DIGITAL RESEARCH®

Pascal/MT+™

Language

Programmer's Guide

For the CP/M-68K™
Operating System

The Pascal/MT+ Language Programmer’s Guide for the CP/M-68K Operating System was prepared using the Digital Research TEX™ Text Formatter and printed in the United States of America.

****
* First Edition: May 1984 *
****

Foreword

The Pascal/MT+™ language is formally based on the definition of standard Pascal as described in the International Standards Organization (ISO) standard DPS/7185. Pascal/MT+ also has several additions to standard Pascal that make it more suitable for commercial programming. You can use Pascal/MT+ to develop high-quality, efficient, maintainable software for both data processing and real – time control applications.

The Pascal/MT+ system, which includes compilers, linkers, and programming tools, is implemented on a variety of operating systems and microprocessors. Pascal/MT+ programs are easily transportable between various target processors and operating systems because the language is consistent. The Pascal/MT+ system can also generate software for use in a ROM-based environment, to operate with or without an operating system.

Th**is** manual describes Pascal/MT+ Release 3.3, a system that runs under the CP/M-68K™ operating system using a Motorola© MC68000 microprocessor with at least 128K bytes of memory.

This manual is divided in five parts. Part 1 (Sections 1 – 2) gives a general description of the Pascal/MT+ programming system, the notational conventions used in the manual, and some guidelines for configuring your own system.

Part 2 (Sections 3 – 5) describes how to use software. This includes the operation of the compiler, linker, and disassembler. The command-line and source-code options for the compiler and the command-line options for the linker are fully described.

Part 3 (Sections 6 – 10) describes a variety of advanced topics, including how to write large programs in segments, how to inter face Pascal/MT+ code with assembly-language code, how to directly access the operating system, and how to write your own error handling procedures. Also included is a set of guidelines for writing ROM-based code.

Part 4 (Section 11) contains some sample Pascal/MT+ programs that illustrate various features of the language. You can study these examples, and then modify them to gain experience with the language. Part 5 is a reference section containing appendixes and an index.

This manual assumes you are already familiar with general aspects of computer programming, and may have programmed in a language such as BASIC. If you are not familiar with Pascal, refer to the Pascal/MT+ Language Reference Manual for a bibliography of textbooks. This manual also assumes that you are familiar with the CP/M-68K operating system and your own hardware components.

Contents

Tables

Figures

Listings

1. Section

Introduction To Pascal/MT+

The Pascal/MT+ Language

Pascal/MT+ is a high-level, block-structured, programming language. It is formally based on the definition of standard Pascal as described in the International Standards Organization (ISO) standard 7185.

The Pascal/MT+ language is a superset of standard Pascal. That is, Pascal/MT+ has all the features and constructs of standard Pascal, as well as enhancements that make it suitable for writing professional applications and system-level programs.

The enhancements fall into four categories:

- **additional data types**
- **enhanced file handling and input/output capability**
- **accesses both the run-time and operating systems**
- **writes modular programs using overlays and chaining**

Collectively, these enhancements make Pascal/MT+ more suitable for commercial programming in both data processing and real-time control applications.

The Pascal/MT+ language is also the basis of a complete software development system that includes compilers, linkers, subroutine libraries, and other programming tools.

Pascal/MT+ Implementation

An implementation of the Pascal/MT+ language is a particular combination of software and hardware components that can translate the language's statements into machine-readable instructions for a target system.

Software components include the compiler, linker, run-time libraries, and other tools such as assemblers, disassemblers, and symbolic debuggers. Hardware components include microprocessors, random access memory, disk storage, and peripheral devices such as consoles and printers. Thus, there can be many implementations of the Pascal/MT+ language, each tailored for a particular hardware/software combination.

Every implementation of Pascal/MT+ must support all the syntactical constructs of the language and translate language statements in conformance with the ISO standard. However, each implementation can differ in the way it internally represents data, or organizes and transfers files.

Digital Research has implementations of Pascal/MT+ for a variety of 8-bit and 16-bit microprocessors and operating system environments. Because of differences in the capabilities of various microprocessors and operating **systems,** not all the extensions of Pascal/MT+ are supported in each implementation.

Pascal/MT+ for CP/M-68K

Pascal/MT+ for CP/M-68K is a complete programming system that includes a compiler, a linker, a disassembler, and a large library of run-time subroutines to help you build better programs faster.

Figure 1-1 illustrates the software development process using the Pasca1/MT+ system.

