CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

NCR DOS

PROGRAMMERS GUIDE.

CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

4.1 TYPICAL MS-DOS MEMORY MAP

Interrupt vector table

Optional extra space (used by NCR for ROM data area)

IO.SYS - MS-DOS interface to hardware

MSDOS.SYS - MS-DOS interrupt handlers, service routines (Interrupt 21H functions)

MS-DOS buffers, control areas, and installed device drivers

Resident part of COMMAND.COM - Interrupt handlers for Interrupts 22H (Terminate Process Exit Address), 23H (Ctrl-Break Handler Address), 24H (Critical Error Handler Address) and code to reload the transient part

External command or utility - (.COM or .EXE file)

User stack for .COM files (256 bytes)

Transient part of COMMAND.COM - Command interpreter, internal commands, batch processor

User memory is allocated from the lowest end of available memory that will meet the allocation request.

4.2 MS-DOS PROGRAM SEGMENT

When an external command is typed, or when you execute a program through the EXEC system call, MS-DOS determines the lowest available free memory address to use as the start of the program. This area is called the Program Segment.

The first 256 bytes of the Program Segment are set up by the EXEC system call for the program being loaded into memory. The program is then loaded following this block. An .EXE file with minalloc and maxalloc both set to zero is loaded as high as possible.

At offset 0 within the Program Segment, MS-DOS builds the Program Segment Prefix control block. The program returns from EXEC by one of five methods:

1. By issuing an Interrupt 21H with AH=4CH

2. By issuing an Interrupt 21H with AH=31H (Keep Process)

3. A long jump to offset 0 in the Program Segment Prefix

4. By issuing an Interrupt 20H with CS:0 pointing at the PSP

5. By issuing an Interrupt 21H with register AH=0 and with CS:0 pointing at the PSP.

Note

Methods 1 and 2 are preferred for both functionality and best operation in future versions of MS-DOS.

All five methods result in transferring control to the program that issued the EXEC. Using method 1 or 2 allows a completion code to be returned. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Process Exit Address, Ctrl-Break Handler Address, and CriticalError Handler Address) addresses are restored from the values saved in the Program Segment Prefix of the terminating program. Control is then given to the terminate address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued; otherwise, COMMAND.COM performs a checksum on the transient part, reloads it if necessary, then issues the system prompt and waits for you to type the next command.

When a program receives control, the following conditions are in effect:

For all programs:

The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.

The environment is a series of ASCII strings (totaling less than 32K) in the form:

NAME=parameter

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros.

Following the last byte of zeros is a set of initial arguments passed to a program that contains a word count followed by an ASCIZ string. If the file is found in the current directory, the ASCIZ string contains the drive and pathname of the executable program as passed to the EXEC function call. If the file is found in the path, the filename is concatenated with the information in the path. Programs may use this area to determine where the program was loaded.

The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last Path and Prompt commands issued will also be in the environment, along with any environment strings defined with the MS-DOS Set command.

The environment that is passed is a copy of the invoking process environment. If your application uses a "keep process" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent Set, Path, or Prompt commands are issued. Conversely, any modification of the passed environment by the application will not be reflected in the parent process environment. For instance, a program cannot change the MS-DOS environment values as the Set command does.

The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix). At 5CH and 6CH in the Program Segment Prefix are file control blocks. These are formatted from the first two parameters, typed when the command was entered. If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) will not appear in this area; redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)

AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

Offset 2 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

For Executable (.EXE) programs:

DS and ES registers are set to point to the Program Segment Prefix.

CS,IP,SS, and SP registers are set to the values set by MS-LINK in the .EXE image.

For Executable (.COM) programs:

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.

The Instruction Pointer (IP) is set to 100H.

The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

A word of zeros is placed on top of the stack. This is to allow a user program to exit to COMMAND.COM by doing a RET instruction last. This assumes, however, that the user has maintained his stack and code segments.

Figure 4.1 illustrates the format of the Program Segment Prefix. All offsets are in hexadecimal.

```
                         (Offsets in Hex)
0 ----------------------------------------------------------
  |               | End of      |Reser-| Long        Offset add |
  |   INT 20H     | alloc.      |ved   | call        Function   |
  |               | block       |      |(5 bytes)    dispatcher |
8 ----------------------------------------------------------
  |Segment addr.|  Terminate address    |  Ctrl-Break exit |
  |Function     |      (IP, CS)         |   address (IP)   |
  |dispatcher   |                       |                  |
10---------------------------------------------------------
  |Ctrl-Break  | Hard error exit address |                  |
  |exit        |       (IP, CS)          |                  |
  |address (CS)|                         |                  |
  -----------------------------------------                 |
  |                                                         |
  |                    Used by MS-DOS                       |
  |                                                         |
  |                        5CH                              |
  |                                                         |
  -----------------------------------------------------------
  |                                                         |
  |   Formatted Parameter Area 1 formatted as standard      |
  |                  unopened FCB 6CH                       |
  -----------------------------------------------------------
  |                                                         |
  |   Formatted Parameter Area 2 formatted as standard      |
  |    unopened FCB (overlaid if FCB at 5CH is opened)      |
80---------------------------------------------------------
  |             Unformatted Parameter Area                  |
  |            (default Disk Transfer Area)                 |
  |        Initially contains command invocation line.      |
  -----------------------------------------------------------
100
```

Figure 4.1. Program Segment Prefix

```
--------------------------------------------------------------
|                          Important                         |
|                                                            |
| Programs must not alter any part of the Program Segment    |
| Prefix below offset 5CH.                                   |
|____________________________________________________________|
```

CHAPTER 5

.EXE FILE STRUCTURE AND LOADING

CHAPTER 5

.EXE FILE STRUCTURE AND LOADING

> **Note**
>
> This chapter describes the .EXE file structure. Use Function Request 4B00H, Load and Execute a Program, to load (or load and execute) an .EXE file.

The .EXE files produced by MS-LINK consist of two parts:

Control and relocation information

The load module

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

The header is formatted as follows. (Note that offsets are in hexadecimal.)

Offset	Contents
00-01	Must contain 4DH, 5AH.
02-03	Number of bytes contained in last page; this is useful in reading overlays.
04-05	Size of the file in 512-byte pages, including the header.

06-07	Number of relocation entries in table.
08-09	Size of the header in 16-byte paragraphs. This is used to locate the beginning of the load module in the file.
0A-0B	Minimum number of 16-byte paragraphs required above the end of the loaded program.
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program. If both minalloc and maxalloc are 0, then the program will be loaded as high as possible.
0E-0F	Initial value to be loaded into stack segment before starting program execution. This must be adjusted by relocation.
10-11	Value to be loaded into the SP register before starting program execution.
12-13	Negative sum of all the words in the file.
14-15	Initial value to be loaded into the IP register before starting program execution.
16-17	Initial value to be loaded into the CS register before starting program execution. This must be adjusted by relocation.
18-19	Relative byte offset from beginning of run file to relocation table.

1A-1B	The number of the overlay as generated by MS-LINK.

The relocation table follows the formatted area described above. This table consists of a variable number of relocation items. Each relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control. The following steps describe this process:

1. The formatted part of the header is read into memory. Its size is 1BH.

2. A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.

3. A Program Segment Prefix is built in the lowest part of the allocated memory.

4. The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5. The load module is read into memory beginning with the start segment.

6. The relocation table items are read into a work area.

7. Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

8. Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

CHAPTER 6

INTEL RELOCATABLE OBJECT MODULE FORMATS

CHAPTER 6

INTEL RELOCATABLE OBJECT MODULE FORMATS

6.1 INTRODUCTION

This chapter presents the object record formats that define the relocatable object language for the 8086 microprocessor. The 8086 object language is the output of all language translators that have the 8086 as the target processor and are to be linked using the Microsoft Linker. The 8086 object language is input and output for object language processors such as linkers and librarians.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. Capabilities are provided that allow efficient use of the memory mapping facilities of the 8086 microprocessor.

The following table lists the record formats that are supported by Microsoft. These record formats are described in this chapter. Record formats that are preceded by an asterisk (*) deviate from the Intel(R) specification.

Table 6.1 Object Module Record Formats

```
    T-MODULE HEADER RECORD
    LIST OF NAMES RECORD
   *SEGMENT DEFINITION RECORD
   *GROUP DEFINITION RECORD
   *TYPE DEFINITION RECORD

 Symbol Definition Records
   *PUBLIC NAMES DEFINITION RECORD
   *EXTERNAL NAMES DEFINITION RECORD
   *LINE NUMBERS RECORD

 Data Records
    LOGICAL ENUMERATED DATA RECORD
    LOGICAL ITERATED DATA RECORD

    FIXUP RECORD
   *MODULE END RECORD
    COMMENT RECORD
```

6.2 DEFINITION OF TERMS

The following terms are fundamental to the 8086 relocation and linkage.

OMF - Object Module Formats.

MAS - Memory Address Space. The 8086 MAS is 1 megabyte (1,048,576). Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE - an "inseparable" collection of object code and other information produced by a translator.

T-MODULE - A module created by a translator, such as Pascal or FORTRAN.

The following restrictions apply to object modules:

1. Every module should have a name. Translators will provide names for T-modules, providing a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.

2. Every T-module in a collection of linked modules must have a different name, so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by the Linker, and is not enforced by it.

FRAME - A contiguous region of 64K of MAS, beginning on a paragraph boundary (i.e., on a multiple of 16 bytes). This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) FRAMEs; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAMEs.

LSEG - Logical Segment - A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS are necessarily determined at translation time: size, although partially fixed, may not be final because the LSEG may be combined at LINK time with other LSEGs, forming a single LSEG. An LSEG must not be larger than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG.

PSEG - Physical Segment - This term is equivalent to FRAME. Some people prefer "PSEG" to "FRAME" because the terms "PSEG" and "LSEG" reflect the "physical" and "logical" nature of the underlying segments.

