC	CX	(CX)=(CX)-1	49		2	X XXXX
DEC	DI	(DI)=(DI)-1	4F		2	X XXXX
DEC	DX	(DX)=(DX)-1	4A		2	X XXXX
DEC	bEA	(bEA)=(bEA)-1	FE	MOD 001 R/M	15+EA	X XXXX
DEC	wEA	(wEA)=(wEA)-1	FF	MOD 001 R/M	15+EA	X XXXX
DEC	SP	(SP)=(SP)-1	4C		2	X XXXX
DEC	SI	(SI)=(SI)-1	4E		2	X XXXX
DIV	bEA	Unsigned divide by (bEA)	F6	MOD 110 R/M	90	U UUUUU
DIV	wEA	Unsigned divide by (wEA)	F7	MOD 110 R/M	155	U UUUUU
DS:		DS segment override	3E		2	
ES:		ES segment override	26		2	
ESC	EA	Escape to external device	D8	MOD --- R/M	8+EA	
HLT		Halt	F4		2	
IDIV	bEA	Signed divide by (bEA)	F6	MOD 111 R/M	112	U UUUUU
IDIV	wEA	Signed divide by (wEA)	F7	MOD 111 R/M	177	U UUUUU
IMUL	bEA	Signed multiply by (bEA)	F6	MOD 101 R/M	90	X UUUUX
IMUL	wEA	Signed multiply by (wEA)	F7	MOD 101 R/M	144	X UUUUX
IN	AL,DX	Byte input from port (DX) to AEX AL	EC		8	
IN	AL,bPort	Input from bPort to AL	E4		10	
IN	AX,DX	Word input from port (DX) to AEX AX	ED		8	
IN	AX,wPort	Input from wPort to AX	E5		10	
INC	AX	(AX)=(AX)+1	40		2	X XXXX
INC	BP	(BP)=(BP)+1	45		2	X XXXX
INC	BX	(BX)=(BX)+1	43		2	X XXXX
INC	CX	(CX)=(CX)+1	41		2	X XXXX
INC	DI	(DI)=(DI)+1	47		2	X XXXX
INC	DX	(DX)=(DX)+1	42		2	X XXXX
INC	bEA	(bEA)=(bEA)+1	FE	MOD 000 R/M	15+EA	X XXXX
INC	wEA	(wEA)=(wEA)+1	FF	MOD 000 R/M	15+EA	X XXXX
INC	SP	(SP)=(SP)+1	44		2	X XXXX
INC	SI	(SI)=(SI)+1	46		2	X XXXX
INT	bData	Typed interrupt	CD		51	CC
INT	3	Type 3 interrupt	CC		52	CC
INTO		Interrupt if overflow	CE		53 or 4	CC
Simple execution of the instruction takes 4 clocks, and actual interrupt, 53.)						
IRET		Return from interrupt	CF		24	RRRRRRRRR
JA	bDISP	Jump if above	77		16 or 4	
JAE	bDISP	Jump if above or equal	73		16 or 4	
JB	bDISP	Jump if below	72		16 or 4	
JBE	bDISP	Jump if below or equal	76		16 or 4	
JC	(Same as JB, JNAE.)					
JCXZ	bDISP	Jump if (CX)=0	E3		18 or 6	
JE	(Same as JZ.)					
JG	bDISP	Jump if greater	7F		16 or 4	
JGE	bDISP	Jump if greater or equal	7D		16 or 4	
JL	bDISP	Jump if less	7C		16 or 4	
JLE	bDISP	Jump if less or equal	7E		16 or 4	
JMP	bDISP	Direct NEAR jump	EB		7	
JMP	wDISP	Direct NEAR jump	E9		7	
JMP	wDISP, wSEG	EA Direct FAR jump			7	
JMP	EA	Indirect FAR jump	FF	MOD 101 R/M	16+EA	
JMP	wEA	Indirect NEAR jump	FF	MOD 100 R/M	7+EA	

Table D–4 **Instruction Set in Alphabetic Order of Instruction Mnemonic (continued)**

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
JNA	(Same as JBE.)					
JNB	(Same as JAE.)					
JNBE	(Same as JA.)					
JNG	(Same as JLE.)					
JNGE	(Same as JL.)					
JNL	(Same as JGE.)					
JNLE	(Same as JG.)					
JNO	bDISP	Jump if no overflow	71		16 or 4	
JNP	(Same as JPO.)					
JNS	bDISP	Jump if no sign	79		16 or 4	
JNZ	bDISP	Jump if not zero	75		16 or 4	
JO	bDISP	Jump if overflow	70		16 or 4	
JPE	bDISP	Jump if parity even	7A		16 or 4	
JPO	bDISP	Jump if parity odd	7B		16 or 4	
JS	bDISP	Jump if sign	78		16 or 4	
JZ	bDISP	Jump if zero	74		16 or 4	
LAHF		(AH)=(FLAGS)	9F		4	
LDS	REG,EA	DS:REG=(wEA+2):(wEA)	C5	MOD REGR/M	16+EA	
LEA	REG,EA	(REG)=effective address	8D	MOD REGR/M	2+EA(2)	
LES	REG,EA	ES:REG=(wEA+2):(wEA)	C4	MOD REGR/M	16+EA	
LODSB		Load byte string	AC		12 (9+13/rep)	
LODSW		Load word string	AD		12 (9+13/rep)	
LOCK		Bus lock prefix	F0		2	
LOOP	bDISP	Loop (CX) times	E2		17 or 5	
LOOPE	(Same as LOOPZ.)					
LOOPNE	(Same as LOOPNZ.)					
LOOPNZ	bDISP	Loop (CX) times while not zero	E0		19 or 5	
LOOPZ	bDISP	Loop (CX) times while zero	E1		18 or 6	
MOV	bAddr,AL	(bAddr)=(AL)	A2		10	
MOV	wAddr,AX	(wAddr)=(AX)	A3		10	
MOV	AH,bData	(AH)=bData	B4		4	
MOV	AL,bAddr	(AL)=(bAddr)	A0		10	
MOV	AL,bData	(AL)=bData	B0		4	
MOV	AX,wAddr	(AX)=(wAddr)	A1		10	
MOV	AX,wData	(AX)=wData	B8		4	
MOV	BH,bData	(BH)=bData	B7		4	
MOV	BL,bData	(BL)=bData	B3		4	
MOV	BP,wData	(BP)=wData	BD		4	
MOV	BX,wData	(BX)=wData	BB		4	
MOV	CH,bData	(CH)=bData	B5		4	
MOV	CL,bData	(CL)=bData	B1		4	
MOV	CX,wData	(CX)=wData	B9		4	
MOV	DH,bData	(DH)=bData	B6		4	
MOV	DI,wData	(DI)=wData	BF		4	
MOV	DL,bData	(DL)=bData	B2		4	
MOV	DX,wData	(DX)=wData	BA		4	
MOV	bEA,bData	(bEA)=(bData)	C6	MOD 000 R/M	10+EA	
MOV	wEA,wData	(wEA)=(wData)	C7	MOD 000 R/M	10+EA	
MOV	bEA,bREG	(bEA)=(bREG)	88	MOD REGR/M	9+EA(2)	
MOV	wEA,wREG	(wEA)=(wREG)	89	MOD REGR/M	9+EA(2)	
MOV	wEA,SR	(wEA)=(SR)	8C	MOD 0SR R/M	9+EA(2)	
MOV	bREG,bEA	(bREG)=(bEA)	8A	MOD REGR/M	8+EA(2)	
MOV	wREG,wEA	(wREG)=(wEA)	8B	MOD REGR/M	8+EA(2)	
MOV	SI,wData	(SI)=wData	BE		4	
MOV	SP,wData	(SP)=wData	BC		4	
MOV	SR,wEA	(SR)=(wEA)	8E	MOD 0SR R/M	8+EA(2)	