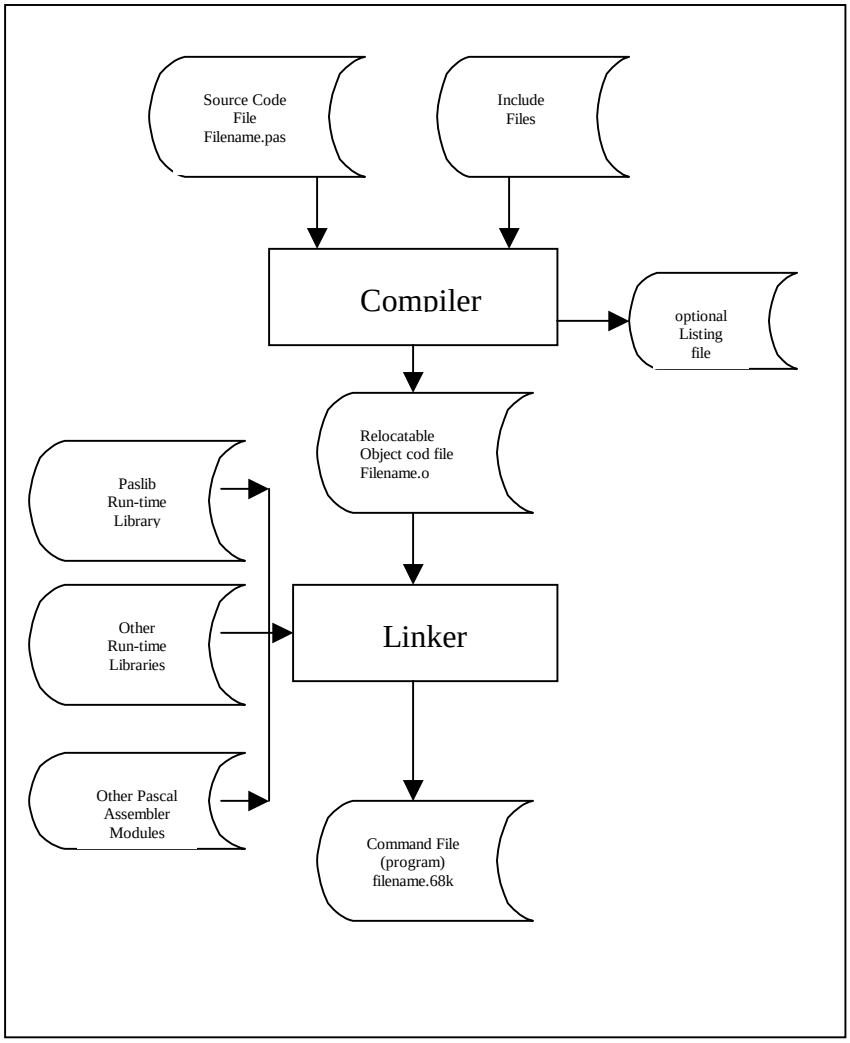


Figure 1-1 Software Development under Pascal/MT+

Pascal/MT+ Documentation Set

The Pascal/MT+ Language Programmer 's Guide for the CP/M-68K Operating System, cited as Programmer's Guide, contains information about using the compiler, linker, and disassembler. It provides general guidelines for creating modular programs using overlays, chaining, and shared variables.

The Programmer's Guide also contains information on advanced programming topics, including how to write large programs in segments, interfacing Pascal/MT+ programs with assembly language modules, direct access to the operating system from Pascal/MT+ programs, and writing your own error handling routines. There is also a set of guidelines for writing ROM–based code.

The Pascal/MT+ Language Reference Manual, cited as Language Reference Manual, describes the Pascal/MT+ language, its syntax, and semantics. It is not a programming tutorial. Rather, it is primarily a reference document and should be used in conjunction with the Programmer 's Guide.

The Pascal/MT+ documentation set assumes you have general experience with computer programming and possibly with standard Pascal. If you are a beginning programmer, or if you are not familiar with Pascal, you should refer to the bibliography of textbooks 1isted in the Language Reference Manua1.

The documentation set also assumes you are familiar with your own hardware components and operating system.

Notational Conventions

The following notational conventions are used throughout this manual:

…	Horizonta1 ellipses indicate the immediately preceding item can occur once, or any number of times in succession.
. . .	Vertical ellipses indicate an omitted portion of a source program or example; only the relevant part is shown.
␣	Represents a blank space.
Bracket [	Source code in examples and program listings has a bracket on the left side to illustrate and emphasize the block structure of the language.
color	Items in color represent literal examples including source code listings, sections of code, or single statements. Also, any system output such as error messages and system prompts are in color. User input is in boldface color.
CTRL	In the text, CTRL represents a control character. Thus, CTRL-C means Control-C. In any listing that shows example console interaction, the symbol ^ is the echo of a control character.
n	A numeric value indicates a decimal number unless otherwise stated.
nH	A numeric value followed by the capital letter H indicates the number is a hexadecimal (base 16) value.
lowercase	Variable information in example statements is in lowercase.
UPPERCASE	Words in uppercase are Pascal/MT+ reserved words or predefined identifiers. For example, ARRAY, ELSE, RECORD, INTEGER, TEXT, WRITELN. Names of procedures and sample programs when referenced in the text are also in uppercase.