FRAME NUMBER - Every FRAME begins on a paragraph boundary. The "paragraphs" in MAS can be numbered from 0 through 65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

PARAGRAPH NUMBER - This term is equivalent to FRAME NUMBER.

PSEG NUMBER - This term is equivalent to FRAME NUMBER.

GROUP - A collection of LSEGs defined at translation time, whose final locations in MAS have been constrained such that there will be at least one FRAME that covers (contains) every LSEG in the collection.

The notation "Gr A(X,Y,Z,)" means that LSEGs X, Y and Z form a group whose name is A. The fact that X, Y and Z are all LSEGs in the same group <u>does not imply any ordering of X, Y and Z in MAS, nor does it imply any contiguity between X, Y and Z</u>.

The Microsoft Linker does not currently allow an LSEG to be a member of more than one group. The Linker will ignore all attempts to place an LSEG in more than one group.

CANONIC - Any location in MAS is contained in exactly 4096 distinct FRAMEs; but one of these FRAMEs can be distinguished because it has a higher FRAME NUMBER. This distinguished FRAME is called the canonic FRAME of the location. In other words, the canonic frame of a given byte is the frame so chosen that the byte´s offset from that frame lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic FRAME of FOO", or of "FOO´s canonic FRAME". By extension, if S is any set of memory locations, then there exists a unique FRAME which has the lowest FRAME NUMBER in the set of canonic FRAMEs of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG or of a group of LSEGs.

SEGMENT NAME - LSEGs are assigned segment names at translation time. These names serve two purposes:

1. They play a role at LINK time in determining which LSEGs are combined with other LSEGs.

2. They are used in assembly source code to specify groups.

CLASS NAME - LSEGs may optionally be assigned Class Names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same Class Name.

The Microsoft Linker applies the following semantics to class names. The class name "CODE" or any class name whose suffix is "CODE" implies that all segments of said class contain only code and may be considered read-only. Such segments may be overlayed if the user specifies the module containing the segment as part of an overlay.

OVERLAY NAME - LSEGs may optionally be assigned an overlay name. The overlay name of an LSEG is ignored by MS-LINK (version 2.40 and later versions), but it is used by Intel relocation and Linkage products.

COMPLETE NAME - The complete name of an LSEG consists of the Segment Name, Class Name, and Overlay Name. LSEGs from different modules will be combined if their Complete Names are identical.

6.3 MODULE IDENTIFICATION AND ATTRIBUTES

A module header record is always the first record in a module. It provides a module name.

In addition to a name, a module may have the attribute of being a main program as well as having a specified starting address. When linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

6.4 SEGMENT DEFINITION

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called LOGICAL SEGMENTS (LSEGs). A module defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, etc.) is maintained. The LSEG information is required when multiple LSEGs are combined and when segment addressability (See Section 6.5, "Segment Addressing") is established. The SEGDEF records are required to follow the first header record.

6.5 SEGMENT ADDRESSING

The 8086 addressing mechanism provides segment base registers from which a 64K-byte region of memory, called a FRAME, may be addressed. There is one code segment base register (CS), two data segment base registers (DS, ES), and one stack segment base register (SS).

The possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would be the case in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that will fit in one memory frame so that all the LSEGs may be addressed using the same base register value. This addressable unit is a GROUP and has been defined earlier in Section 6.2, "Definition of Terms."

To allow addressability of objects within a GROUP to be established, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol FOO" or "the segments with class name ROM."

The GRPDEF records within a module must follow all SEGDEF records as GRPDEF records may reference SEGDEF records in defining a GROUP. The GRPDEF records must also precede all other records except header records, as the Linker must process them first.

6.6 SYMBOL DEFINITION

MS-LINK supports three different types of records that fall into the class of symbol definition records. The two most important types are PUBLIC NAMES DEFINITION RECORDs (PUBDEFs) and EXTERNAL NAMES DEFINITION RECORDS (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by MS-LINK for the allocation of communal variables (see Section 6.14 "Microsoft Type Representations for Communal Variables").

6.7 INDICES

"Index" fields occur throughout this document. An index is an integer that selects some particular item from a collection of such items. (List of examples: NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, TYPE INDEX.)

> **Note**
>
> An index is normally a positive number. The index value zero is reserved, and may carry a special meaning dependent upon the type of index (e.g., a Segment Index of zero specifies the "Unnamed," absolute pseudo-segment; a Type Index of zero specifies the "Untyped type", which is different from "Decline to state").

In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in one or two bytes, as required.

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

6.8 CONCEPTUAL FRAMEWORK FOR FIXUPS

A "fixup" is some modification to object code, requested by a translator, performed by the Linker, achieving address binding.

> **Note**
>
> This definition of "fixup" accurately represents the viewpoint maintained by the Linker. Nevertheless, the Linker can be used to achieve modifications of object code (i.e., "fixups") that do not conform to this definition. For example, the binding of code to either hardware floating point or software floating point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of "fixup" is not intended to disallow or disparage object code modifications.

8086 translators specify a fixup by giving four data:

1. The place and type of a LOCATION to be fixed up.

2. One of two possible fixup MODEs.

3. A TARGET, which is a memory address to which LOCATION must refer.

4. A FRAME defining a context within which the reference takes place.

LOCATION - There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.)

1. A BASE is the high-order word of a pointer (and the Linker doesn't care if the low-order word of the pointer is present or not).

2. An OFFSET is the low-order word of a pointer (and the Linker doesn't care if the high-order word follows or not).

3. A HIBYTE is the high-order half of an OFFSET (and the Linker doesn't care if the low-order half precedes or not).

4. A LOBYTE is the low-order half of an OFFSET (and the Linker doesn't care if the high-order half follows or not).

```
Pointer:  +----+----+----+----+
          |                   |
          +----+----+----+----+

Base:               +----+----+
                    |         |
                    +----+----+

Offset:   +----+----+
          |         |
          +----+----+

Hibyte:        +----+
               |    |
               +----+

Lobyte:   +----+
          |    |
          +----+
```

Figure 6.1. LOCATION Types

A LOCATION is specified by two data: (1) the LOCATION type, and (2) where the LOCATION is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUP record.

MODE - The Linker supports two kinds of fixups: "self-relative" and "segment-relative."

Self-relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-relative fixups support all other addressing modes of the 8086.

TARGET - The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered to be the lowest byte in the object being referenced.) A TARGET is specified in one of eight ways. There are four "primary" ways, and four "secondary" ways. Each primary way of specifying a TARGET uses two kinds of data: an INDEX-or-FRAME-NUMBER 'X', and a displacement 'D'.

(T0) X is a SEGMENT INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T1) X is a GROUP INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.

(T2) X is an EXTERNAL INDEX. The TARGET is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the INDEX.

(T3) X is a FRAME NUMBER. The TARGET is the Dth byte in the FRAME identified by the FRAME NUMBER (i.e., the address of TARGET is (X*16)+D).

Each secondary way of specifying a TARGET uses only one data item: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed.

(T4) X is a SEGMENT INDEX. The TARGET is the 0th (first) byte in the LSEG identified by the INDEX.

(T5) X is a GROUP INDEX. The TARGET is the 0th (first) byte in the LSEG in the specified group that is eventually LOCATEd lowest in MAS.

(T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is the External Name identified by the INDEX.

(T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is (X*16).

> **Note**
>
> The Microsoft Linker does not support methods T3 and T7.

The following nomenclature is used to describe a TARGET:

TARGET:	SI(<segment name>), <displacement>	[T0]
TARGET:	GI(<group name>), <displacement>	[T1]
TARGET:	EI(<symbol name>), <displacement>	[T2]
TARGET:	SI (<segment name>)	[T4]
TARGET:	GI (<group name>)	[T5]
TARGET:	EI (<symbol name>)	[T6]

The following examples illustrate how this notation is used:

TARGET:	SI(CODE), 1024	The 1025th byte in the segment "CODE".
TARGET:	GI(DATAAREA)	The location in MAS of a group called "DATAAREA".
TARGET:	EI(SIN)	The address of the external subroutine "SIN".
TARGET:	EI(PAYSCHEDULE), 24	The 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE".

FRAME - Every 8086 memory reference is to a location contained within some FRAME; where the FRAME is designated by the content of some segment register. For the Linker to form a correct, usable memory reference, it must know what the TARGET is, and to which FRAME the reference is being made. Thus, every fixup specifies such a FRAME, in one of six ways. Some ways use data, X, which is in INDEX-or-FRAME-NUMBER, as above. Other ways require no data.

The six ways of specifying frames are:

(F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.

(F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (i.e., the canonic FRAME defined by the LSEG in the group that is eventually LOCATEd lowest in MAS).

(F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are three cases:

- (F2a) The symbol is defined relative to some LSEG, and there is no associated GROUP. The LSEGs canonic FRAME is specified.

- (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated GROUP. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF record that gives the symbol's definition.

- (F2c) Regardless of how the symbol is defined, there is an associated GROUP. The canonic FRAME of the GROUP is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record.)

(F3) X is a FRAME NUMBER (specifying the obvious FRAME).

(F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.

(F5) No X. The FRAME is determined by the TARGET. There are four cases:

- (F5a) The TARGET specified a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).

- (F5b) The TARGET specified a GROUP INDEX: in this case, the FRAME is determined as in (F1).

- (F5c) The TARGET specified an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2).

- (F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case the FRAME is determined as in (F3).

> **Note**
>
> The Microsoft Linker does not support frame methods F2b, F3, and F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the FRAME specified by a self-relative reference is usually the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference is the canonic FRAME of the LSEG containing the TARGET.

6.9 SELF-RELATIVE FIXUPS

A self-relative fixup operates as follows: A memory address is implicitly defined by LOCATION; namely the address of the byte following LOCATION (because at the time of a self-relative reference, the 8086 IP (Instruction Pointer) is pointing to the byte following the reference).

For 8086 self-relative references, if either LOCATION or TARGET are outside the specified FRAME, the Linker gives a warning. Otherwise, there is a unique 16-bit displacement which, when added to the address implicitly defined by LOCATION, will yield the relative position of TARGET in the FRAME.