Table D–4 Instruction Set in Alphabetic Order of Instruction Mnemonic (continued)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
MOVS	(Use MOVSB, MOVSW.)					
MOVSB		Move byte string	A4		18 (9+17/rep)	
MOVSW		Move word string	A5		18 (9+17/rep)	
MUL	bEA	Unsigned multiply by (bEA)	F6	MOD 100 R/M	71	X UUUUX
MUL	wEA	Unsigned multiply by (wEA)	F7	MOD 100 R/M	124	X UUUUX
NEG	bEA	Byte negate bEA	F6	MOD 011 R/M	16+EA	X XXXXS
(Note: Carry Flag is C if destination is 0.)						
NEG	wEA	Negate wEA	F7	MOD 011 R/M	16+EA	X XXXXS
(Note: Carry Flag is C if destination is 0.)						
NOP	(Same as XCHG AX,AX)					
NOT	bEA	Byte invert bEA	F6	MOD 010 R/M	16+EA	
NOT	wEA	Invert wEA	F7	MOD 010 R/M	16+EA	
OR	AL,bData	(AL)=(AL) OR bData	0C		4	C XXUXC
OR	AX,wData	(AX)=(AX) OR wData	0D		4	C XXUXC
OR	bEA,bData	(bEA)=(bEA) OR bData	80	MOD 001 R/M	17+EA	C XXUXC
OR	wEA,wData	(wEA)=(wEA) OR wData	81	MOD 001 R/M	17+EA	C XXUXC
OR	bEA,REG	(bEA)=(bEA) OR (bREG)	08	MOD REGR/M	16+EA(3	C XXUXC
OR	wEA,REG	(wEA)=(wEA) OR (wREG)	09	MOD REGR/M	16+EA(3	C XXUXC
OR	REG,bEA	(bREG)=(bREG) OR (bEA)	0A	MOD REGR/M	9+EA(3)	C XXUXC
OR	REG,wEA	(wREG)=(wREG) OR (wEA)	0B	MOD REGR/M	9+EA(3)	C XXUXC
OUT	DX,AL	Byte output (AL) to port (DX)	EE		8	
OUT	DX,AX	Word output (AX) to port (DX)	EF		8	
OUT	bPort,AL	Output (AL) to bPort	E6		10	
OUT	wPort,AX	Output (AX) to wPort	E7		10	
POP	AX	Pop stack to AX	58		8	
POP	BX	Pop stack to BX	5B		8	
POP	BP	Pop stack to BP	5D		8	
POP	CX	Pop stack to CX	59		8	
POP	DI	Pop stack to DI	5F		8	
POP	DS	Pop stack to DS	1F		8	
POP	DX	Pop stack to DX	5A		8	
POP	EA	Pop stack to EA	8F	MOD 000 R/M	17+EA	
POP	ES	Pop stack to ES	07		8	
POP	SI	Pop stack to SI	5E		8	
POP	SP	Pop stack to SP	5C		8	
POP	SS	Pop stack to SS	17		8	
POPF		Pop stack to FLAGS	9D		8	RRRRRRRRR
PUSH	AX	Push (AX) onto stack	50		11	
PUSH	BP	Push (BP) onto stack	55		11	
PUSH	BX	Push (BX) onto stack	53		11	
PUSH	CS	Push (CS) onto stack	0E		11	
PUSH	CX	Push (CX) onto stack	51		11	
PUSH	DI	Push (DI) onto stack	57		11	
PUSH	DS	Push (DS) onto stack	1E		10	
PUSH	DX	Push (DX) onto stack	52		11	
PUSH	EA	Push (EA) onto stack	FF	MOD 110 R/M	16+EA	
PUSH	ES	Push (ES) onto stack	06		10	
PUSH	SI	Push (SI) onto stack	56		11	
PUSH	SP	Push (SP) onto stack	54		11	
PUSH	SS	Push (SS) onto stack	16		11	X XXXXX
PUSHF		Push FLAGS onto stack	9C		10	
RCL	bEA,1	Rotate bEA left thru carry 1 bit	D0	MOD 010 R/M	15+EA	X X
RCL	wEA,1	Rotate wEA left thru carry 1 bit	D1	MOD 010 R/M	15+EA	X X

Table D–4 Instruction Set in Alphabetic Order of Instruction Mnemonic (continued)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
RCR	bEA,CL	Rotate bEA right thru carry (CL) bits	D2	MOD 011 R/M	20+EA +4/bit	X X
RCR	wEA,CL	Rotate wEA right thru carry (CL) bits	D3	MOD 011 R/M	20+EA +4/bit	X X
RCR	bEA,1	Rotate bEA right thru carry 1 bit	D0	MOD 011 R/M	15+EA	X X
RCR	wEA,1	Rotate wEA right thru carry 1 bit	D1	MOD 011 R/M	15+EA	X X
REP	(Same as REPZ.)					
REPE	(Same as REPZ.)	AND (ZF)=1			2	
REPNE	(Same as REPNZ.)					
REPNZ		Repeat while (CX)≠0 AND (ZF)=0	F2		2	
REPZ		Repeat while (CX)≠0	F3			
RET	wData	FAR return, ADD data to REG SP	CA		17	
RET		FAR return	CB		18	
RET		NEAR return	C3		8	
RET	wData	NEAR return; (SP)=(SP)+(wData)	C2		12	
ROL	bEA,CL	Rotate bEA left (CL) bits	D2	MOD 000 R/M	20+EA +4/bit	X X
ROL	wEA,CL	Rotate wEA left (CL) bits	D3	MOD 000 R/M	20+EA +4/bit	X X
ROL	bEA,1	Rotate bEA left 1 bit	D0	MOD 000 R/M	15+EA	X X
ROL	wEA,1	Rotate wEA left 1 bit	D1	MOD 000 R/M	15+EA	X X
ROR	bEA,CL	Rotate bEA right (CL) bits	D2	MOD 001 R/M	20+EA +4/bit	X X
ROR	wEA,CL	Rotate wEA right (CL) bits	D3	MOD 001 R/M	20+EA +4/bit	X X
ROR	bEA,1	Rotate bEA right 1 bit	D0	MOD 001 R/M	15+EA	X X
ROR	wEA,1	Rotate wEA right 1 bit	D1	MOD 001 R/M	15+EA	X X
SAHF		(FLAGS)=(AH)	9E		4	RRRRRRRRR
SAL	(Same as SHL.)					
SAR	bEA,CL	Shift signed bEA right (CL) bits	D2	MOD 111 R/M	20+EA +4/bit	X XXUXX
SAR	wEA,CL	Shift signed wEA right (CL) bits	D3	MOD 111 R/M	20+EA +4/bit	X XXUXX
SAR	bEA,1	Shift signed bEA right 1 bit	D0	MOD 111 R/M	15+EA	X XXUXX
SAR	wEA,1	Shift signed wEA right 1 bit	D1	MOD 111 R/M	15+EA	X XXUXX
SBB	AL,bData	(AL)=(AL)-bData-CF	1C		4	X XXXXX
SBB	AL,wData	(AX)=(AX)-wData-CF	1D		4	X XXXXX
SBB	bEA,bData	(bEA)=(bEA)-bData-CF	80	MOD 011 R/M	17+EA	X XXXXX
SBB	bEA,bData	(bEA)=(bEA)-bData-CF	82	MOD 011 R/M	17+EA	X XXXXX
SBB	wEA,wData	(wEA)=(wEA)-wData-CF	81	MOD 011 R/M	17+EA	X XXXXX
SBB	wEA,bData	(wEA)=(wEA)-Ext(bData)-CF	83	MOD 011 R/M	17+EA	X XXXXX
SBB	bEA,REG	(bEA)=(bEA)-(bREG)-CF	18	MOD REGR/M	16+EA(3)	X XXXXX
SBB	wEA,REG	(wEA)=(wEA)-(wREG)-CF	19	MOD REGR/M	16+EA(3)	X XXXXX
SBB	REG,bEA	(bREG)=(bREG)-(bEA)-CF	1A	MOD REGR/M	9+EA(3)	X XXXXX
SBB	REG,wEA	(wREG)=(wREG)-(wEA)-CF	1B	MOD REGR/M	9+EA(3)	X XXXXX
SCASB		Scan byte string	AE		15 (9+15/rep)	X XXXXX
SCASW		Scan word string	AF		15 (9+15/rep)	X XXXXX
SHL	bEA,CL	Shift bEA left (CL) bits	D2	MOD 100 R/M	20+EA +4/bit	X X

Table D-4 Instruction Set in Alphabetic Order of Instruction Mnemonic (continued)