This manual also uses the following symbolic convent ions to formally describe the syntax of Pascal/MT+ statements:

|	The vertical bar indicates a choice between the items it separates. You pronounce the symbol "or . "
{}	Items inside curly braces are optional. Optional items can be repeated any number of times.
<>	Items inside angle brackets in lowercase letters, or in a combination of lowercase letters and digits separated by a hyphen, represent variable information for you to select. These items are described or defined more explicitly in the text, if necessary.
literals	Any item not in angle brackets or curly braces are literal. Enter them just as they appear in text.

End of Section 1

2. Section

Getting Started

Hardware Requirements

The Pascal/MT+ system runs under the CP/M-68K operating system using a Motorola MC68000 microprocessor. The compiler and linker need at least 192K bytes of memory, but it is recommended that your system have 256K bytes to handle large programs.

The size of a program developed with Pascal/MT+ depends on the size of the source code and on the number of run-time subroutines it uses. For example, compiling, linking, and then using the RELOC utility on the minimal program TEST1.PAS (described later in this section) generates a command file of 5K bytes.

Pascal/MT+ System Files

Digital Research supplies the Pascal/MT+ system in a variety of disk formats. When you receive your distribution disks, be sure to examine the file named READ.ME. This file completely describes the contents of all the other files on each of the distribution disks.

The Pascal/MT+ system uses a variety of filetypes, described in Table 1-1.

Table 2-1. Pascal/MT+ System Filetypes

Filetype	Contents
S	assembly-language source file for AS68
DIS	disassembled listing (de fault)
DOC	document file; contains printable text in **ASCII** form
ERR	error message file output by compiler
L68	library file; contains subroutines
LIS	print file output by compiler
0	relocatable 68K-format object file; contains relocatable object code emitted by the compiler
PAS	Pascal source file; contains source code in **ASCII** form (the compiler also accepts SRC as a source file type)
TDT	temporary initialized data file used by disassembler, DIS68; normally erased at end of compilation
TNP	temporary file used by compiler; normally erased at end of compilation
TRL	temporary object file used by disassembler, DIS68; normally erased after compilation
TSY	temporary symbol table file used by disassembler, DIS68; normally erased at end of compilation
TXT	text file; contains text of messages output by compiler
68K	command file; runs directly under CP/M-68K
nnn	hexadecimal n; used for numbering overlays

Installing Your System

The first thing you should do when you receive your Pascal/MT+ system is make a backup copy of all the distribution disks.

Note: You have certain responsibilities when copying Digital Research products. Read your licensing agreement.

When installing your own system, you might find it convenient to copy only specific files from the distribution disks. The way in which you configure your system depends on **its** actual hardware capabilities.

System with Hard-disk Drive

If your system has a hard disk, the easiest way to configure it is to put the compiler files, the linker, and run-time library files on one logical drive.

The compiler files are as follows:

- **MT68.68K**
- **MT68.000**
- **MTERRS.TXT (optional)**

The linker and run-time libraries are as follows:

- **LINK68 . 68K**
- **PASLI B. L68**
- **BCDREALS.L68**
- **FPREALS.L68**
- **FULLHEAP . 0**

System with Floppy-disk Drives

If your system has two floppy-disk drives, you can use one disk for compiling and another disk for linking. You can use other disks for the programming tools, assorted source code, and examples.

To configure separate disks for compiling and linking, perform the following steps:

1. Install the CP/M-68K operating system, the **PIP** utility, and a text editor on two blank disks. Label one disk as the compiler and the other as the linker.

Put the following files on the compiler disk:

- MT68.68K
- MT68.000
- MTERRS.TXT (optional)

2. Put the following files on the linker disk:

- LINK68.68K
- PASLIB.L68
- BCDREALS.L68
- FPREALS.L68
- FULLHEAP.O

This suggestion is one way of configuring your disks. All the compiler modules must be on one disk. For simplicity, put all the related relocatable files on the same disk.

Note that compiler can run without the error message file MTERRS.TXT. If your compiler disk is short of space, you can eliminate this file.

Compiling and Linking a Program

If you have never used Pascal/MT+ before, the following step- by-step example shows you how to compile, link, and run a simple program. This example assumes that you are using a CP/M-68K system with two disk drives and that you are familiar with CP/M-68K.

1. Put the compiler disk in drive A.

2. Using the text editor, create a file called TEST1.PAS and enter the following program. Use PIP to put the file on drive B.

```
PROGRAM simple example;

VAR
 i : INTEGER;

BEGIN
 WRITELN('This is just a test');
 FOR i := 1 TO lO DO
    WRITELN(i);
 WRITELN('A11 Done')
END.
```

3. Now, compile the program with the following command:

```
A>mt68 b:testl
```

   If you examine your directory, you will see a file named **TEST1.O** that contains the relocatable object code emitted by the compiler. If the compiler detects any errors, correct your source program and try again.