If the LOCATION is an OFFSET, the displacement is added to LOCATION modulo 65536; no errors are reported.

If the LOCATION is a LOBYTE, the displacement must be within the range {-128:127}, otherwise the Linker will give a warning. The displacement is added to LOCATION modulo 256.

If the LOCATION is a BASE, POINTER, or HIBYTE, it is unclear what the translator had in mind, and the action taken by the Linker is undefined.

6.10 SEGMENT-RELATIVE FIXUPS

A segment-relative fixup operates in the following way: a non-negative 16-bit number, FBVAL, is defined as the FRAME NUMBER of the FRAME specified by the fixup, and a signed 20-bit number, FOVAL, is defined as the distance from the base of the FRAME to the TARGET. If this signed 20-bit number is less than 0 or greater than 65535, the Linker reports an error. Otherwise, FBVAL and FOVAL are used to fixup LOCATION in the following fashion:

1. If LOCATION is a POINTER, then FBVAL is added (modulo 65536) to the high-order word of POINTER, and FOVAL is added (modulo 65536) to the low-order word of POINTER.

2. If LOCATION is a BASE, then FBVAL is added (modulo 65536) to the BASE; FOVAL is ignored.

3. If LOCATION is an OFFSET, then FOVAL is added (modulo 65536) to the OFFSET; FBVAL is ignored.

4. If LOCATION is a HIBYTE, then (FOVAL/256) is added (modulo 256) to the HIBYTE; FBVAL is ignored. (The indicated division is "integer division", i.e., the remainder is discarded.)

5. If LOCATION is a LOBYTE, then (FOVAL modulo 256) is added (modulo 256) to the LOBYTE; FBVAL is ignored.

6.11 RECORD ORDER

A object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

> **Note**
>
> The syntactic description language used below is defined in WIRTH: CACM, November 1977, vol.£20, no.£11, pp.£822-823. The character strings represented by capital letters above are not literals but are identifiers that are further defined in the section describing the record formats.

```
object file   = tmodule

tmodule       = THEADR seg-grp   {component}   modtail

seg_grp       = {LNAMES} {SEGDEF} {TYPDEF | EXTDEF | GRPDEF}

component     = data | debug_record

data          = content_def | thread_def |
                     TYPDEF | PUBDEF | EXTDEF

debug_record = LINNUM

content_def   = data_record {FIXUPP}

thread_def    = FIXUPP (containing only thread fields)

data_record   = LIDATA | LEDATA

modtail       = MODEND
```

The following rules apply:

1. A FIXUPP record always refers to the previous DATA record.
2. All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.
3. COMENT records may appear anywhere in a file, except as the first or last record in a file or module, or within a contentdef.

6.12 INTRODUCTION TO THE RECORD FORMATS

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

SAMPLE RECORD FORMAT
(SAMREC)

```
----------------------///---------||||-----------
|     |          |            |            |     |
| REC | RECORD   |    NAME    |   NUMBER   | CHK |
| TYP | LENGTH   |            |            | SUM |
| xxH |          |            |            |     |
|     |          |            |            |     |
----------------------///---------||||-----------
                 |            |
                 +----rpt-----+
```

TITLE and OFFICIAL ABBREVIATION

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

The BOXES

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide boxes with three slashes in the top and bottom represent a variable number of bytes, one or more, depending upon content. The wide boxes with four vertical bars in the top and bottom represent 4-byte fields.

RECTYP

The first byte in each record contains a value between 0 and 255, indicating which record type the record is.

RECORD LENGTH

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

NAME

Any field that indicates a "NAME" has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

NUMBER

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

REPEATED OR CONDITIONAL FIELDS

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the "repeated" or "rpt" brackets below the boxes.

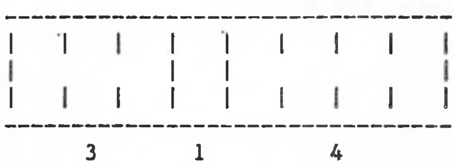
Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

CHKSUM

The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

BIT FIELDS

Descriptions of contents of fields will sometimes be at the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.



T-MODULE HEADER RECORD (THEADR)

```
------------------------///------------
|      |           |            |     |
| REC  |  RECORD   |     T      | CHK |
| TYP  |  LENGTH   |   MODULE   | SUM |
| 80H  |           |   NAME     |     |
|      |           |            |     |
------------------------///------------
```

Every module output from a translator must have a T-MODULE HEADER RECORD.

T-MODULE NAME

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD (LNAMES)

```
------------------------///------------
|      |           |            |     |
| REC  |  RECORD   |    NAME    | CHK |
| TYP  |  LENGTH   |            | SUM |
| 96H  |           |            |     |
|      |           |            |     |
------------------------///------------
                   |            |
                   +----rpt-----+
```

This Record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes and/or Groups.

The ordering of LNAMES records within a module, together with the ordering of names within each LNAMES Record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "Name Indices" in the Segment Name Index, Class Name Index and Group Name Index fields of the SEGDEF and GRPDEF Records.

NAME

This repeatable field provides a name, which may have zero length.

SEGMENT DEFINITION RECORD (SEGDEF)

```
------------------///------------------///-----///---///------
|   |          |         |         |          |     |     |   |
|REC| RECORD   | SEGMENT | SEGMENT | SEGMENT  |CLASS|OVER |CHK|
|TYP| LENGTH   |  ATTR   | LENGTH  |  NAME    |NAME |LAY  |SUM|
|98H|          |         |         |  INDEX   |INDEX|NAME |   |
|   |          |         |         |          |     |INDEX|   |
------------------///------------------///-----///---///------
```

SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEGs, are defined implicitly by the sequence in which SEGDEF Records appear in the object file.

SEG ATTR

The SEG ATTR field provides information on various attributes of a segment, and has the following format:

```
 ------------------------
|     |        |        |
| ACB | FRAME  |  OFF   |
|  P  | NUMBER |  SET   |
|     |        |        |
|     |        |        |
 -------------r----------
      |                 |
      +---conditional---+
```

The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:

```
 ---------------------------------
|   |   |   |   |   |   |   |   |
:       A   |       C       | B | P |
|   |   |   |   |   |   |   |   |
 ---------------------------------
```

"A" (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

A=0 SEGDEF describes an absolute LSEG.
A=1 SEGDEF describes a relocatable, byte-aligned LSEG.
A=2 SEGDEF describes a relocatable, word-aligned LSEG.
A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
A=4 SEGGDEF describes a relocatable, page-aligned LSEG.

If A=0, the FRAME NUMBER and OFFSET fields will be present. Using MS-LINK, absolute segments may be used for addressing purposes only; for example, defining the starting address of a ROM and defining symbolic names for addresses within the ROM. MS-LINK will ignore any data specified as belonging to an absolute LSEG.

"C" (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGs are combined: Let X,Y be LSEGs, and let Z be the LSEG resulting from the combination of X,Y. Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY. Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y. Let LZ denote the length of the (combined) LSEG Z; let dx (0=DXLX) be the offset in X of a byte, and let dy similarly be the offset in Y of a byte. The following table gives the length LZ of the combined LSEG Z, and the offsets dx′ and dy′ in Z for the bytes corresponding to dx in X and dy in Y. Intel defines additionally alignment types 5 and 6 and also processes code and data placed in segment with align-type.

Table 6.2. Combination Attribute Example

C	LZ	dx′	dy′	
2	LX+LY+G	dx	dy+LX+G	"Public"
5	LX+LY+G	dx	dy+LX+G	"Stack"
6	MXY	dx	dy	"Common"

Table 6.2 has no lines for C=0, C=1, C=3, C=4 and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2. C1, C4, and C7 all have different meanings according to the Intel standard.

"B" (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly 64K (65536). In this case the SEGMENT LENGTH field must contain zero.

The "P" field must always be zero. The "P" field is the "Page resident" field in Intel-Land.

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of the FRAME NUMBER should be done.

SEGMENT LENGTH

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero; if so, MS-LINK will not delete the segment from the module. The SEGMENT LENGTH field is only big enough to hold numbers from 0 to 64K-1 inclusive. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to give the segment a length of 64K.

SEGMENT NAME INDEX

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, TAXDATA, MODULENAME_CODE, STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES Record(s).

CLASS NAME INDEX

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of Class Names is to allow the programmer to define a "handle" used in the ordering of the LSEGs in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the LNAMES Record(s).

OVERLAY NAME INDEX

> **Note**
>
> This is ignored in MS-LINK versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The Overlay Name is a name the translator and/or MS-LINK, at the programmer's request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the LNAMES Record(s).

> **Note**
>
> The "Complete Name" of a segment is a 3-component entity comprising a Segment Name, a Class Name and an Overlay Name. (The latter two components may be null.)

GROUP DEFINITION RECORD
(GRPDEF)

```
-----------------------///---------///------------
|     |           |            |            |     |
| REC |  RECORD   |   GROUP    |   GROUP    | CHK |
| TYP |  LENGTH   |   NAME     | COMPONENT  | SUM |
| 9AH |           |   INDEX    | DESCRIPTOR |     |
|     |           |            |            |     |
--------------------///----------///-------------
                               |            |
                               +--repeated--+
```

GROUP NAME INDEX

The Group Name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some FRAME which "covers" every LSEG of the group.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the LNAMES Record(s).

GROUP COMPONENT DESCRIPTOR

Each GROUP COMPONENT DESCRIPTOR has the following format:

```
-----------///-----
|     |           |
| SI  |  SEGMENT  |
|     |   INDEX   |
|(FFH)|           |
|     |           |
-------------------
```

The first byte of the DESCRIPTOR contains OFFH; the DESCRIPTOR contains one field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

Intel defines 4 other group descriptor types, each with its own meaning. They are OFEH, OFDH, OfBH, and OfAH. The Microsoft Linker will treat all of these values the same as OFFH (i.e., it always expects OFFH followed by a segment index, and it does not, in fact, check to see if the value is actually OFF).