Instruction	Operand	Summary	Op Cd	Memory Organization	Clocks	Flags ODITSZAPC
SHL	wEA,CL	Shift wEA left (CL) bits	D3	MOD 100 R/M	20+EA +4/bit	X X
SHL	bEA,1	Shift bEA left 1 bit	D0	MOD 100 R/M	15+EA	X X
SHL	wEA,1	Shift wEA left 1 bit	D1	MOD 100 R/M	15+EA	X X
SHR	bEA,CL	Shift bEA right (CL) bits	D2	MOD 101 R/M	20+EA +4/bit	X X
SHR	wEA,CL	Shift wEA right (CL) bits	D3	MOD 101 R/M	20+EA +4/bit	X X
SHR	bEA,1	Shift bEA right 1 bit	D0	MOD 101 R/M	15+EA	X X
SHR	wEA,1	Shift wEA right 1 bit	D1	MOD 101 R/M	15+EA	X X
SS:		SS segment override	36		2	
STC		Set carry flag	F9		2	S
STD		Set direction flag	FD		2	S
STI		Set interrupt flag	FB		2	S
STOSB		Store byte string	AA		11 (9+10/rep)	
STOSW		Store word string	AB		11 (9+10/rep)	
SUB	AL,bData	(AL)=(AL)-bData	2C		4	X XXXXX
SUB	AX,wData	(AX)=(AX)-wData	2D		4	X XXXXX
SUB	bEA,bData	(bEA)=(bEA)-bData	80	MOD 101 R/M	17+EA	X XXXXX
SUB	bEA,bData	(bEA)=(bEA)-bData	82	MOD 101 R/M	17+EA	X XXXXX
SUB	wEA,wData	(wEA)=(wEA)-wData	81	MOD 101 R/M	17+EA	X XXXXX
SUB	wEA,bData	(wEA)=(wEA)-Ext(bData)	83	MOD 101 R/M	17+EA	X XXXXX
SUB	bEA,REG	(bEA)=(bEA)-(bREG)	28	MOD REGR/M	16+EA(3)	X XXXXX
SUB	wEA,REG	(wEA)=(wEA)-(wREG)	29	MOD REGR/M	16+EA(3)	X XXXXX
SUB	REG,bEA	(bREG)=(bREG)-(bEA)	2A	MOD REGR/M	9+EA(3)	X XXXXX
SUB	REG,wEA	(wREG)=(wREG)-(wEA)	2B	MOD REGR/M	9+EA(3)	X XXXXX
TEST	AL,bData	FLAGS=(AL) TEST (bData)	A8		4	X XXUXC
TEST	AX,bData	FLAGS=(AX) TEST (wData)	A9		4	X XXUXC
TEST	bEA,bData	FLAGS=(bEA) TEST bData	F6	MOD 000 R/M	10+EA	C XXUXC
TEST	wEA,wData	FLAGS=(wEA) TEST wData	F7	MOD 000 R/M	10+EA	C XXUXC
TEST	bEA,bREG	FLAGS=(bEA) TEST (bREG)	84	MOD REGR/M	9+EA(3)	C XXUXC
TEST	wEA,wREG	FLAGS=(wEA) TEST (wREG)	85	MOD REGR/M	9+EA(3)	C XXUXC
WAITX		Wait for TEST signal	9B		3+WAITX	
XCHG	AX,AX	NOP	90		3	
XCHG	AX,BP	Exchange (AX), (BP)	95		3	
XCHG	AX,BX	Exchange (AX), (BX)	93		3	
XCHG	AX,CX	Exchange (AX), (CX)	91		3	
XCHG	AX,DI	Exchange (AX), (DI)	97		3	
XCHG	AX,DX	Exchange (AX), (DX)	92		3	
XCHG	AX,SI	Exchange (AX), (SI)	96		3	
XCHG	AX,SP	Exchange (AX), (SP)	94		3	
XCHG	bREG,bEA	Exchange bREG, bEA	86	MOD REGR/M	17+EA(4)	
XCHG	wREG,wEA	Exchange wREG, wEA	87	MOD REGR/M	17+EA(4)	
XLAT	TABLE	Translate using (BX)	D7		11	
XOR	AL,bData	(AL)=(AL) XOR bData	34		4	C XXUXC
XOR	AX,wData	(AX)=(AX) XOR wData	35		4	C XXUXC
XOR	bEA,bData	(bEA)=(bEA) XOR bData	80	MOD 110 R/M	17+EA	C XXUXC
XOR	wEA,wData	(wEA)=(wEA) XOR wData	81	MOD 110 R/M	17+EA	C XXUXC
XOR	bEA,REG	(bEA)=(bEA) XOR (bREG)	30	MOD REGR/M	16+EA(3)	C XXUXC
XOR	wEA,REG	(wEA)=(wEA) XOR (wREG)	31	MOD REGR/M	16+EA(3)	C XXUXC
XOR	REG,bEA	(bREG)=(bREG) XOR (bEA)	32	MOD REGR/M	9+EA(3)	C XXUXC
XOR	REG,wEA	(wREG)=(wREG) XOR (wEA)	33	MOD REGR/M	9+EA(3)	C XXUXC

Assembler Reserved Words for Assembler

The words reserved for use by the Assembly language are listed below.

A
AAA
AAD
AAM
AAS
ABS
ADC
ADD
AH
AL
AND
ARPL
ASSUME
AT
AX
BH
BL
BOUND
BP
BX
BYTE
CALL
CBW
CH
CL
CLC
CLD
CLI
CLTS
CMC
CMP
CMPS
CMPSB
CMPSW
COMMON
CS
CWD
CX
DAA
DAS
DB
DD
DEC
DH
DI
DIV
DL
DS
DUP
DW
DWORD
DX
EJECT
END
ENDP
ENDS
ENTER
EQ
EQU
ES
ESC
EVEN
EXTRN
FAC
FALC
FAR
GE
GEN
GENONLY
GROUP
GT
HIGH
HLT
IDIV
IMUL
IN
INC
INCLUDE
INT
INTO
IRET
JA
JAE
JB
JBE
JC
JCXZ
JE
JGE
JL
JLE
JMP
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
JZ
LABEL
LAHF
LAR
LDS
LE
LEA
LEAVE
LENGTH
LES
LGDT
LIDT
LIST
LLDT
LMSW
LOCK
LODS
LODSB
LODSW
LOOP
LOOPE
LOOPNE
LOOPNZ
LOOPZ
LOW
LSL
LT
MASK
MEMORY
MOD
MOV
MOVS
MOVSB
MOVSW
MUL
NAME
NE
NEAR
NEG
NIL
NOGEN
NOLIST
NOP
NOPAGING
NOT
NOTHING

NOXREF
OFFSET
OR
ORG
OUT
PAGE
PAGELENGTH
PAGEWIDTH
PAGING
PARA
POP
POPA
POPF
PROC
PTR
PUBLIC
PURGE
PUSH
PUSHA
PUSHF
RCL
RCR
RECORD
REP
REPE
REPNE
REPNZ
REPZ
RESTORE
RET
ROL
ROR
SAHF
SAL
SAR
SAVE
SBB
SCAS
SCASB
SCASW
SEG
SEGMENT
SGDT
SHL
SHOR
SHR
SI
SIDT
SIZE
SLDT
SMSW
SP
SS
STACK
STC
STD
STI
STOS
STOSB
STOSW
STR
SUB
TEST
THIS
TITLE
TYPE
VERR
VERW
WAIT
WAITX
WIDTH
WORD
XCHG
XLAT
XLATB
XOR
?
??SEG

Assembly Control Directive

The Unisys Assembly language contains facilities to control the format of the assembly listing and to sequence the reading of "included" source files. These facilities are invoked by assembly control directives. They must occur on one or more separate lines within the source, and cannot be intermixed on the same line as other source code.

An assembly control line must begin with the character "$". Such a line may contain one or more controls, separated by spaces. For example:

$TITLE(Parse Table Generator) PAGEWIDTH(132) EJECT

Description of Directives

Table F–1 lists the meanings of individual directives.