4. Now, log on to drive B, and link the program using the fo1lowing command:

```
B>link68 testl,paslib.168
```

   Your directory will now contain a file named **TEST1.68K** that runs directly under **CP/M-68K.**

5. To run the program, enter the command:

```
B>testl
```

Although the test program shown above is very simple, it demonstrates the essential steps in the development process of any program: editing, compiling, and linking.

3. Section

Using the Compiler

Compiler Organization

The Pascal/MT+ compiler is composed of two files:

- **`MT68.68K`**
- **`MT68.000`**

When you invoke the compiler, CP/M-68K loads the root module, MT68.68K, which performs the initial processing, then chains to the second module, MT68.000, to continue processing.

Compiler Operation

The Pascal/MT+ compiler processes a source-code file in three separate steps called passes or phases.

- **Phase 0 checks the syntax and generates a token file named PASTMP.TOK. This file contains an intermediate language (tokenized) version of the source code.**
- **Phase 1 generates a table of the symbols that are defined in the source code. The compiler uses this symbol table when generating the relocatable object-code file in Phase 2.**
- **Phase 2 generates the relocatable object-code f ile.**

The compiler also creates temporary files on the same disk containing the source code file. Under normal conditions, the compiler deletes the temporary files when finished processing. However, if the compiler terminates abnormally, the temporary files can remain in the directory.

When you compile the program, make sure there is enough space on the disk, or use the Td option to specify a different disk for the temporary files. See Command-line Options in this section.

Invoking the Compiler

You invoke the compiler with a command line of the form

MT68 **<** filespec> [<opt ions **>** **]**

where <filespec> is the source-code file to be compiled, and the <options> are a list of optional parameters that control the compilation process.

The compiler can read the source file from any disk. The <filespec> must be in Digital Research standard filespec format, and end with a carriage return, line feed, and CTRL-Z.

When you create Pascal/MT+ programs, make sure that your text editor does not insert nonprintable formatting characters in the source file. The compiler cannot process a file containing any nonprintable control characters except tabs. Some text editor**s** use nonprintable **ASCII** characters to control formatting.

If you do not specify a filetype, the compiler searches for the file **with** no filetype. If the compiler cannot find the file, it assumes a SRC filetype, then a PAS filetype. If the compiler still cannot find the file, it displays an error message and stops processing.

The compiler generates a relocatable object-code file with the same filename as the input source program. The relocatable file has the filetype O.

Compilation Data

The Pascal/MT+ compiler periodically outputs information during Phases 0 and 1 to assure you it **is** running properly.

During Phase 0, the compiler outputs a plus sign (+) to the console after scanning every 16 lines of

source code.

At the beginning of Phase 1, the compiler indicates the total amount of memory space available. The compiler also indicates the amount of memory space available after the predefined (internal compiler) symbols are loaded. This second indication is the amount of memory left for user symbols. Both amounts are shown in decimal.

During Phase 1, the compiler also outputs a pound sign (#) to the console each time it reads a procedure or function. At completion of Phase 1, the compiler indicates the total number of bytes remaining in memory.

Phase 2 generates the relocatable object code. During this phase, each time the compiler encounters a procedure or function, it displays the procedure's name, its offset from the beginning of the module, and its size in decimal.

When the processing is completed at the end of Phase 2, the compiler displays the following diagnostic information:

```
Lines :  lines of source code compiled
Errors:  number of errors detected
Code  :  bytes of code generated (in decimal)
Bss   :  bytes of block storage used (in decimal)
```

Compilation Errors

During Phase 0, when the compiler detects a syntax error, it displays the line containing the error. If you are using the MTERRS.TXT file, the compiler also displays an error description. In all other phases, the compiler displays an error identification number.

When the compiler is building the symbol table in Phase 1, overflow occurs if not enough space remains for the current symbol. Symbol table overflow is a non-recoverable error. You can overcome the problem by using the $Kn option to eliminate unused symbols from the table (see Source Code Options in this section). You can also try to segment the program into smaller modules and compile them separately (see Section 7, "Writing Segmented Programs").

In all phases, when the compiler detects an error it asks if you want to continue or stop, unless you use the C command line option. See Command Line Options, in this section.

Note: You must ensure that all the compiler overlays are on the same disk. If the overlay manager in the run-time system cannot find an overlay, it displays an error message and stops processing.

Usually you can find a missing overlay by ensuring that the filename is correct and that it is on the disk. If you cannot find it, recopy the overlay file from your distribution disk. If you are sure the overlay is on the disk and you still get an error message, then the file is corrupted.

When all processing is completed, the ERR file generated by the compiler summarizes all non-syntactic errors.

Appendix A contains a complete list of the error messages, their causes, and suggested responses.

Compiler Command-line Options

Command-line options control specific actions of the compiler, such as where it writes the output files. All command-line options are single letters that start with a dollar sign ($) or a pound sign (#). If you specify more than one option, do not put any blanks between the options.

Certain options require an additional parameter to specify a disk drive or other I/O device.

The command-line options are listed below.