TYPE DEFINITION RECORD
(TYPDEF)

```
-----------------------///---------///-----------
|     |          |           |            |     |
| REC |  RECORD  |   NAME    |   EIGHT    | CHK |
| TYP |  LENGTH  | (USUALLY  |    LEAF    | SUM |
| 8EH |          |   NULL)   | DESCRIPTOR |     |
|     |          |           |            |     |
---------------------///----------///------------
                             |            |
                             +--repeated--+
```

The Microsoft Linker uses TYPDEF records only for communal variable allocation. This is not Intel's intended purpose. See Section 6.14, "Microsoft Type Representations for Communal Variables."

As many "EIGHT LEAF DESCRIPTOR" fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

NAME

Use of this field is reserved. Translators should place a single byte containing 0 in it (which is the representation of a name of length zero).

EIGHT LEAF DESCRIPTOR

This field can describe up to eight Leaves.

```
-----------///------
|     |            |
|  E  |    LEAF    |
|  N  | DESCRIPTOR |
|     |            |
|     |            |
-----------///------
      |            |
      +----rpt-----+
```

The EN field is a byte: the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=1).

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:

```
 -------
|       |
|   0   |
|  to   |
|  128  |
|       |
 -------

 -------------------
|       |           |
|       |     0     |
|  129  |    to     |
|       |   64K-1   |
|       |           |
 -------------------


 -------------------------
|       |                 |
|       |        0        |
|  132  |       to        |
|       |     16M-1       |
|       |                 |
 -------------------------

 -------------------------
|       |                 |
|       |      -2G-1      |
|  136  |       to        |
|       |      2G-1       |
|       |                 |
 -------------------------
```

The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

The second format, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following four bytes, sign extended if necessary.

PUBLIC NAMES DEFINITION RECORD (PUBDEF)

```
---------------------///------///--------------///---------
|    |          |          |         |       |       |     |
| REC|  RECORD  |  PUBLIC  | PUBLIC  |PUBLIC | TYPE  | CHK |
| TYP|  LENGTH  |   BASE   |  NAME   |OFFSET | INDEX | SUM |
| 90H|          |          |         |       |       |     |
|    |          |          |         |       |       |     |
---------------------///-----///---------------///---------
                           |                         |
                           +---------repeated--------+
```

This record provides a list of one or more PUBLIC NAMEs; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

PUBLIC BASE

The PUBLIC BASE has the following format:

```
-----///---------///-----------------
|          |          |          |
|  GROUP   | SEGMENT  |  FRAME   |
|  INDEX   |  INDEX   |  NUMBER  |
|          |          |          |
|          |          |          |
-----///---------///-----------------
                      |          |
                      +conditional+
```

The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX associates a group with the public symbol, and is used as described in Section 6.8, "Conceptual Framework for Fixups," case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero SEGMENT INDEX selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the "frame of reference" for references to all public symbols defined in this record; that is, MS-LINK will perform the following actions:

1. Any fixup of the form:

   TARGET: EI(P)

   FRAME: TARGET

   (where "P" is a public symbol in this PUBDEF record) will be converted by MS-LINK to a fixup of the form:

   TARGET: SI(L),d

   FRAME: GI(G)

   where "SI(L)" and "d" are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The "normal" action would have the frame specifier in the new fixup be the same as in the old fixup: FRAME: TARGET.)

2. When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optionally) FRAME NUMBER fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol´s value, an error occurs.

A GROUP INDEX of zero selects no group. MS-LINK will not alter the FRAME specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field.

PUBLIC NAME

The PUBLIC NAME field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

PUBLIC OFFSET

The PUBLIC OFFSET field is a 16-bit value, which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol. This field is ignored by the Linker.

EXTERNAL NAMES DEFINITION RECORD
(EXTDEF)

```
-----------------------///---------///-----------
|     |           |           |           |     |
| REC |  RECORD   | EXTERNAL  |   TYPE    | CHK |
| TYP |  LENGTH   |   NAME    |   INDEX   | SUM |
| 8CH |           |           |           |     |
|     |           |           |           |     |
-----------------------///---------///-----------
                  |                       |
                  +-------repeated--------+
```

This record provides a list of external names, and for each name, the type of object it represents. MS-LINK will assign to each External Name the value provided by an identical Public Name (if such a name is found).

EXTERNAL NAME

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is referenced within some FIXUPP Record in the module.
The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered 1, 2, 3, 4, ... These numbers are used as "External Indices" in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records to refer to a particular External Name.

> **Note**
>
> 8086 External Names are numbered positively: 1,2,3,... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning.

External indices may not reference forward. For example, an external definition record defining the kth object must precede any record referring to that object with index k.

TYPE INDEX

This field identifies a single preceding TYPDEF (Type Definition) record containing a descriptor for the type of object named by the External Symbol.

The TYPE INDEX is used only in communal variable allocation by the Microsoft Linker.

<u>LINE NUMBERS RECORD</u>
(LINNUM)

```
-----------------------///------------------------------------
|      |          |           |           |           |       |
| REC  |  RECORD  |   LINE    |   LINE    |   LINE    | CHK   |
| TYP  |  LENGTH  |  NUMBER   |  NUMBER   |  NUMBER   | SUM   |
| 94H  |          |   BASE    |           |  OFFSET   |       |
|      |          |           |           |           |       |
-----------------------///------------------------------------
                              |                       |
                              +--------repeated-------+
```

This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

LINE NUMBER BASE

The LINE NUMBER BASE has the following format:

```
-----///---------///-----
|          |           |
|   GROUP  |  SEGMENT  |
|   INDEX  |   INDEX   |
| (ignored)|           |
|          |           |
-----///---------///-----
```

The SEGMENT INDEX determines the location of the first byte of code corresponding to some source line number.

LINE NUMBER

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

LINE NUMBER OFFSET

The LINE NUMBER OFFSET field is a 16-bit value, which is the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0).

LOGICAL ENUMERATED DATA RECORD
(LEDATA)

```
-----------------------///-----------------------------
|     |            |           |           |     |     |
| REC | RECORD     |  SEGMENT  | ENUMERATED|     | CHK |
| TYP | LENGTH     |  INDEX    |   DATA    | DAT | SUM |
| AOH |            |     .     |  OFFSET   |     |     |
|     |            |           |           |     |     |
-----------------------///-----------------------------
                                           |     |
                                           +-rpt-+
```

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD.

ENUMERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

DAT

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)

```
---------------------///----------------------///----------
|     |          |          |           |          |     |
| REC |  RECORD  | SEGMENT  | ITERATED  | ITERATED | CHK |
| TYP |  LENGTH  |  INDEX   |   DATA    |   DATA   | SUM |
| A2H |          |          |  OFFSET   |  BLOCK   |     |
|     |          |          |           |          |     |
---------------------///----------------------///----------
                                        |          |
                                        +-repeated--+
```

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA RECORD.

ITERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory.

ITERATED DATA BLOCK

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:

```
-----------------------------///-----
|          |             |           |
| REPEAT   |   BLOCK     |           |
|  COUNT   |   COUNT     |  CONTENT  |
|          |             |           |
|          |             |           |
-----------------------------///-----
```

> **Note**
>
> The Linker cannot handle LIDATA records whose ITERATED DATA BLOCK is larger than 512 bytes.

REPEAT COUNT

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

BLOCK COUNT

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero, then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero, then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKs.

CONTENT

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

If BLOCK COUNT is zero, then this field is a 1-byte count followed by the indicated number of data bytes.

If BLOCK COUNT is non-zero, then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

> **Note**
>
> From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

FIXUP RECORD
(FIXUPP)

```
 ------------------------///------------
 |     |            |          |     |
 | REC |   RECORD   |  THREAD  | CHK |
 | TYP |   LENGTH   |    or    | SUM |
 | 9CH |            |   FIXUP  |     |
 |     |            |          |     |
 ------------------------///------------
                    |          |
                    +----rpt---+
```

This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers. Each fixup is specified by a FIXUP field that specifies four data: a location, a mode, a target and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specified by reference to a preceding THREAD field.

A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided; four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and Thread Number (0 - 3) appears (in the same or another FIXUPP record).

THREAD

THREAD is a field with the following format:

```
-----------///-----
|     |           |
| TRD |   INDEX   |
|     |           |
|     |           |
-----------///-----
      |           |
      +conditional+
```

The TRD DAT (ThReaD DATa) subfield is a byte with this internal structure:

```
 ---------------------------------
 |   |   |   |   |   |   |   |   |
 | 0 | D | Z |   METHOD  | THRED |
 |   |   |   |   |   |   |   |   |
 ---------------------------------
```

The "Z" is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The "D" subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

METHOD is a 3-bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If D=0, then METHOD = (0, 1, 2, 3, 4, 5, 6, 7) mod 4, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of Index or Frame Number is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by MS-LINK.

If D=1, then METHOD = 0, 1, 2, 4, 5, corresponding to methods F0, ..., of specifying a frame. Here, METHOD indicates what kind (if any) of Index is required to specify the frame. Note that methods 3 and 5d are not supported by MS-LINK.

THRED is a number between 0 and 3, and associates a Thread Number to the frame or target defined by the THREAD field.

INDEX contains a Segment Index, Group Index, or External Index depending on the specification in the METHOD subfield. This subfield will not be present if F4 or F5 are specified by METHOD.

FIXUP

FIXUP is a field with the following format:

```
----------------------///---------///---------///-----
|          |     |           |           |           |
|  LOCAT   | FIX |   FRAME   |  TARGET   |  TARGET   |
|          | DAT |   DATUM   |  DATUM    |   DIS-    |
|          |     |           |           | PLACEMENT |
|          |     |           |           |           |
----------------------///---------///---------///-----
                 |           |           |           |
                 +conditional+conditional+conditional+
```

LOCAT is a byte pair with the following format:

```
----------------------------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | M | S |      LOC      |  D A T A   R E C O R D   O F F S E T  |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
----------------------------------------------------------------
|                               |                               |
+-----------lo byte-------------+------------hi byte------------+
```

M is a 1-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

> **Note**
>
> Self-relative fixups may not be applied to LIDATA records.