Table F–1 Assembly Control Directives

EJECT	The control line containing EJECT begins a new page.
GEN	All macro calls and macro expansion, including intermediate levels of expansion, appear in the listing.
NOGEN	Only macro calls, not expansions, are listed. However, if an expansion contains an error, it is listed.
GENONLY	Only the final results of macro expansion, and not intermediate expansions or calls, are listed. This is the default mode.
INCLUDE (file)	Subsequent source lines are read from the specified file until the end of the file is reached. At the end of the included file, source input resumes in the original file just after the INCLUDE control line.
LIST	Subsequent source lines appear in the listing.
NOLIST	Subsequent source lines do not appear in the listing.

Table F-1 **Assembly Control Directives** (continued)

PAGELENGTH (*n*)	Pages of the listing are formatted n lines long.
PAGEWIDTH (*n*)	Lines of the listing are formatted a maximum of n characters wide.
PAGING	The listing is separated into numbered pages. This is the default.
NOPAGING	The listing is continuous, with no page breaks inserted.
SAVE	The setting of the LIST/NOLIST flag and the GEN/NOGEN/GENONLY flag is stacked, up to a maximum nesting of 8.
RESTORE	The last SAVEd flags are restored.
TITLE (text)	The text is printed as a heading on subsequent listing pages. The default title is the null string. The text must be parenthetically balanced. (See section 10 for details.)

Using a Printer With Assembly Listings

The listing produced by the Assembler is paginated with titles and form numbers. Since the entire page image is formatted in such a listing, you should print it with an APPEND or COPY to [Lpt] rather than with the Executive's PRINT command.

(You can use the PRINT command to print such a listing, but only by overriding many of its default values. These were chosen to make the printing of text files created with the Editor most convenient.)

Sample Assembler Modules

This section contains three complete sample Assembler modules. The first, shown in figure G–1, is a source module of the Assembler itself. It is the module that translates the Assembler's internal error numbers into textual error messages.

The second module, shown in figure G–2, is a skeleton of a standalone Assembler main program and illustrates how the run time stack is allocated in an Assembler module. This example follows a bare minimum of the standard system conventions and does not link properly to standard object module procedures.

The third module, shown in figure G–3, is an Assembler main program compatible with Unisys conventions and linkable with standard object module procedures, as described in section 11, Accessing Standard Services from Assembly Code.

Figure G-1 Error Message Module Program

```
; Error message module for the assembler
;
; Suitable for loading into an overlay in order to save
space in the resident

PUBLIC pAscizFromErc
; pAsciz = pAscizFromErc(erc, ofUpArrow)
;
; Given an error code in DS:[BP+8] (1st arg.).
;
; Returns ES:BX = pointer to null-terminated ASCII string.
;
; Stores flag indicating whether upArrow is to accompany
error message
; in location pointed to by DS:[BP+6] (2nd arg.).
;
; Define the segments we are going to use here. Do this here
in order to
; get them in the desired physical order.
;
; The storage layout consists of the procedure followed by a
packed group
; of ASCII strings, followed by two parallel arrays.

asmErr SEGMENT WORD PUBLIC 'CODE'     ; Segment for code of
pAscizFromErc
asmErr ENDS

asmEr1 SEGMENT WORD PUBLIC 'ERRORS'   ; Segment for ASCII
text of messages
asmEr1 ENDS

asmEr2 SEGMENT WORD PUBLIC 'ERRORS'   ; Offsets of text,
indexed by erc
rgRaRgCh LABEL WORD
asmEr2 ENDS

asmEr3 SEGMENT WORD PUBLIC 'ERRORS'   ; Array of upArrow
flags, indexed by erc
rgfUpArrow LABEL BYTE
asmEr3 ENDS

; Address everything in this module thru CS (which points to
base of ErrGroup)
ErrGroup GROUP asmErr, asmEr1, asmEr2, asmEr3

asmErr SEGMENT
ASSUME CS:ErrGroup                    ; Tell assembler where
CS will point
```

Figure G–1 Error Message Module Program (continued)

```
pAscizFromErc   PROC FAR                    ; Procedure enty point
      PUSH    BP
      MOV     BP, SP                       ; Save caller's BP, set
up ours

      MOV     BX, [BP+8]                   ; BX = erc
      CMP     BX, ercMax                   ; Check index
      JB      indexOk
      MOV     BX, ercMax - 1               ; index too large, use
internal error msg

indexOk:
      MOV     AL, rgfUpArrow[BX]           ; Fetch upArrow flag
for this erc
      MOV     DI, [BP+6]                   ; Fetch caller's
DS-relative pointer
      MOV     [DI], AL                     ; Store it

      SHL     BX, 1                        ; BX = erc*2 to index
word array
      MOV     BX, rgRaRgch[BX]             ; Fetch CS relative
offset to msg text
      MOV     AX, CS
      MOV     ES, AX                       ; Return segment of
text in ES

      POP     BP
      RET     4
pAscizFromErc   ENDP
asmErr ENDS

asmEr1 SEGMENT
; This macro generates the text and the two arrays
%*DEFINE(Err(fUpArrow, erc, rgch))
(%IF (%erc GT ercMax) THEN (ercMax EQU %erc) FI
orgch   EQU     $               %'Remember where string starts'
        DB    '%rgch',0         %'The null terminated ASCII
string'
asmEr2 SEGMENT
        ORG   %erc*2
        DW    ErrGroup:orgch    %'The errGroup(CS) relative
offset of ASCII text'
asmEr2 ENDS
asmEr3 SEGMENT
        ORG   %erc
        DB    %fUpArrow         %'The upArrow flag'
asmEr3 ENDS
)
```

Figure G–1 Error Message Module Program (continued)

```
; Initialize text and arrays

ercMax EQU 0

%Err(1,00,Invalid numeric constant)
%Err(1,01,Syntax error)
%Err(0,02,Expression too complex)
%Err(0,03,Internal error #1)
%Err(0,04,Invalid arithmetic operation for relocatable or
external expression)
%Err(1,05,Invalid use of register in expression)
%Err(0,06,Invalid use of PTR, must operate upon address
expression)
%Err(1,07,Undefined symbol)
%Err(0,08,Forward reference to EQU''ed register not
permitted)
%Err(0,09,SIZE and LENGTH must operate upon data symbol)
%Err(1,10,Invalid argument to ASSUME, must not be forward
reference)
%Err(0,11,PROC/ENDP nesting too deep)
%Err(0,12,Mismatched PROC/ENDP)
%Err(0,13,Invalid origin for absolute segment)
%Err(0,14,Invalid redefinition of symbol)
%Err(0,15,Mismatched SEGMENT/ENDS)
%Err(0,16,Expression must be absolute)
%Err(0,17,Value too large for field)
%Err(1,18,Strings > 2 characters allowed only in DB)
%Err(0,19,Invalid SEGMENT/GROUP prefix)
%Err(0,20,Label phase error, Pass 2 value differs from Pass
1 value)
%Err(0,21,No ASSUME CS: in effect, NEAR label cannot be
defined)
%Err(0,22,Invalid GROUP member, must be a SEGMENT name)
%Err(0,23,Limit of 255 EXTRN symbols per object module
exceeded)
%Err(0,24,Duplicate declaration for symbol)
%Err(1,25,Not an address expression)
%Err(0,26,Argument to END must be a NEAR/FAR label defined
in this module)
%Err(0,27,Invalid argument to ORG, not absolute or offset)
%Err(0,28,Too many GROUPs)
%Err(0,29,Too many SEGMENTs)
%Err(0,30,Too many GROUP members)
%Err(0,31,SEGMENT nesting too deep)
%Err(0,32,Invalid destination operand)
%Err(0,34,Operand must be a BYTE, WORD or DWORD)
%Err(0,35,Operands not reachable thru segment registers)
%Err(0,36,Too little space reserved due to forward
reference)
%Err(0,37,Invalid combination of index and base registers)
%Err(0,38,Invalid types of operands for this instruction)
```