B, BCD Representation

The B option tells the compiler to internally represent REAL numbers using Binary Coded Decimal (BCD) instead of a floating- point format. The default is to represent REAL numbers using floating-point format. Refer to Appendix D for more information about internal representation of data.

C, Continue on Error

The C option tells the compiler to continue processing the source-code file whenever it encounters an error. The default is to stop at each error and ask whether to continue or not.

Dd, Disassembled File Location

The Dd option tells the compiler to put the disassembled listing file on the I/O device d. d can be any logical disk drive, A through 0, or the currently logged-in drive. You can also specify X, the console or P, the printer. By default, the compiler outputs the disassembled listing file at the console.

Ed, **Error File Locat ion**

The Ed option **tells** the compiler that the error **message text file, NTERRS.TXT, is** located on disk d. d can be any logical **disk drive, A** through O. By default, the compiler searches for **MTERRS.TXT** on the default (currently logged-in) disk.

Od, MT68.000 Overlay Location

The Od option tells the compiler that the overlay file MT68.000 is located on disk d. By default, the compiler searches for MT68.000 on the same drive as the MT68.68K file.

Pd, Print (Listing) File Location

The Pd option tells the compiler to put the print file (LIS) on the I/O device d. d can be any logical disk drive, A through 0, or the currently logged-in drive, designated by Q. You can also specify X (the console) or P (the printer) . By default, the compiler does not create a print file.

Q, Quiet Operation

The Q option tells the compiler not to display any unnecessary diagnostic messages on the console. By default, the compiler displays all diagnostic messages on the console.

Rd, Relocatable File Location

The Rd option tells the compiler to put the relocatable object-code file on disk d. d can be any logical disk drive, A through O. By default, the compiler puts the relocatable object-code file on the default (currently logged-in) disk.

Td, Temporary File Location

The Td option tells the compiler to put the temporary files on disk d. d can be any logical disk drive, A through O. By default, the compiler puts the temporary files on the default (currently logged-in) disk.

V, View Procedures and Functions

The V option tells the compiler to print at the console the name of each procedure and function it encounters in the source-code file during Phase 0. Such procedure and function names can be useful for finding errors during Phase 0. By default, the compiler does not print the names of procedures and functions during Phase 0.

X, Extended Relocatable Object File

The **X** option tells the compiler to generate an extended relocatable object-code file containing encoded source-code line number information. By default, the compiler does not generate this

information, and you cannot disassemble the object-code file. The **X** option also tells the compiler not to erase the temporary files at the end of compilation because these files are used by the disassembler.

@, Pointer Character Equivalence

The @ option tells the compiler to treat the @ character as equivalent to the standard pointer reference character (^) . When you use this option, you cannot use the @ character as the first character in an identifier. By default, the compiler does not treat @ as equivalent to ^.

The following is an example command line:

```
A>mt68 a:testprog $rbtbvpp
```

This command line tells the compiler to read the source-code file from drive A, write the relocatable object-code file and the temporary files to drive B, print procedure and function names during Phase 0, and send the listing file to the printer.

Table 3-1 summarizes the compiler command-line options and their de fault values.

Table 3-1 Compiler Command-line Options

Option	Meaning	Default
B	Use BCD rather than floating point binary for real numbers.	Floating point binary reals.
C	Continue compiling when error is encountered.	Compiler stops and ask s on each error.
Dd	Put the disassembled listing on device d: d = A..O, X,P	Show disassembled listing on console.
Ed	MTERRS.TXT file is on disk d:d = A..O	MTERRS.TXT on default disk.
Od	MT68 .000 file is on disk d: d = A..O	MT68.000 on same disk as MT68.68K.
Pd	Put the print (listing) file on device d: d = A..O, X,P	No print file.
Q	Quiet; suppress any unnecessary console messages.	Compiler outputs all messages.
Rd	Put the relocatable object- code file on disk d: d = A..O	Relocatable file on default disk.
Td	Put the temporary files on disk d: d = A..O	Put temporary files on default disk.
V	Print the name of each procedure and function found in source code during Phase 0.	Procedure names not printed.
X	Generate an ex tended relocatable object-code file including disassembler information; do not erase temporary files used by the disassembler.	Relocatable file cannot be disassembled and temporary files are erased.
@	Make the @ character equivalent to the ^ character.	@ not equivalent to ^

Note: The A option has no effect as in other implementations; the compiler ignores it.

Source Code Options

Source-code options are special instructions to the compiler that you put in the program source code A source-code option is a single lower or uppercase letter preceded by a dollar sign, embedded in a comment The option must be the first item in the comment. Certain source-code options require additional parameter **s.**

You can put any number of options in a source program, but only one option per comment is allowed. You cannot place blanks between the dollar sign and the option letter. The compiler accepts blanks between the option letter and the parameter.

The source-code options are listed below.

E, Entry-point Record Generation

The E option controls the generation of entry-point records in the relocatable object-code file. Enable the E option using a + parameter and disable it using a – parameter. E+ is the default.