"S" is a 1-bit subfield that specifies that the length of the TARGET DISPLACEMENT subfield. If it is present in this FIXUP field (see below), it will be either two bytes (containing a 16-bit non-negative number, S=0) or three bytes (containing a signed 24-bit number in 2's complement form, S=1).

> **Note**
>
> 3-byte subfields are a possible future extension, and are not currently supported. Thus, S=0 is currently mandatory.

LOC is a 3-bit subfield indicating that the byte(s) in the preceding DATA Record to be fixed up are a "lobyte" (LOC=0), an "offset" (LOC=1), a "base" (LOC=2), a "pointer" (LOC=3), or a "hibyte" (LOC=4). Other values in LOC are invalid.

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDs.

> **Note**
>
> If the preceding DATA record is an LIDATA record, it is possible for the value of DATA RECORD OFFSET to designate a "location" within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is an error. MS-LINK's action on such a malformed record is undefined.

FIX DAT is a byte with the following format:

```
---------------------------------
|   |       |       |   |   |       |
| F |     FRAME     | T | P | TARGT |
|   |       |       |   |   |       |
---------------------------------
     See Note 1         See Note 2
```

Note 1: Frame method 2b, F3, and F5d are not supported.

Note 2: Target method T3 and T7 are not supported.

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero, FRAME is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a FRAME. If F=1, then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

"T" is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

"P" is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGT is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7). When the target is specified by a thread (T=1), then TARGT specifies a thread number (0-3).

FRAME DATUM is the "referent" portion of a frame specification, and is a Segment Index, a Group Index, an External Index. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

TARGET DATUM is the "referent" portion of a target specification, and is a Segment Index, a Group Index, an External Index or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

TARGET DISPLACEMENT is the 2-byte displacement required by "primary" ways of specifying TARGETs. This 2-byte subfield is present if P=0.

> Note
>
> All these methods are described in Section 6.8, "Conceptual Framework for Fixups."

MODULE END RECORD
(MODEND)

```
-----------------------------------///------------
|      |              |       |            |      |
| REC  |   RECORD     | MOD   |   START    | CHK  |
| TYP  |   LENGTH     | TYP   |   ADDRS    | SUM  |
| 8AH  |              |       |            |      |
|      |              |       |            |      |
-----------------------------------///------------
                              |            |
                              +conditional+
```

This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

MOD TYP

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:

```
----------------------------------
|    |    |    |    |    |    |    |    |
| MATTR   | Z  | Z  | Z  | Z  | Z  | L  |
|    |    |    |    |    |    |    |    |
----------------------------------
```

MATTR is a 2-bit subfield that specifies the following module attributes:

MATTR	MODULE ATTRIBUTE
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

"L" indicates whether the START ADDRS field is interpreted as a logical address that requires fixing up by MS-LINK. (L=1). Note: with MS-LINK, L must always equal 1.

"Z" indicates that this bit has not currently been assigned a function. These bits are required to be zero.

Physical start addresses (L=0) are not supported.

The START ADDRS field (present only if MATTR is 1 or 3) has the following format:

START ADDRS

```
-----------///---------///-----------------
|     |           |           |           |
| END |   FRAME   |  TARGET   |  TARGET   |
| DAT |   DATUM   |   DATUM   |   DIS-    |
|     |           |           | PLACEMENT |
|     |           |           |           |
-----------///---------///-----------------
      |           |           |           |
      +conditional+conditional+conditional+
```

The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only "primary" fixups are allowed. Frame method F4 is not allowed.

COMMENT RECORD
(COMENT)

REC TYP 88H	RECORD LENGTH	COMMENT TYPE	COMMENT ///	CHK SUM

This record allows translators to include comments in object text.

COMMENT TYPE

This field indicates the type of comment carried by this record. This allows comments to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

NP	NL	Z	Z	Z	Z	Z	Z	COMMENT CLASS

The NP (NOPURGE) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting COMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMENT records.

The COMMENT CLASS field is defined as follows:

0	Language translator comment.
1	Intel copyright comment. The NP bit must be set.
2-155	Reserved for Intel use. (See note 1 below.)
156-255	Reserved for users. Intel products will apply no semantics to these values. (See Note 2 below.)

COMMENT

This field provides the commentary information.

Notes:

1. Class value 129 is used to specify a library to add to the Linker's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length. The "NODEFAULTLIBRARYSEARCH" switch causes the linker to ignore all comment records whose class value is 129.

2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.

6.13 NUMERIC LIST OF RECORD TYPES

*6E RHEADR
*70 REGINT
*72 REDATA
*74 RIDATA
*76 OVLDEF
*78 ENDREC
*7A BLKDEF
*7C BLKEND
*7E DEBSYM
80 THEADR
*82 LHEADR
*84 PEDATA
*86 PIDATA
88 COMENT
8A MODEND
8C EXTDEF
8E TYPDEF
90 PUBDEF
*92 LOCSYM
94 LINNUM
96 LNAMES
98 SEGDEF
9A GRPDEF
9C FIXUPP
*9E (none)
A0 LEDATA
A2 LIDATA
*A4 LIBHED
*A6 LIBNAM
*A8 LIBLOC
*AA LIBDIC

> **Note**
>
> Record types preceded by an asterisk (*) are not supported by the Microsoft Linker. They will be ignored if they are found in an object module.

6.14 MICROSOFT TYPE REPRESENTATIONS FOR COMMUNAL VARIABLES

This section defines the Microsoft standard for communal variable allocation on the 8086 and 80286.

A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char    foo[4];         /* In file a.c */
char    foo[1];         /* In file b.c */
char    foo[1024];      /* In file c.c */
```

If the objects produced from a.c, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array "foo".

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

The TYPDEF for a communal variable has the following format:

```
-------------------------///-----------
| REC | RECORD |   |     EIGHT    | CHK |
| TYP | LENGTH | 0 |     LEAF     | SUM |
| 8EH |        |   | DESCRIPTOR   |     |
-------------------------///-----------
```

The EIGHT LEAF DESCRIPTOR field has the following format:

```
---------///------
| E |    LEAF    |
| N | DESCRIPTOR |
---------///------
```

The EN field specifies whether the next 8 leaves in the LEAF DESCRIPTOR field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The LEAF DESCRIPTOR field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:

```
------------------///-----///-----
| NEAR | VAR |  LENGTH  |  VAR    |
| 62H  | TYP |    IN    | SUBTYP  |
|      |     |   BITS   |         |
------------------///------///----
                        |         |
                        +---------+
                        (optional)
```

The VARiable TYPe field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by the Linker. The format for communal variables not in the default data segment (far variables) is as follows:

```
-----------------///-------///----
| FAR | VAR |  NUMBER   | ELEMENT |
| 61H | TYP |    OF     |  TYPE   |
|     | 77H | ELEMENTS  |  INDEX  |
-----------------///-------///----
```

The VARiable TYPe field must be ARRAY (77H). The length field specifies the NUMBER OF ELEMENTS, and the ELEMENT TYPE INDEX is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the LENGTH IN BITS or NUMBER OF ELEMENTS fields is the same as the format for the LEAF DESCRIPTOR field, described in the TYPDEF record format section of this manual.

Link time semantics:

All EXTDEFs referencing a TYPDEF of one of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two communal variable definitions are said to match if the names given in the definitions match. If two matching definitions disagree about whether a communal variable is near or far, the linker will assume the variable is near.

If the variable is near, then its size is the largest of the sizes specified for it. If the variable is far, then the Linker issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

"Huge" communal variables:

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

If the linker finds matching huge and near communal variable definitions, it issues a warning message, since it is impossible for a near variable to be larger than 64K bytes.

CHAPTER 7

PROGRAMMING HINTS

CHAPTER 7

PROGRAMMING HINTS

7.1 INTRODUCTION

This chapter describes recommended MS-DOS 3.1 programming procedures. By using these programming hints, you can ensure compatibility with future versions of MS-DOS.

The hints are organized into the following categories:

- Interrupts
- System Calls
- Device Management
- Memory Management
- Process Management
- File and Directory Management
- Miscellaneous

7.2 INTERRUPTS

Never explicitly issue Interrupt 22H (Terminate Process Exit Address).

> This should only be done by the DOS. To change the terminate address, use Function 35H (Get Interrupt Vector) to get the current address and save it, then use Function 25H (Set Interrupt Vector) to change the Interrupt 22H entry in the vector table to point to the new terminate address.

Use Interrupt 24H (Critical Error Handler Address) with care.

> The Interrupt 24H handler must preserve the ES register.
>
> Only system calls 01H-0CH can be made by an Interrupt 24H handler. Making any other calls will destroy the MS-DOS stack and prevent successful use of the Retry or Ignore options.
>
> The registers SS, SP, DS, BX, CX, and DX must be preserved when using the Retry or Ignore options.

When an Interrupt 24H (Critical Error Handler Address) is received, always IRET back to MS-DOS with one of the standard responses.

> Programs that do not IRET from Interrupt 24H leave the system in an unpredictable state until a function call other than 01H-0CH is made. The Ignore option may leave data in internal system buffers that is incorrect or invalid.

Avoid trapping Interrupt 23H (Ctrl-Break Handler Address) and Interrupt 24H (Critical Error Handler Address). Don't rely on trapping errors via Interrupt 24H as part of a copy protection scheme.

> These might not be included in future releases of the operating system.

Interrupt 23H (Ctrl-Break Handler Address) must never be issued by a user program.

> Interrupt 23H must be issued only by MS-DOS.

Save any registers your program uses before issuing Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write).

These interrupts destroy all registers except for the segment registers.

Avoid writing or reading an interrupt vector directly to or from memory.

Use Functions 25H and 35H (Set Interrupt Vector and Get Interrupt Vector) to set and get values in the interrupt table.

7.3 SYSTEM CALLS

Use new system calls.

Avoid using system calls that have been superseded compatibility with pre-2.0 versions of MS-DOS. See Section 1.8, "Old System Calls," of this manual for a list of these new calls.

Avoid using system calls 01H-0CH and 26H (Create New PSP).

Use the new "tools" approach for reading and writing on standard input and output. Use Function 4B00H (Load and Execute Program) instead of 26H to execute a child process.

Use file-sharing calls if more than one process is in effect.

See "File Sharing," in Section 1.5.2, "File-Related Function Requests" in Chapter 1 for more information.

Use networking calls where appropriate.

Some forms of IOCTL can only be used with Microsoft Networks. See Section 1.6, "Microsoft Networks," in this manual for a list of these calls.

When selecting a disk with Function 0EH (Select Disk), treat the value returned in AL with care.

The value in AL specifies the maximum number of logical drives; it does not specify which drives are valid.

7.4 DEVICE MANAGEMENT

Use installable device drivers.

MS-DOS provides a modular device driver structure for the BIOS, allowing you to configure and install device drivers at boot time. Block device drivers transmit a block of data at a time, while character device drivers transmit a byte of data at a time.

Examples of both types of device drivers are given in Chapter 2, "MS-DOS Device Drivers."

Use buffered I/O.

The device drivers can handle streams of data up to 64K. When sending a large amount of output to the screen, you can send it with one system call. This will increase performance.

Programs that use direct console I/O via Function 06H and 07H (Direct Console I/O and Direct Console Input) and that want to read Ctrl-Break as data should ensure that Ctrl-Break checking is off.

> The program should ensure that Ctrl-Break checking is off by using Function 33H (Ctrl-Break Check).

Be compatible with international support.

> To provide support for international character sets, MS-DOS recognizes all possible byte values as significant characters in filenames and data streams. Pre-2.x versions ignored the high bit in the MS-DOS filename.

7.5 MEMORY MANAGEMENT

Use memory management.

> MS-DOS keeps track of allocated memory by writing a memory control block at the beginning of each area of memory. Programs should use Functions 48H (Allocate Memory), 49H (Free Allocated Memory), and 4AH (Set Block) to release unneeded memory.
>
> This will allow for future compatibility.
>
> See Section 1.3, "Memory Management," for more information.

Only use allocated memory.

> Don't directly access memory that was not provided as a result of a system call. Do not use fixed addressing, use only relative references.
>
> A program that uses memory that has not been allocated to it may destroy other memory control blocks or cause other applications to fail.

7.6 PROCESS MANAGEMENT

Use the EXEC Function Call to load and execute programs.

> The EXEC Function (4B00H) is the preferred way to load programs and program overlays. Using the EXEC call instead of hard-coding information about how to load an .EXE file (or always assuming that your file is a .COM file) will isolate your program from changes in future releases of MS-DOS and .EXE file formats.

Use Function 31H (Keep Process), instead of Interrupt 27H (Terminate But Stay Resident). Function 31H allows programs to terminate and stay resident that are greater than 64K.

Programs should terminate using End Process (4CH).

Programs that terminate by

- a long jump to offset 0 in the PSP,
- issuing an Interrupt 20H with CS:0 pointing at the PSP,
- issuing an Interrupt 21H with AH=0, CS:0 pointing at the PSP, or
- a long call to location 50H in the PSP with AH=0

must ensure that the CS register contains the segment address of the PSP.

7.7 FILE AND DIRECTORY MANAGEMENT

Use the MS-DOS file management system.

> Using the MS-DOS file system will ensure program compatibility with future MS-DOS versions through compatible disk formats and consistent internal storage. This will ensure compatibility with future MS-DOS versions.

Use file handles instead of FCBs.

> A handle is a 16-bit number that is returned by MS-DOS when a file is opened or created using Functions 3CH, 3DH, 5AH, or 5BH (Create Handle, Open Handle, Create Temporary File, or Create New File). The MS-DOS file-related function requests that use handles are listed in Table 1.5 in Chapter 1, "System Calls."
>
> These calls should be used instead of the old file-related functions that use FCBs (file control blocks). This is because a file operation can simply pass its handle rather than having to maintain FCB information. If FCBs must be used, be sure the program closes them and does not move them around in memory.

Close all files that have changed in length before issuing an Interrupt 20H (Program Terminate), Function 00H (Terminate Program), Function 4CH (End Process), or Function 0DH (Reset Disk).

> If a changed file is not closed, its length will not be recorded correctly in the directory.

Close all files when they are no longer needed.

Closing unneeded files will optimize performance in a networking environment.

Only change disks if all files on the disk are closed.

Information in internal system buffers may be written incorrectly to a changed disk.

7.7.1 Locking Files

Programs should not rely on being denied access to a locked region.

Determine the status of the region by attempting to lock it, and examine the error code.

Programs should not close a file with a locked region or terminate with an open file that contains a locked region.

The result is undefined. Programs that might be terminated by an Interrupt 23H or Interrupt 24H (Ctrl-Break Handler Address or Critical Error Handler Address) should trap these interrupts and unlock any locked regions before exiting.

7.8 MISCELLANEOUS

Avoid timing dependencies.

Various machines use CPUs of different speeds. Also, programs that rely upon the speed of the clock for timing will not be dependable in a networking environment.

Use the documented interface to the operating system. If either the hardware or media change, the operating system will be able to use the features without modification.

Don´t use the OEM (Original Equipment Manufacturer) -provided ROM support.

Don´t directly address the video memory.

Don´t use undocumented function calls, interrupts, or features. These items may change or not continue to exist in future versions of MS-DOS. Use of these features would make your program highly non-portable.