Figure G-1 Error Message Module Program (continued)

```
%Err(0,39,May not move immediate value to segment register)
%Err(0,40,Invalid shift count)
%Err(0,41,RET outside of PROC/ENDP)
%Err(0,42,Operand must be NEAR or FAR)
%Err(0,43,NEAR jump to different ASSUME CS:)
%Err(0,44,Conditional jump to FAR label)
%Err(0,45,SHORT jump to farther away than 128 bytes)
%Err(0,46,Segment size exceeds 64K bytes)
%Err(0,47,No END statment or open SEGMENT/ENDS PROC/ENDP)
%Err(1,48,Missing right ''%1)'')
%Err(1,49,Invalid character following the Metacharacter)
%Err(0,50,Invalid control)
%Err(0,51,Undefined macro or control)
%Err(1,52,Invalid call pattern)
%Err(1,53,Invalid pattern argument to MATCH)
%Err(1,54,Invalid LOCAL symbol definition)
%Err(0,55,Macro or INCLUDE nesting level too deep)
%Err(0,56,Invalid PAGEWIDTH or PAGELENGTH)
%Err(0,57,SAVE/RESTORE nesting level too deep)
%Err(0,58,RESTORE without matching SAVE)
%Err(0,59,Attempt to redefine builtin function)
%Err(0,60,Macro attempts to redefine itself)
%Err(0,61,Instruction always uses ES:, may not be
overridden)
%Err(0,62,May not index NEAR or FAR expression)
%Err(0,63,Attempt to divide or MOD by 0)
%Err(0,64,Two memory operands are illegal)
%Err(1,65,DUP factor must be positive integer and not
forward reference)
%Err(1,66,Symbol may not be both EXTRN and PUBLIC)
%Err(0,67,Internal Error #2)

asmErl ENDS
END
```

Figure G-2 Standalone Main Program

```
Macro Assembler 8.1.1      13:39 12-Oct-87     Page    1

                                 ; Skeleton main program

                                 Main        SEGMENT     WORD
                                 ASSUME CS: Main

                                 Begin:
                                 ; Put program here, the code
below is hardware specific.  It beeps
                                 ; then shuts up for
one-second intervals.

0000 B040                        Loopx:    MOV       AL, 40h
0002 E644                                  OUT       44h, AL
0004 B9FFFF                                MOV       CX, 0FFFFh
; beeper on for about a second
0007 E2FE                                  LOOP      $

0009 33C0                                  XOR       AX, AX
; faster than MOV AX, 0
000B E644                                  OUT       44h, AL
000D B9FFFF                                MOV       CX, 0FFFFh
; beeper off for about a second
0010 E2FE                                  LOOP      $
0012 EBEC                                  JMP       Loopx
                                 ; End of beeper code

                                 Main      ENDS

                                 Stack     SEGMENT STACK
; must have combine type STACK
0000 (    96                               DW       60h
DUP(?)    ; BTOS requires about 60h word min. stack
     0000)


; to run and use debugger.
                                 Stack     ENDS

                                 END       Begin
; tell assembler where to start



There were no errors detected
```

Figure G-3 Unisys-Compatible Main Program

```
Macro Assembler 8.1.1       15:26 12-Oct-87      Page   1

                                ; Sample main program which links with object module procedures from CTOS.lib
                                ; This program forever outputs to video the string "Now is the time ... "
                                ; followed by an iteration count.

                                ; Declare the OS and object module procedures as external, accessible by
                                ; FAR CALLs
                                EXTRN   WriteBsRecord: FAR, WriteByte: FAR, ErrorExit: FAR

                                ; First declare the code segment so that it is loaded first.  Class = Code
                                ; so that it will be physically near code.  Note that it need not be PUBLIC.

                                Main          SEGMENT    WORD 'Code'
                                Main ENDS

                                ; Next declare the segment which will contain all constant data which will be
                                ; combined with other segment(s) of same name and class
                                Const         SEGMENT    WORD PUBLIC 'Const'

0000 4E6F772069732074           rgchMsg DB 'Now is the time for all good men to come to the aid of their party.'
     68652074696D6520
     666F7220616C6C20
     676F6F64206D656E
     20746F20636F6D65
     20746F2074686520
     616964206F662074
     6865697220706172
     74792E

0043 4300                       cbMsg   DW SIZE rgchMsg ; count of bytes in message

                                Const ENDS

```

Figure G-3 Unisys-Compatible Main Program (continued)

```
                               ; Next declare segment
containing all variable data which will be
                               ; combined with other
segment(s) of same name and class
                               Data          SEGMENT    WORD
PUBLIC 'Data'
                               EXTRN BsVid:    BYTE ; We
write to video using SAM's pre-opened bytestream
                                                    ; which
is located in the data segment.  It is important
                                                    ; to
locate this declaration within the Data SEGMENT/ENDS
                                                    ;
directives as shown here.

0000 0000                      cloop          DW  0
0002 0000                      cbWrittenRet   DW  ?

                               Data ENDS

                               ; Stack segment should have
name and class of Stack to be properly
                               ; combined with other stack
modules (the sizes of which are estimated
                               ; by the compilers).   Space
allocated here need only be sufficient for
                               ; procedures in this module
plus fixed overhead of AT LEAST 60h bytes
                               ; for interrupts and OS
calls.
                               Stack SEGMENT STACK
'Stack' ; note especially combine type = STACK
0000 (   96                                   DW 60h DUP (?)
     0000)

     00C0                      wStackLimit    EQU THIS WORD
; Initial top-of-stack label.  Because

Macro Assembler 8.1.1      15:26 12-Oct-87     Page    2


; of the way the linker combines stack

; segments, this will label the end of the

; combined segments.
                               Stack ENDS

                               Dgroup GROUP Const, Data,
Stack ;
```

Figure G-3 Unisys-Compatible Main Program (continued)

```
All addressing of variable/constants is
; through a group named Dgroup which is known
; to all object modules and must be loaded
; into SS and DS.

                                    ; Begin program code
                                    Main SEGMENT
                                    ASSUME CS: Main
; All code is relative to start of Main
                                    Begin:
0000 B8----                              MOV    AX, Dgroup
; Load Dgroup into SS and DS
0003 8ED0                                MOV    SS, AX
                                    ASSUME SS: Dgroup
; Tell Assembler about new register contents
0005 BCC000                 R            MOV    SP, OFFSET
Dgroup:wStackLimit ; Initialize stack pointer. MUST
; IMMEDIATELY follow the instruction
; which loads SS
0008 8ED8                                MOV    DS, AX
; Load DS register and ...
                                    ASSUME DS: Dgroup
; tell the Assembler about it

                                    Loopx:
                                    ; Call WriteBsRecord(pbsVid,
prgchMsg, cbMsg, pcbWrittenRet)
000A 1E                                  PUSH   DS
; 1st argument is pbsVid
000B 8D060000               E            LEA    AX, bsVid
000F 50                                  PUSH   AX

0010 1E                                  PUSH   DS
; 2nd argument is prgchMsg
0011 8D060000               R            LEA    AX, rgchMsg
0015 50                                  PUSH   AX

0016 FF364300               R            PUSH   cbMsg
; 3rd argument is cbMsg

001A 1E                                  PUSH   DS
; 4th argument is pointer to cbWritten Return
001B 8D060200               R            LEA    AX, cbWrittenRet
001F 50                                  PUSH   AX
0020 9A0000----             E            CALL   WriteBsRecord
0025 23C0                                AND    AX, AX
```

Figure G-3 Unisys-Compatible Main Program (continued)

```
0027 754E                   86          JNE   Error
; Test erc, jump if non-zero
                            87
0029 A10000            R    88          MOV   AX, cloop
002C E81A00                 89          CALL printHex
; print and increment loop count
002F FF060000          R    90          INC   cloop
                            91
                            92     ; Call WriteByte(pbsVid,
0Ah)
0033 1E                     93          PUSH DS
; 1st argument is pointer to bsVid
0034 8D060000          E    94          LEA   AX, bsVid
0038 50                     95          PUSH AX
0039 B00A                   96          MOV   AL, 0Ah
; 2nd argument is char to write to vid
003B 32E4                   97          XOR   AH, AH
; Zero AH
003D 50                     98          PUSH AX
003E 9A0000----        E    99          CALL WriteByte
0043 23C0                  100          AND   AX, AX
0045 7530                  101          JNE   Error
; Test erc, jump if non-zero
                           102

Macro Assembler 8.1.1      15:26 12-Oct-87     Page   3


0047 EBC1                  103          JMP   Loopx
; Loop forever (until ACTION-FINISH)
                           104
                           105     ; Local procedure to convert
number in AX to hex and output it to video
                           106     PrintHex PROC NEAR
0049 B90400                107          MOV   CX, 4
; Initialize digit count
                           108     Printl:
004C 51                    109          PUSH CX
; Save digit count on stack
004D B104                  110          MOV   CL, 4
004F D3C0                  111          ROL   AX, CL
; Position to next digit
0051 50                    112          PUSH AX
; Save this value, the procedure we are about
                           113
; to CALL may clobber any register value !
0052 8BD8                  114          MOV   BX, AX
0054 80E30F                115          AND   BL, 0Fh
; Mask upper nybble of BL
0057 80C330                116          ADD   BL, '0'
; Convert to ASCII
```