E+ tells the compiler to generate entry-point records for variables, procedures, and functions declared at the outermost (global) level o f the program. You can reference a global variable, procedure, or function in a separate module if the module uses the same declaration and the reserved word EXTERNAL .

E- tells the compiler not to generate entry-point records, thus making all variables, procedures, and functions local to the block where they are defined.

I, Include Files

The I option tells the compiler to include a specified file for compilation in the input stream of the original program. The compiler supports only one level of file inclusion, so you cannot nest Include files. The form of the option is

I <filespec>

where <filespec> must be in standard format. If you omit the drive specification, the compiler looks on the default drive. If you omit the filetype, the compiler supplies the same filetype as the original source file. If the compiler cannot find the file, it displays an error message and stops processing. The file must end with a carriage return, line feed, and CTRL-Z.

Kn, Symbol Table Space Reduction

The Kn option tells the compiler to make more room in the symbol table for user symbols by removing any predefined symbols that are unreferenced in the source program. Examples of predefined symbols are the Pascal/MT+ reserved words and names of predefined functions and procedures. These predefined symbols normally take about 6K bytes of symbol table space.

The form of the option is

Kn

where n is an integer parameter ranging from 0 to 15. Each integer corresponds to a different group of routines as defined in Table 3-2.

You must enter all K options before the reserved words PROGRAM or MODULE in the source code. You can use as many K options as required, but place only one integer parameter after each letter K. Note that if the program makes any reference to a symbol removed with the K option, the compiler issues the following error message:

```
UNDECLARED IDENTIFI ER
```

Table 3-2 $K Option Values

Group	Routines Removed
0	ROUND, TRUNC, EXP, LN, ARCTAN, SQRT, COS, SIN
1	COPY, IN SERT, PO S, DELETE, LENGTH, CONCAT
2	GNB, WNB, CLOSEDEL, OPENX, BLOCKREAD, BLOCKWRITE
3	CLOSE, OPEN, PURGE, CHAIN
4	WRD, HI, LO, SWAP, ADDR, SIZEOF, INLINE, EXIT, PACK, UNPACK
5	IORESULT, PAGE, NEW, DISPOSE
6	SUCC, PRED, EOF, EOLN
7	TSTBIT, CLRBIT, SETBIT, SHR, SHL
8	RESET, REWRITE, GET, PUT, ASSIGN, MOVEL EFT, MOVE R IG HT, FILLCHAR
9	READ, RE ADLN
10	WRI TE, WRI TEL N
11	unused
12	MEMAVAIL, MAXAVAIL
13	SEEKREAD, SEEKWRITE
14	unused on the 68000
15	unused on the 68000

L, P; Listing Controls

The L option controls the listing that the compiler generates during Phase 0. Enable the L option with the + parameter and disable it with the – parameter. L+ is the default.

The P option starts a new page by placing a form-feed character in the listing file.

R, Run-time Range Checking

The R option tells the compiler to generate run-time code that performs range checking for array subscripts and assignment to subrange variables. Enable the R option with the + parameter and disable it with the – parameter. R- is the default. Refer to Section 9.6 for information on range checking.

T,W; Type and ISO Standard Checking

The T option controls the compiler 's strict type -checking/non ISO-standard warning facility. The **W** option controls the display of warning messages pertaining to the **T** option. Enable both options with the + parameter and disable them with the - parameter. The default value for both options **is -.**

The T+ option tells the compiler to per form strict type checking. If the **T** and **W** options are both

enabled and the compiler detects a non ISO-standard feature, the compiler displays the message

```
NON-ISO STANDARD FEATURE
```

For example, when both options are enabled, string operations generate this message because the **STRING** data type is non ISO- standard.

X, Exception (Error) Checking at Run-time

The X option tells the compiler to generate code that performs error checking at run-time. Error checking covers division by zero (both integer and real numbers) and real number over flow/underflow.

You enable the X option with the + parameter and disable it with the – parameter. By default, error checking is always enabled in this version. Refer to Section 9.6 for information on run-time error handling.

The following examples show proper source-code compiler options:

```
($p)
($e+)
($kO)
($i d:userfile.lib)
```

For reference, Table 3-3 summarizes the source-code compiler options.

Table 3-3 Compiler Source Code Options

Option	Function	Default
E +/-	controls entry point generation; makes variables and routines either global or local	E+
I<filespec>	includes another source file into the input stream, for example, ($I MATH.LIB}	
Kn	removes predefined routines to save space in symbol tab le (n = 0..15)	
L +/-	controls the 1isting of source code	L+
P	enters a form feed in the LIS file	
R+/-	controls range checking code	R-
T +/ –	controls strict type checking	T-
W +/-	generates warning messages for non-ISO standard features	W-
X +/-	controls exception checking code	X+

Note: The Cn, Qn, S, and Z options have no effect as they do in other implementations; the compiler ignores them.