Use the .EXE format rather than the .COM format.

.EXE files are relocatable and .COM files are direct memory images that load at a specific place and have no room for additional control information to be placed in them. .EXE files have headers that can be expanded for compatibility with future versions of MS-DOS.

Use the environment to pass information to applications.

The environment allows a parent process to pass information to a child process. COMMAND.COM is usually the parent process to every application, so default drive and path information can easily be passed to the application.

INDEX

INDEX

.EXE files 5-1

Absolute Disk Read (Interrupt 25H) 1-50
Absolute Disk Write (Interrupt 26H) 1-53
Allocate Memory (Function 48H) 1-208
Archive bit 3-5
ASCIZ string 1-219, 1-229
Assign list 1-16
Attribute byte 1-15
Attribute field 2-9
AUTOEXEC file 3-2
Auxiliary Input (Function 03H) 1-62
Auxiliary Output (Function 04H) 1-63

BASE 6-10
BIN format file 2-3
BIOS Parameter Block (BPB) 2-18, 2-22, 2-31
Bit 8 2-14
Bit 9 2-14
Block devices
  device drivers 2-26
  disk drives 2-4
  example 2-37
  installation 2-17
Boot sector 2-32
BPB pointer . . . . 2-17 to 2-18
Buffered Keyboard Input (Function 0AH) 1-71
BUILD BPB 2-22
Busy bit 2-14, 2-27, 2-29

Cancel Assign List Entry (Function 5FH, Code 04H) 1-272
Canonic Frame 6-4
Carry flag 1-25
Case-Mapping Call 1-150
Change Current Directory (Function 3BH) 1-158

Change Directory Entry (Function 56H) 1-235
Character device driver, example 2-56
Character devices 2-3
Check Keyboard Status (Function 0BH) 1-73
Class name, LSEG 6-5
CLOCK device 2-10, 2-35
Close File (Function 10H) 1-82
Close Handle (Function 3EH) 1-167
Cluster 3-3
Combination Attribute . . 6-26
COMENT 6-52
Command code field 2-13
Command processor 3-1
COMMAND.COM 3-1
COMMENT RECORD 6-52
Compatibility, ensuring . 7-1
Complete name, LSEG . . . 6-6
COMSPEC 4-4
CON device 2-5
CONFIG.SYS 2-1, 2-7
Control blocks 4-1
Control information . . . 5-1
Ctrl-Break Address (Interrupt 23H) 3-1
Ctrl-Break Check (Function 33H) 1-142
Ctrl-Break Handler Address (Interrupt 23H) 1-42
Create Directory (Function 39H) 1-154
Create File (Function 16H) 1-95
Create Handle (Function 3CH) 1-160
Create New File (Function 5BH) 1-251
Create New PSP (Function 26H) 1-116
Create Temporary File (Function 5AH) 1-248
Critical Error Handler Address (Interrupt 24H) 1-44, 3-1

Delete Directory Entry (Function 41H) 1-174
Delete File (Function 13H) 1-89
Device control 1-12
Device drivers
  block 2-3
  creating 2-4
  dumb 2-18
  example 2-37, 2-56

installable 2-1
installing 2-7
non-resident 2-1
preserving registers . . 2-37
resident 2-1
smart 2-18
Device handles 1-9
Device header 2-7
Device interrupt routine . 2-6
Device management, programming hints 7-4
Device strategy routine . 2-6
Device-related function requests 1-12
Direct Console I/O (Function 06H) 1-66
Direct Console Input (Function 07H) 1-68
Directory entry 1-14
Directory-related function requests 1-12
Disk allocation 3-2
Disk Directory 3-3
Disk formats
standard MS-DOS 3-10
Disk Transfer Address (DTA) 1-93, 1-229, 4-4
Dispatch table 2-36
Display Character (Function 02H) 1-61
Display String (Function 09H) 1-70
Done bit 2-14, 2-36
Dumb device driver 2-18
Duplicate File Handle (Function 45H) 1-200

EIGHT LEAF DESCRIPTOR . . 6-31
End address 2-17
End Process (Function 4CH) 1-225, 4-3
Error bit 2-37
Error codes 1-25
Error handling 1-30, 3-1
EXE device drivers 2-3
EXE files 5-1
EXE format file 2-3
EXE loader 2-3
EXTDEF 6-36
Extended error codes . . . 1-27
Extended FCB 1-20
EXTERNAL NAMES DEFINITION RECORD 6-36

FAT 2-23, 3-7
FAT ID byte 2-31
FCB 1-19
File Allocation Table . . 3-7
File and directory management, programming hints 7-7
File attributes 1-14
File Control Block
  definition 1-19
  extended 1-22
  fields 1-20
  format 1-20
  opened 1-19
  unopened 1-19
File locking, programming hints 7-8
File-related function requests 1-10
File-sharing function requests 1-11
Filename separators . . . 1-124
Filename terminators . . . 1-125
Find First File (Function 4EH) 1-229
Find Next File (Function 4FH) 1-232
FIXUP RECORD 6-43
FIXUPP 6-43
Fixups
  definition 6-9
  segment-relative 6-11, 6-17
  self-relative 6-11, 6-16
FLUSH 2-30
Flush Buffer, Read Keyboard (Function 0CH) 1-74
Force Duplicate File Handle (Function 46H) 1-203
Format 3-3
FRAME
  definition 6-3
  specifying 6-14
FRAME NUMBER 6-4
Free Allocated Memory (Function 49H) 1-211
Function requests (Interrupt 21H) 1-40
  alphabetic order 1-34
  calling 1-23
  definition 1-1
  device-related 1-12

directory-related . . . 1-13
file-related 1-10
file-sharing 1-11
Function 00H 1-58
Function 01H 1-60
Function 02H 1-61
Function 03H 1-62
Function 04H 1-63
Function 05H 1-64
Function 06H 1-66
Function 07H 1-68
Function 08H 1-69
Function 09H 1-70
Function 0AH 1-71
Function 0BH 1-73
Function 0CH 1-74
Function 0DH 1-76, 1-93
Function 0EH 1-77
Function 0FH 1-79
Function 10H 1-82
Function 11H 1-84
Function 12H 1-87
Function 13H 1-89
Function 14H 1-91
Function 15H 1-93
Function 16H 1-95
Function 17H 1-97
Function 19H 1-99
Function 1AH 1-100
Function 1BH 1-102
Function 1CH 1-104
Function 21H 1-106
Function 22H 1-108
Function 23H 1-111
Function 24H 1-113
Function 25H . . . . 1-41 to 1-44, 1-115
Function 26H 1-116
Function 27H 1-117
Function 28H 1-120
Function 29H 1-123
Function 2AH 1-127
Function 2BH 1-129

Function 2CH 1-131
Function 2DH 1-133
Function 2EH 1-135
Function 2FH 1-137
Function 30H 1-138
Function 31H 1-140
Function 33H 1-142
Function 35H . . . . 1-41 to 1-43, 1-144
Function 36H 1-146
Function 38H 1-148, 1-152
Function 39H 1-154
Function 3AH 1-156
Function 3BH 1-158
Function 3CH 1-160
Function 3DH 1-162
Function 3EH 1-167
Function 3FH 1-169
Function 40H 1-171
Function 41H 1-174
Function 42H 1-176
Function 43H 1-179
Function 44H, Code 08H . 1-192
Function 44H, Code 09H . 1-194
Function 44H, Code 0AH . 1-196
Function 44H, Code 0BH . 1-198
Function 44H, Codes 00H and 01H 1-182
Function 44H, Codes 02H and 03H 1-185
Function 44H, Codes 04H and 05H 1-187
Function 44H, Codes 06H and 07H 1-189
Function 45H 1-200
Function 46H 1-203
Function 47H 1-206
Function 48H 1-208
Function 49H 1-211
Function 4AH 1-214
Function 4BH, Code 00H . 1-217
Function 4BH, Code 03H . 1-222
Function 4CH 1-225
Function 4DH 1-227
Function 4EH 1-229
Function 4FH 1-232
Function 54H 1-234

Function 56H 1-235
Function 57H 1-238
Function 58H 1-241
Function 59H 1-244
Function 5AH 1-248
Function 5BH 1-251
Function 5CH, Code 00H . 1-254
Function 5CH, Code 01H . 1-258
Function 5EH, Code 00H . 1-261
Function 5EH, Code 02H . 1-263
Function 5FH, Code 02H . 1-265
Function 5FH, Code 03H . 1-268
Function 5FH, Code 04H . 1-272
Function 62H 1-274
handling errors 1-25
memory management . . . 1-4
network-related 1-16
numeric order 1-32
process management . . . 1-6
standard character I/O . 1-2
system-management . . . 1-17

Get Assign List Entry (Function 5FH, Code 02H) 1-265
Get Country Data (Function 38H) 1-148
Get Current Directory (Function 47H) 1-206
Get Current Disk (Function 19H) 1-99
Get Date (Function 2AH) . 1-127
Get Default Drive Data (Function 1BH) 1-102
Get Disk Free Space (Function 36H) 1-146
Get Disk Transfer Address (Function 2FH) 1-137
Get Drive Data (Function 1CH) 1-104
Get Extended Error (Function 59H) 1-244
Get File Size (Function 23H) 1-111
Get Interrupt Vector (Function 35H) 1-41 to 1-42, 1-144
Get Machine Name (Function 5EH, Code 00H) 1-261
Get MS-DOS Version Number (Function 30H) 1-138
Get PSP (Function 62H) . . 1-274
Get Return Code Child Process (Function 4DH) 1-227
Get Time (Function 2CH) . 1-131
Get Verify State (Function 54H) 1-234

Get/Set Allocation Strategy (Function 58H) 1-241
Get/Set Date/Time of File (Function 57H) 1-238
Get/Set File Attributes (Function 43H) 1-179
GROUP 6-4
Group Definition Record . 6-29
GRPDEF 6-29

Handles
  definition 1-9
  device 1-9
Handling errors 1-25
Header 5-1
HIBYTE 6-10
Hidden files 1-84, 1-87, 3-5
High-level language . . . 1-24

I/O Control for Devices (Function 44H) 2-10
Index fields 6-8
Indices 6-8
INIT 2-16, 2-18
INIT code 2-11
Installable device drivers 2-5
Instruction Pointer (IP) . 4-5
Internal stack 1-24, 2-37
Interrupt entry point . . 2-1 to 2-2, 2-35
Interrupt handlers . . . . 1-23, 1-41 to 1-43, 4-1
Interrupt routines 2-11
Interrupt-handling routine 1-115
Interrupts
  21H 1-40
  address of handlers . . 1-23
  alphabetic order 1-31
  definition 1-1
  Interrupt 20H 1-38, 1-58
  Interrupt 21H 1-24, 1-40
  Interrupt 22H 1-41
  Interrupt 23H . . . . . . 1-42, 1-60 to 1-61, 1-64, 1-69, 1-71
  Interrupt 24H 1-44
  Interrupt 25H 1-50
  Interrupt 26H 1-53
  Interrupt 27H 1-56

issuing 1-23
numeric order 1-31
programming hints . . . 7-1
vector table 1-23
IO.SYS file 3-5
IOCTL 1-12
IOCTL bit 2-11
IOCTL Block (Function 44H, Codes 4 and 5) 1-187
IOCTL Character (Function 44H, Codes 2 and 3) 1-185
IOCTL Data (Function 44H, Codes 0 and 1) 1-182
IOCTL Is Changeable (Function 44H, Code 08H) 1-192
IOCTL Is Redirected Block (Function 44H, Code 09H) 1-194
IOCTL Is Redirected Handle (Function 44H, Code 0AH) 1-196
IOCTL Retry (Function 44H, Code 0BH) 1-198
IOCTL Status (Function 44H, Codes 6 and 7) 1-189

Keep Process (Function 31H) 1-140

LEDATA 6-40
Length of Record Field . . 2-12
LIDATA 6-41
LINE NUMBERS RECORD . . . 6-38
LINNUM 6-38
List of Names Record . . . 6-23
LNAMES 6-23
Load and Execute Program (Function 4BH, Code 00H) 1-217
Load module 5-1, 5-3
Load Overlay (Function 4BH, Code 03H) 1-222
Loadsize 5-3
LOBYTE 6-10
LOCATION, types 6-10
Lock (Function 5CH, Code 00H) 1-254
LOGICAL ENUMERATED DATA RECORD 6-40
LOGICAL ITERATED DATA RECORD 6-41
Logical sector 3-7
Logical sector numbers . . 3-9
Logical Segment 6-3
LSEG 6-3

Make Assign List Entry (Function 5FH, Code 03H) 1-268
MAS 6-2
Maxalloc 5-3
MEDIA CHECK 2-19
Media descriptor byte . . 2-19, 2-31
Media, determining 2-34
Memory Address Space . . . 6-2
Memory control block . . . 1-4
Memory management function requests 1-4
Memory management, programming hints 7-5
Microsoft Networks 1-15, 7-4
Microsoft Networks Manager´s Guide 1-15
Microsoft Networks User´s Guide 1-15
Microsoft record types . . 6-55
Minalloc 5-3
MODE 6-11
MODEND 6-50
MODULE 6-3