Figure G-3 Unisys-Compatible Main Program (continued)

```
005A 80FB39                        CMP  BL, '9'
; Check for hex A..F
005D 7603                          JBE  Print2
; Not above 9
005F 80C307                        ADD  BL, 'A'-'0'-10
                              Print2:
0062 1E                            PUSH DS
; 1st argument is pointer to bsVid
0063 8D060000         E            LEA  AX, bsVid
0067 50                            PUSH AX

0068 53                            PUSH BX
; 2nd argument is char to write
0069 9A0000----       E            CALL WriteByte
006E 23C0                          AND  AX, AX
; Test erc and jump if non-zero
0070 7505                          JNE  Error

0072 58                            POP  AX
; Restore word we are outputting
0073 59                            POP  CX
; Restore loop count
0074 E2D6                          LOOP Print1
; Loop until CX becomes zero
0076 C3                            RET
; Return to main program

                              PrintHex ENDP

                              ; On fatal error AX contains
erc
                              Error:
0077 50                            PUSH AX
; Only argument to ErrorExit is erc
0078 9A0000----       E            CALL ErrorExit

                              Main      ENDS

                              END       Begin
; tell assembler where to start execution



There were no errors detected
```

BTOS Stack Format

This appendix describes the conventional (medium model) stack format used by the BTOS Assembly language. This format (shown in figure H-1) originated with PL/M, and most of the high-level compilers adhere to it.

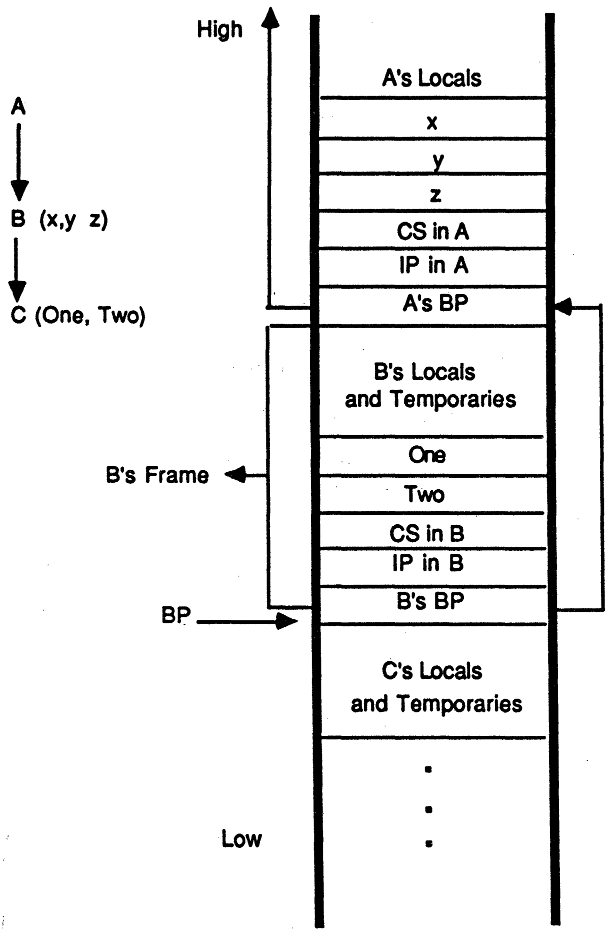
The initial value of the SP (stack pointer) register is at the highest address of the stack. The stack grows down toward lower addresses as objects are pushed onto it. As the stack grows, the address of the top of the stack (SP) becomes smaller. Each location shown on the stack is a word.

Figure H-1 shows two nested procedure calls. Procedure A calls procedure B, which in turn calls procedure C. As indicated in the figure, there is a stack frame for each call. The stack frame consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

When procedure A calls procedure B, the values of A's local variables are on the stack. The passed parameters x, y, and z are pushed in the same order in which they appear in the procedure call. Next, the values of the CS and IP registers are pushed. These represent the point at which execution in A should continue after the return.

Finally, the value of the BP (base pointer) register is pushed. Each stack frame has an associated base pointer. The base pointer is a point of reference from which the called procedure determines where to find needed values of passed or local parameters. For example, in figure H-1, the location of z is BP+6 (using A's BP as a reference point). You can find the first of B's local variables at BP-2.

Figure H-1 **BTOS II Stack Format**



While procedure A was executing, (before it called B), the BP register pointed to the location immediately above A's local variables. When A calls B, that value of the

BP register must be saved for the return, and it is the last item pushed on to the stack (called "A's BP" in figure H-1). Then the BP register is updated to contain the value of SP (the top of the stack), and B's frame begins.

After several calls, there is a chain of BP values marking the various frames. It is possible to trace back through the stack by following this chain from one BP to the previous one, and so on. For example, the Swapper does a stack trace when an overlay is swapped out. It follows the chain of BP values and, by reference from them, corrects the return CS:IP values of any swapped-out procedures to point to the Overlay Manager. The Debugger also does a stack trace when you give a CODE-T command.

For this and other reasons, the stack format must be correct, and the assembly language code must conform to the stack convention.

Stack Frame Prologue and Epilogue

Using the Debugger, you can see the instructions generated by a compiler that immediately precede and follow a procedure call. They are as follows:

Prologue	PUSH BP
	MOV BP,SP
	SUB SP,n

Epilogue	MOV SP, BP POP BP RET mb

In the above prologue, you can see the value of BP (pointer to the frame of the caller) being pushed onto the stack, after which BP is set equal to SP (setting up a pointer to the current frame). In the SUB SP,n instruction, the number of bytes (n) of stack space for local and temporary variables is subtracted from the value of SP. The result is the correct top-of-stack position after the called procedure's local and temporary variables have been placed on the stack.

In the epilogue, the stack pointer is set equal to the base pointer, and the local variables are eliminated from the stack. Then the next location (the value of the previous BP) is popped, after which BP points to its previous location in the BP chain. With the RET (return to CS:IP) instruction, m designates the number of bytes of passed parameters to be popped, leaving SP at the low end of the previous procedure's local variables.

Converting Data or Code Files to Object Modules

There are times when you may need to convert a program or data file into an object module so that you can link it to other object modules to form a new run file. The WRAP command provides this capability.

The WRAP Command

You use the WRAP command (shown in figure I–1) to encapsulate data, code, or other programs in an object module format, which you can then link into an object module using the Linker. (You implement this command using the run file WRAP.run).

For example, if you are writing in assembly language, you can start your source with the folowing statements:

segmentname SEGMENT [PUBLIC] [classname]
PUBLIC [publicname]
data file

The names given for the Segment name and Classname parameters correspond to the names on the SEGMENT state. The name you enter in the Module name field is used by the Librarian to refer to the module. You can use the name entered in Public name as the address of the first byte of the data by specifying this name as External in other modules.

Table I–1 explains each field of the WRAP command.

Figure I-1 WRAP Command Form

```
Wrap
  Data (input) filename
  Object (output) filename
  [Module name]
  [Segment name]
  [Public name]
  [Class]
```

Table I-1 WRAP Command Options

Field	Action/Explanation
Mandatory fields:	
Data (input) filename	Enter the name of the data file whose contents you want to wrap.
Object (output) filename	Enter the name of the object file where you want to place the wrapped data. The default is DataFileName.obj.
Optional fields:	
[Module name]	Enter a name to be used as the internal module name. The default is DataFile Name.
[Segment name]	Enter a name to be used as the internal segment name.
[Public name]	Enter a name to be used as the internal public name. The default is DataFileName
[Class]	Enter a name to be used as the internal class name. The default is DataFileName.