Conditional Compilation

Pascal/MT+ supports conditional compilation directives so that you can compile alternative versions of a single source-code file. This facility can be very useful when compiling large application programs that are designed to run in different hardware or operating system environments. You can isolate the environment dependent code and then compile different versions based on some conditional test**.**

The conditional compilation directives are

- **&SET**
- **&IF**
- **&ELSE**
- **&END**
- **&MSG**

Put conditional compilation directives in the source code as you do for other options. Each directive begins with an ampersand character (&), and must be in the first column.

You use the &IF and &END directives to delimit the section of source code you want to conditionally compile. The syntax is shown below.

```
&IF ( <variable> )

   <source line 1>
   .
   .
   .
   <source line n>

[&ELSE]

   <alternate source line 1>
   .
   .
   .
   <alternate source line n>

&END
```

If the value of the <variable> is TRUE, the source code in lines 1 through n is compiled. If the value is FALSE, the compiler ignores the lines and continues compiling at the line immediately following &END.

If the value is FALSE and you use the optional &ELSE directive to specify an alternative section of code, the compiler ignores lines 1 through n, compiles the alternate source code instead, and continues at the line immediately following &END**.**

You must define the test <variable> using the syntax

```
6 SET <variable>
```

The most common way to use conditional compilation is to put several &SET directives in an Include file and select the proper version by placing comments around any directives not wanted. To compile a different version, simply remove the comments .

For example, if the Include file contains the code

```
(* &SET verl *)
&SET ver2
```

the compiler processes the source code delimited by

```
&IF ver2
   .
   .
   .
&END
```

However, if the Include file contains the code

```
&SET verl
(* &SET ver2 *)
```

the compiler processes the source code delimited by

```
&IF verl
   .
   .
   .
&END
```

The &MSG directive outputs a diagnostic message to the console that tells you which section of code is being conditionally compiled. For example,

```
&IF ver2
&MSG Now compiling version #2
   .
   .
   .
&END
```

The message must be an **ASCII** string not exceeding 80 characters.

End of Section 3

4. Section

Using the Linker

LINK68™ is the linkage editor that combines object-code files into a command file. You can also use LINK68 to link a program as a set of overlays (see "Overlays," in Section 7).

LINK68 accepts the object-code files produced by Pascal/MT+ compiler and produces an executable file in the 68K command file format. LINK68 also accepts object-code files produced by any CP/M-68K language processor including AS68, CB68K, and the Digital Research C compiler.

LINK68 resolves all references to external symbols and concatenates the object-code files in the order you specify in the command line. The entry point of the resulting command file is the first instruction in the first object-code file.

Figure 4-1 illustrates LINK68 operation.

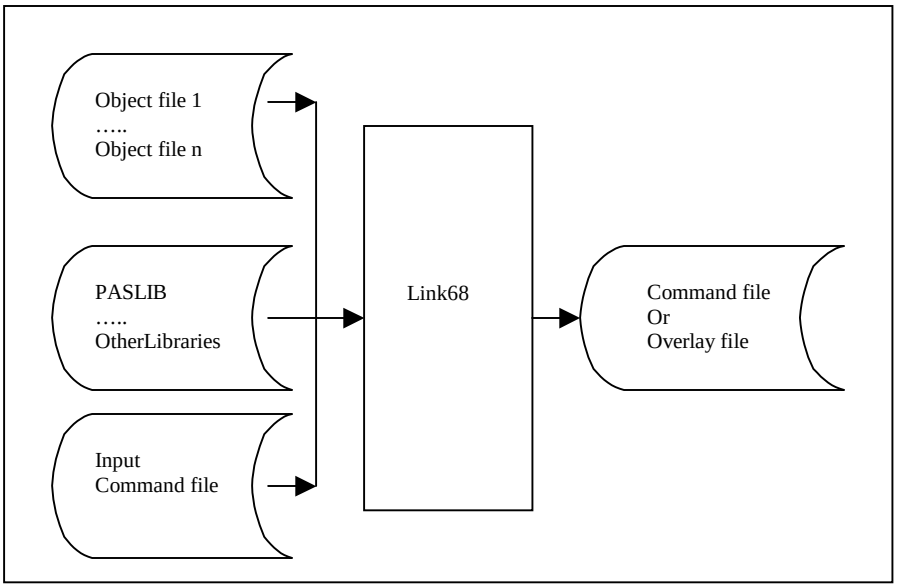


Figure 4-1 LINK68 Operation

Run-time Libraries

Although the Pascal/MT+ compiler generates native machine code, each implementation requires a library of run-time routines to handle file processing and other features that are not supported by the native hardware.

PASLIB

The main run-time library is called PASLIB, for Pascal Library. All I/O is per formed and all set variables are manipulated with PASLIB routines. Console I/O is assumed by the initialization routine, **_INI,** so the I/O routines are always loaded. You can avoid this by writing a replacement **_INI** routine and linking it before linking PASLIB to resolve the **_INI** reference.