MODULE END RECORD 6-50
Module header record . . . 6-6
Move File Pointer (Function 42H) 1-176
MS-DOS initialization . . 3-1
MS-DOS memory map 4-1
MS-LINK 5-1
MSDOS.SYS file 3-1, 3-5
Multiple media 2-18
Multitasking 2-1

Name field 2-11
NCR-DOS Manual 1-9
Network-related function requests 1-16
NON DESTRUCTIVE READ NO WAIT 2-27
NON FAT ID bit 2-10
NUL device 2-10
Numeric record types . . . 6-54

Object Module Formats . . 6-2
OFFSET 6-10
Old system calls 1-18
OMF 6-2
Open File (Function OFH) . 1-79

Open Handle (Function 3DH) 1-162
Opened FCB 1-19
Overlay Name, LSEG 6-5

PARAGRAPH NUMBER 6-4
Parameter block 1-218
Parse File Name (Function 29H) 1-123
Path command 4-4
Physical Segment 6-4
Pointer to Next Device field 2-9
Predefined device handles 1-9
Print Character (Function 05H) 1-64
Printer Setup (Function 5EH, Code 02H 1-263
Process management function requests 1-6
Process management, programming hints 7-6
Program End Process (Interrupt 20H) 1-38
Program segment 4-2
Program Segment Prefix . . 1-19, 1-24, 1-116, 1-218, 4-3, 5-3
Programming hints
  device management . . . 7-4
  file and directory management 7-7
  file locking 7-8
  interrupts 7-1
  memory management . . . 7-5
  miscellaneous 7-8
  process management . . . 7-6
  recommendations 7-1
  system calls 7-3
Prompt command 4-4
PSEG
  definition 6-4
  NUMBER 6-4
PUBDEF 6-33
PUBLIC NAMES DEFINITION RECORD 6-33

Random Block Read (Function 27H) 1-117
Random Block Write (Function 28H) 1-120
Random Read (Function 21H) 1-106
Random Write (Function 22H) 1-108
Read Handle (Function 3FH) 1-169
Read Keyboard (Function 08H) 1-69

Read Keyboard and Echo (Function 01H) 1-60
Read Only Memory 3-1
READ or WRITE 2-24
Record format, sample . . 6-20
Record formats 6-1
Record order 6-18
Record size 1-93
Record types
Registers, treatment of . 1-24
Relocatable memory images 6-1
Relocation information . . 5-1
Relocation item offset value 5-4
Relocation table 5-3
Remove Directory (Function 3AH) 1-156
Rename File (Function 17H) 1-97
request header 2-11
Request packet 2-2
Reset Disk (Function 0DH) 1-76, 1-93
Resident device drivers . 2-1
ROM 3-1
Root directory 3-4

Search for First Entry (Function 11H) 1-84
Search for Next Entry (Function 12H) 1-87
Sector count 2-36
SEGDEF 6-24
Segment addressing 6-7
Segment definition 6-6
Segment definition record 6-24
Segment Name, LSEG 6-5
Segment-relative fixups . 6-11, 6-17
Select Disk (Function 0EH) 1-77
Self-relative fixups . . . 6-11, 6-16
Sequential Read (Function 14H) 1-91
Sequential Write (Function 15H) 1-93
Set Block (Function 4AH) . 1-214, 4-5
Set command 4-4
Set Country Data (Function 38H) 1-152
Set Date (Function 2BH) . 1-129
Set Disk Transfer Address (Function 1AH) 1-100
Set Interrupt Vector (Function 25H) 1-41 to 1-44, 1-115

Set Relative Record (Function 24H) 1-113
Set Time (Function 2DH) . 1-133
Set/Reset Verify Flag (Function 2EH) 1-135
Smart device driver . . . 2-18
Standard character I/O function requests 1-2
Start sector 2-36
Start segment value . . . 5-4
static request header . . 2-2
STATUS 2-29
Status field 2-14
Strategy entry point . . . 2-1 to 2-2, 2-35
Strategy routines 2-11
Superseded system calls . 1-18
Symbol definition 6-8
SYSINIT 2-2
System calls
  definition 1-1
  programming hints . . . 7-3
  replacements for old . . 1-18
  superseded calls 1-2
  types of 1-1
System files 1-84, 1-87, 3-5
System prompt 3-2
System-management function requests 1-17

T-MODULE 6-3
T-module Header Record (THEADR) 6-23
TARGET 6-12
Terminate But Stay Resident (Interrupt 27H) 1-56
Terminate Process Exit Address (Interrupt 22H) 1-41
Terminate Program (Function 00H) 1-58
THEADR 6-23
Transfer address 2-36
TYPDEF 6-30
Type Definition Record . . 6-30
Type-ahead buffer 2-30

Unit code field 2-13
Unlock (Function 5CH, Code 01H) 1-258
Unopened FCB 1-19
User stack 4-1

Vector table 1-23
Volume ID 2-23
Volume label 3-5

Wildcard characters . . . 1-84, 1-87, 1-125
Write Handle (Function 40H) 1-171

SEE REVERSE SIDE OF THIS FORM FOR INSTRUCTIONS

DOCUMENTATION ORDER FORM

NCR Customer No.	Purchase Order No.	Purchase Order Date

SHIP TO:	Ship To No.:
Company Name	
Address	
City/State/Zip	
Attention	
In case we have questions regarding your order *Area Code ( ) Number — Ext.*	
☐ Change of Address	

BILL TO:	Bill To No.:
Company Name	
Address	
City/State/Zip	
Attention	
In case we have questions regarding your order *Area Code ( ) Number — Ext.*	
☐ Change of Address	

Ship VIA

International Only — Customs Declaration

Comments:

DOCUMENT NUMBER	QUANTITY	DESCRIPTION	UNIT PRICE	AMOUNT
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
			Subtotal	
			State/Local Taxes	
			Total	

Tax Exempt? ☐ No ☐ Yes — Tax Exempt No.

For your convenience you can:

1. Mail this order to:
   NCR Corporation
   Order Processing
   Publication Services
   Dayton, Ohio 45479
2. Submit this order to your local NCR office
3. Call our toll-free number:
   1-800-543-2010
   In Ohio:
   1-800-543-6691
   8:00 A.M.-4:30 P.M.
   EST - Weekdays

NCR

Signature — Date

ORDERING INFORMATION

HOW TO ORDER

For a fast, easy way to order and receive the documents you need, complete this Documentation Order Form as follows.

1. NCR CUSTOMER NO.

Enter your 8-digit NCR customer number.

> ***CUSTOMER NUMBERS ARE REQUIRED ON ALL ORDERS.***
>
> *If you are unsure of your customer number, please contact your local NCR office.*

2. PURCHASE ORDER NO.

Enter your purchase order number.

3. PURCHASE ORDER DATE

Enter the purchase order date.

4. SHIP TO

a. If you wish to have documents shipped to a location different than the location associated with your NCR customer number, enter the appropriate information in the space provided.

b. If the order is to be shipped to your NCR customer number location, no entry is required in this space.

5. BILL TO

No entry is required unless the BILL TO location is different from the SHIP TO location.

6. SHIP VIA

Enter your preferred method of delivery. All orders for items in stock will be processed and shipped within one week of receipt of order via UPS or mail for domestic shipment and air shipment for international orders. Rush orders will be processed and delivered within 48 hours.

7. CUSTOMS DECLARATION

(For International Orders)

If a customs declaration is required, enter the full text of the declaration in the space provided.

8. COMMENTS

Use this space for special comments about your order.

9. DOCUMENT NUMBER

Enter the number of the document you wish to order. NCR document numbers have a 2-character prefix, followed by 4 to 7 characters. (D1-0000-00).

10. QUANTITY

Enter the quantity desired. The following discounts apply when any document is purchased in quantities of 10 or more on one order.

- 10-49 10% discount
- 50-99 15% discount
- 100 or more 20% discount

11. DESCRIPTION

Enter the title of the document.

12. UNIT PRICE

Enter the unit price of the document. (Prices listed are those in effect at the time of printing and are subject to change without notice.) All prices are quoted in U.S. dollars.

13. AMOUNT

To calculate the line item amount, multiply the quantity by the unit price.

14. SUBTOTAL

Add all entries in the amount column and enter the sum in the SUBTOTAL space.

15. STATE/LOCAL TAXES

Calculate applicable state and local taxes by multiplying the SUBTOTAL amount by the percentage of tax. Enter that number in the STATE/LOCAL TAXES space. If tax exempt, your tax exempt number must be entered in the space provided to the left.

16. TOTAL

Enter the total amount of the order.

THREE CONVENIENT WAYS TO ORDER

Enter your order using one of the following convenient methods:

U.S. Customers

- Mail the Documentation Order Form to NCR Corporation, Order Processing—Publication Services, Dayton, Ohio 45479.
- Submit the order to your local NCR office.
- Call our toll-free number: 1-800-543-2010; in Ohio, 1-800-543-6691. Phone orders are taken from 8:00 am to 4:30 pm EST—weekdays.

International Customers

- Customers located outside of the United States should contact their local NCR office for ordering and pricing information.

PAYMENT METHODS

Payments are accepted by check, money order, or purchase order. Please do not send cash.

RETURNS

If you wish to return documents for credit, please contact our Order Processing department (1-800-543-2010; in Ohio, 1-800-543-6691) within 30 days of the shipment date. Full credit will be given if documents are returned because of an NCR error. A credit of up to 75% of the net amount may be issued for the return of unused, shrink-wrapped documents. Returns will NOT be accepted without prior approval from Publication Services.

OUT OF STOCK ITEMS

Every attempt will be made to ensure that your order is filled completely and accurately. If a document is temporarily out of stock, it will be placed on back order. When a partial shipment is made, your packing list will include a notification of this condition. Back orders will be automatically filled when the document is returned to our inventory.

Retain a copy of this order for your records. Thank you.

NCR

READER'S COMMENTS FORM

BOOK TITLE	BOOK NO.	PRINT DATE

To help us plan future editions of this document, please take a few minutes to answer the following questions. Explain in detail using the space provided. Include page numbers where applicable.

Are there any technical errors or misrepresentations in the document?

Is the material presented in a logical and consistent order?

Is it easy to locate specific information in the document?

Is there any information you would like to have added to the document?

Are the examples relevant to the task being described?

Could parts of the document be deleted without affecting the document's usefulness?

Did the document help you to perform your job?

Any general comments?

NAME

TITLE

COMPANY

ADDRESS

TELEPHONE NO. ( )

Thank you for your evaluation of this document.
Fold the form as indicated and mail to NCR. No postage is necessary in the U.S.A.

TAPE

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.3 DAYTON, OHIO

POSTAGE WILL BE PAID BY ADDRESSEE

NCR Corporation

ATTENTION: Publication Services
WHQ-4

Dayton, Ohio 45409

NCR
Personal Computer Division
Augsburg, Germany

ST-2106-34 0885