Glossary

Absolute symbol. An absolute symbol is a symbol that has a specified place in memory (as, for example, an address within BTOS).

Address expression. An address expression is a description consisting of one or more symbols, or an indexed or nonindexed parameter.

Alignment attribute. An alignment attribute specifies whether the segment can be aligned on a byte, word, or paragraph boundary.

Application. An application is a program solution to a data processing problem.

Applications. Applications are programs that provide a complete user interface.

Application partition. An application partition is a section of user memory reserved for the execution of an application.

ASCII. ASCII, the American Standard Code for Information Interchange, defines the character set codes used for information exchange between equipment.

Assemble. ASSEMBLE is the Executive command you use to display the Assembler command form.

Assembler. The Assembler translates Assembly 8086 programs into BTOS object modules (machine code).

Assembly. 8086 Assembly is the low level language you can use to write BTOS programs. You use the BTOS Assembler to convert the programs into BTOS object modules.

Asynchronous Terminal Emulator. The Asynchronous Terminal Emulator (ATE) allows a workstation to emulate an asynchronous character-oriented ASCII terminal (glass TTY).

ATE. See Asynchronous Terminal Emulator.

BASIC. BASIC is one of the high level languages you can use to write BTOS programs. You can use the BASIC Compiler to convert the programs into BTOS object modules, or you can use the BASIC Interpreter to edit and run BASIC programs.

Bind. Bind is a command that activates the Linker to create a version 6 run file. Version 6 run files are required for protected mode compatibility.

BSWA. See Byte Stream Work Area.

Byte stream. A byte stream (part of the Sequential Access Method) is a readable or writable sequence of 8-bit bytes.

Byte stream work area. The Byte Stream Work Area (BSWA) is a 130-byte memory work area for the exclusive use of SAM procedures.

Class Name. A class name is a symbol used to designate a class.

Client process. A client process requests system service. Any process can be a client process, since any process can request system service.

C. C is one of the high level languages you can use to write BTOS programs. You can use the C Compiler to convert the programs into BTOS object modules.

COBOL. COBOL is one of the high level languages you can use to write BTOS programs. You can use the COBOL Compiler to convert the programs into BTOS object modules.

Code listing. A code listing is an English-language display of compiled code.

Code segment. A code segment is a variable-length (up to 64Kb) logical entity consisting of reentrant code and containing one or more complete procedures.

Compiler. BTOS Compilers translate high level language programs into BTOS object modules (machine code).

Configuration file. Configuration files specify the characteristics of the parallel printer, serial printer, or other devices attached to a communications channel.

Crash dump. A crash dump is the output (memory dump) resulting from a system failure.

CTOS.lib. The CTOS.lib file is part of the Language Development software; it is a library of object modules that provide operating system run time support.

Cursor RAM. The cursor RAM allows software to specify a 10-bit by 15-bit array as a pattern of pixels in place of the standard cursor.

Customizer. The BTOS Customizer software provides object module files that allow you to customize the operating system.

DAM. See Direct Access Method.

DAWA. See Direct Access Work Area.

DCB. See Device Control Block.

Debugger. The Debugger is a BTOS programming tool that is packaged with the Customizer. It allows you to debug programs written in FORTRAN, Pascal, and Assembly at the symbolic instruction level.

Descriptor Table. A Descriptor Table (only applicable in protected mode) contains descriptors that define the segment's type, length, and protection level.

Device control block. A memory-resident Device Control Block (DCB) exists for each device. The DCB contains device information generated at system build. (For a disk, the information includes the number of tracks and sectors per track.)

DGroup. DGroup usually includes data, constant, and stack Linker segments.

Direct Access Method. The Direct Access Method (DAM) provides random access to disk file records identified by record number. When you create the DAM file, you specify the record size. DAM supports COBOL Relative I/O and any BTOS language program can use a direct call for DAM.

Direct access work area. A Direct Access Work Area (DAWA) is a 64-byte memory work area for the exclusive use of the Direct Access Method (DAM) procedures.

$Directories. When BTOS receives a request with the directory $, the directory name is expanded to $nnn on B24, B26, and B27 workstations and <$000>nnnnn> on B28, B38, and B39 workstations. (nnn and nnnn represent the application user number.)

Double-precision. Double-precision parameters designate two words to store an item of data to maintain a high level of precision.

DS allocation. An option in the Linker, DS allocation locates DGroup at the end of a 64Kb segment that the DS register addresses.

8086 Assembly Language. 8086 Assembly language is the low level language you can use to write BTOS programs. You use the BTOS Assembler to convert the programs into BTOS object modules.

Environment. An environment is a program that has control of the system at any given time. Environments include the SignOn form, the Executive, the Mail Manager, utilities (such as Floppy Copy), applications (such as a word processor), and Compilers.

Escape sequence. An escape sequence is a sequence of characters that activates a function.

Executive. The Executive is the BTOS user interface program; it provides access to many convenient utilities for file management.

External reference. An external reference is a reference from one object module to variables and entry points of other object modules.

Extraction. Librarian extraction copies an object module from a library into a separate disk file. Extraction does not delete the extracted module from the library.

Field. A field is an area in a display form that contains parameters.

File access methods. Several file access methods augment the file management system capabilities. File access methods are object module procedures located in the standard BTOS library. They provide buffering and use the asynchronous input/output capabilities of the file management system to overlap input/output and computation.

Font. The BTOS Font Designer software allows programmers to design or edit characters by drawing or erasing pixels.

Font Designer. The BTOS Font Designer is a program that allows you to design character display fonts that display when your program runs.

Forms. The BTOS Forms software allows programmers to design user-entry forms for applications.

Forms Designer. The BTOS Forms Designer is a program that allows you to develop display forms for user entry when your program runs.

Forms.lib. The Forms.lib file is part of the Language Development software; it is an object module library for Forms Run Time support.

FORTRAN. FORTRAN is one of the high level languages you can use to write BTOS programs. You can use the FORTRAN Compiler to convert the programs into BTOS object modules.

Global Descriptor Table. A Global Descriptor Table contains code and data segments used by the operating system and available to the entire application set.

Group. A group is a named collection of linker segments that the BTOS loader addresses at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64Kb of each other.

High Performance COBOL. See COBOL.

Indexed address. An indexed address is an address expression that uses index registers.

Indexed Sequential Access Method. The BTOS Indexed Sequential Access Method (ISAM) provides random access to fixed-length records identified by multiple keys stored in disk files.

ISAM. See Indexed Sequential Access Method.

Kb. The abbreviation for kilobyte, 1 Kb = app. 1 x 10^3 bytes.

Language Development. The BTOS Language Development software provides the Linker, Librarian, and Assembler programs (LINK, LIBRARIAN, BIND, and ASSEMBLE Executive commands).

LED. LED stands for light-emitting diode (the red light on a keyboard key).

.lib. .lib is the standard file name suffix for library files.

Librarian. The Librarian is a program that creates and maintains object module libraries. The Linker can search automatically in such libraries to select only those object modules that a program calls.

Library. A library is a stored collection of object modules (complete routines or subroutines) that are available for linking into run files.

Library file. A library file contains one or more object modules. The file name normally includes the suffix **.lib.**

Link. LINK is the Executive command that displays the Linker command form.

Linked-list data structure. A linked-list data structure contains elements that link words or link pointers connect.

Linker. The Linker is a program that combines object modules (files that Compilers and Assemblers produce) into run files.

Linker segment. A Linker segment is a single entity consisting of all segment elements with the same segment name.

Link pointer. A link pointer is a 32-bit address that points to the next block of data.

Link word. A link word is a 16-bit address that points to the next block of data.

List file. The Linker list file (suffix .map) contains an entry for each Linker segment, identifying the segment relative address and length in the memory image. You can direct the Linker to list public symbols and line numbers.

Long-lived memory. Long-lived memory is an area of memory in an application partition. It is used for parameters or data passed from an application to a succeeding application in the same partition.

Math Server. The Math Server is a BTOS system service that provides emulation of a numeric coprocessor and the context saving of multiple floating point applications. The context saving is an extension of the BTOS multi-tasking, which allows multiple floating point applications to execute asynchronously.

.map. .map is the standard file name suffix for list files.

Mb. The abbreviation for megabyte, 1 Mb = app. 1 x 10^6 bytes.

Memory array. A memory array is data space the BTOS Loader allocates above the highest task address.

.obj. .obj is the standard file name suffix for object module files.

Object module. An object module is the result of a single Compiler or Assembler function. You can link the object module with other object modules into BTOS run files.

Object Module Procedure. An object module procedure is similar to a system call because it is available through the same mechanism, but it does not interface with the operating system. The task is executed solely by the instructions contained within the object module.

Offset. The offset is the number of bytes between the beginning of a segment and the memory location.

Overlay. An overlay is a code segment made up of the code from one or more object modules. An overlay is loaded into memory as a unit and is not permanently memory-resident. See also Virtual code segment management.

Parameter. A parameter is a variable or constant that is transferred to and from a subroutine or program.

Pascal. Pascal is one of the high level languages you can use to write BTOS programs. You can use the Pascal Compiler to convert the programs into BTOS object modules.

Physical address. A physical address is an address that does not specify a segment base and is relative to memory location 0.

Pixels. Pixels are square-shaped cells which make up the dot matrix of a character symbol.

Pointer. A pointer is an address that specifies a storage location for data.

Process. A process is a program that is running.

Protected Mode. Protected Virtual Address Mode (commonly called protected mode) is a mode of operation of the Intel 80286 and 80386 microprocessors.

Public procedure. A public procedure is a procedure that has a public address; a module other than the defining module can reference the address.

Public symbol. A public symbol is an ASCII character string associated with a public variable, a public value, or a public procedure.

Public value. A public value is a value that has a public address; a module other than the defining module can reference the address.

Public variable. A public variable is a variable that has a public address; a module other than the defining module can reference the address.

Real Mode. Real mode is the only mode of operation for the Intel 8086 and 80186 microprocessors and is the mode of the 80286 and 80386 microprocessors when they are reset. (Refer to Protected Mode.)

Record Sequential Access Method. Record Sequential Access Method (RSAM) files are sequences of fixed-length or variable-length records. You can open the files for read, write, or append operations.

Relocation. The BTOS Loader relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file.

Relocation directory. The relocation directory is an array of locators that the BTOS Loader uses to relocate the task image.

Resident. The resident portion of a program remains in memory throughout execution.

Resident program. A resident program is a program that is fully loaded into memory prior to execution. It contains no overlays and it stays in memory throughout execution.

Reverse video. Reverse video displays dark characters on a light screen.

RSAM. See Record Sequential Access Method.

.run. .run is the standard file name suffix for run files.

Run file. A run file is a complete program: a memory image of a task in relocatable form, linked into the standard format BTOS requires. You use the Linker to create run files.

Run file checksum. The Run-file checksum is a number the Linker produces based on the summation of words in the file. The system uses the checksum to check the validity of the run file.

Run-Time Library. A Run-Time Library is a library (group of object modules) that is used by an application when the application is running.

SAM. See Sequential Access Method.

SamGen. See SAM Generation.

SAM Generation. SAM generation permits the specification of device-dependent object modules to be linked to an application.

Segment. A segment is a contiguous area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind can be either shared or nonshared.

Segment address. The segment address is the segment base address. For an 8086/80186 microprocessor, a segment address refers to a paragraph (16 bytes).

Segmented address. A segmented address is an address that specifies both a segment base and an offset.

Segment element. A segment element is a section of an object module. Each segment element has a segment name.

Segment override. Segment override is operating code that causes the 8086/80186 to use the segment register specified by the prefix instead of the segment register that it would normally use when executing an instruction.

Selector. A selector is the index into a Descriptor Table.

Sequential Access Method. Sequential Access Method (SAM) files emulate a conceptual, sequential character-oriented device known as a byte stream to provide device-independent access to devices.

Short-lived memory. Short-lived memory is the memory area in an application partition. When BTOS loads a task, it allocates short-lived memory to contain the task code and data. A client process can also load short-lived memory in its own partition.

Sort/Merge. Sort/Merge includes a Sort utility and a Merge utility that provide sorting and merging of a sequence of data records.

Stack. A stack is a region of memory accessible from one end by means of a stack pointer.

Stack frame. The stack frame is a region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

Stack pointer. A stack pointer is the indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Submit file escape sequence. A submit file escape sequence consists of two or three characters that indicate the presence of the escape sequence (% or >), followed by a code to identify the special function, followed by an argument to the function.

Swapping program. A swapping program contains a resident program and overlays. The resident part of a swapping program is loaded into memory prior to execution. The overlays are loaded during execution as they are needed.

.sym. .sym is the standard file name suffix for the symbol file.

Symbol. Symbols can be alphanumeric and/or any other characters, such as underscore, period, dollar sign, pound sign, or exclamation mark.

Symbol file. The Linker symbol file (suffix .sym) contains a list of all public symbols.

Symbolic instructions. Symbolic instructions are instructions containing mnenomic characters corresponding to Assembly language instructions. These instructions cannot contain user-defined public symbols.

Sys.Cmds. The Executive command file ([Sys]<sys>Sys.Cmds) contains information on each Executive command.

System build. System build is the collective name for the sequence of actions necessary to construct a customized BTOS image.

System Calls. System calls are subroutines that are provided by BTOS to interface to the operating system.

System Common Access Table (SCAT). SCAT is a table that contains the addresses of structures or information commonly used throughout the operating system and applications.

System image. The system image file ([Sys]<sys>SysImage.Sys) contains a run file copy of BTOS.

System partition. The system partition contains BTOS and dynamically installed system services.

System process. A system process is any process that is not terminated when the user calls Exit.

System Service. A system service is a program that performs a service for other programs. An application notifies a system service that it wants its service performed by issuing a request.

System service process. A system service process is an operating system process that services and responds to requests from client processes.

Task. A task consists of executable code, data, and one or more processes.

Task image. A task image is a program stored in a run file that contains code segments and/or static data segments.

Text file. A text file contains bytes that represent printable characters or control characters (such as tab, new line, etc.).

UCB. See User Control Block.

Unresolved external reference. An unresolved external reference is a public symbol that is not defined, but is used by the modules you are linking.

User control block. The User Control Block (UCB) contains the default volume, directory, password, and file prefix set by the last Set Path or Set Prefix operation.

User process. A user process is any process that is terminated when the user calls Exit.

Utility. A utility is a program provided as part of an operating system; the utility performs standard data–maintenance functions, such as file save and restore, disk compression, and file copy. Other programs can call the utility to perform the task.

Utilities. Utilities are programs that use the Executive user interface (such as Floppy Copy or Ivolume).

Version 4 Run Files. Version 4 run files are run files that have been linked with the Linker's Link command. Version 4 run files are not protected mode compatible.

Version 6 Run Files. Version 6 run files are run files that have been linked with the Linker's Bind command.

Video attributes. Video attributes control the presentation of characters on the display.

Virtual code segment management. Virtual code segment management is the virtual memory method BTOS supports. The method works as follows: The Linker divides the code into task segments that reside on disk (in the run file). As the run file executes, only the task segments that are required at a particular time reside in the application partition's main memory; the other task segments remain on disk until the application requires them. When the application no longer requires a task segment, another task segment overlays it.

Index

A

D

N

O

T

U

V

Help Us To Help You

Publication Title

Form Number Date

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments

Name

Title Company

Address (Street, City, State, Zip)

Telephone Number

Help Us To Help You

Publication Title

Form Number Date

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments

Name

Title Company

Address (Street, City, State, Zip)

Telephone Number

No Postage
necessary
if mailed in the
United States

BUSINESS REPLY MAIL

First Class Permit No. 817 Detroit, MI 48232

Postage Will Be Paid By Addressee

Unisys Corporation
ATTN: Corporate Product Information
P.O. Box 418
Detroit, MI 48232-9975 USA

No Postage
necessary
if mailed in the
United States

BUSINESS REPLY MAIL

First Class Permit No. 817 Detroit, MI 48232

Postage Will Be Paid By Addressee

Unisys Corporation
ATTN: Corporate Product Information
P.O. Box 418
Detroit, MI 48232-9975 USA