Other Libraries

Other libraries contain routines that are required by any program using real numbers in either BCD or floating-point format, or per forming calculations with transcendental functions, or random access I/O operations. Table 4-1 summarizes these libraries. Appendix C contains a complete list of the routines in each library.

Table 4-1 Required Libraries

Library	Contents
BCDREALS	BCD real-number routines
FPREALS	Floating-point real-number routines
FULLHEAP	Memory management routines

Note: You must use LINK68 to create an executable command file even when a single object-code file contains no undefined symbols.

Invoking the Linker

You invoke LINK68 with a command line of the form:

LINK68 {file=} object-file-1[,object-file-2,...object-file-n]

where file is the name of the command file you want to create, and object-f ile-1 through object-file-n are the object-code files to link.

If you invoke LINK68 without a command tail, the linker lists the options and returns control to the operating system.

If you enter a filename to the left of the equal sign, LINK68 creates the output file with that name. For example, the command

```
A>link68 math = sin,cos, tan
```

creates the command file MATH. However, if you omit the filename to the left of the equal sign, LINK68 creates the command file using the first filename in the command line and assigns the default filetype 68K. For example,

```
A>link68 sin, cos, tan
```

creates the command file SIN.68K.

LINK68 ignores anything past a backslash (\) character, so you can put comments in a command line. See the example below.

Errors

When LINK68 detects an error while processing, it returns an error message in the following form:

```
LINK68: < Error Message>
```

Most linkage errors are nonrecoverable and prevent your program from linking. Appendix B contains a complete list of the LINK68 errors with explanations and suggested user responses.

Redirecting Output

Normally, LINK68 sends all diagnostic output to the console. However, you can redirect this output by using the > character in the command line. For example, the command

```
A>link68 [tem[b:] myfile.68k = moda, modb >d:lnkmsgs.txt
```

creates MYFILE.68K on drive A, using drive B for the temporary files, and sends the diagnostic output to the file **LNKMSGS.TXT** on drive D.

Linker Command-line Options

When you invoke LINK68, you can specify command-line options that control the link operation.

There are two kinds of options: global and local. Global options apply to the entire link operation. Local options apply only to the individual files being linked. You enclose both kinds of options in square brackets.

You enclose global options in square brackets immediately preceding the command filename (if specified) in the command line. You enclose local options in square brackets immediately following the filename to which they apply.

You can use spaces between filenames to improve readability in the command line, and you can put more than one option in square brackets by separating the options with commas. LINK68 also allows you to abbreviate an option name to its shortest unambiguous form.

The command-line options are listed below.

ABSOLUTE

Tells LINK68 to generate an absolute command file with no relocation bits. The default is a relocatable command file.

ALLMODS

Tells LINK68 to load all modules from a library, even if they are not referenced. The default action is to include only those modules that are actually referenced.

BSSBASE[n]

Specifies the base address for the Block Storage Segment (bss) containing the uninitialized data in discontiguous programs. n is a hexadecimal value. The default value is the first even word after the Data segment. You cannot use this option when linking overlaid programs.

COMMAND

Tells LINK68 that the following named file contains the rest of the command line. LINK68 ignores the rest of the main command line. Nested command files are not allowed.

The format of this option is

```
COMMAND [filename]
```

where filename is the file containing the rest of the command line.

DATABASE (n)

Specifies the base address of the Data segment in discontiguous programs. n is a hexadecimal value. The default is the first even word after the Text segment. You cannot use this option when linking overlaid programs.

IGNORE

Tells LINK68 to ignore 16-bit address overflow and continue processing. The default action is to issue an error message and stop processing.

INCLUDE

Tells LINK68 to load an unreferenced module from a library. The format for this option is

```
filename [INCLUDE [module-name]]
```

where module-name is the module you want to load.

LOCALS

Tells LINK68 to put local symbols in the symbol table. The default is no local symbols. LOCALS only

applies from the point in the command line that it appears.

The NOLOCALS option turns this option of f. Use LOCALS and NOLOCALS in combination to put local symbols from specific files in the symbol table. LINK68 always ignores local symbols starting with L.

NOLOCALS

See LOCALS.

SYMBOLS

Tells LINK68 to put the symbol table in the command file. The default is no symbol table in the command file.

TEMPFILES[d:]

Tells LINK68 to use drive d for the temporary files it creates during processing. The default is the currently logged-in drive. If you use TEMPFILES, it must precede any input files on the command line.

TEXTBASE [n]

Specifies the base address for the Text segment t. n is a hexadecimal value. The default is 0H. You can use this option when linking overlaid programs.

UNDEFINED

Tells LINK68 to ignore the presence of undefined symbols in the input files. LINK68 lists the undefined symbols, and then continues processing. The default action is to list any undefined symbols and then stop processing.

The following are examples of LINK68 command lines. Addresses are in hexadecimal.

```
A>link68 [sym, tem[b:]] math = mathmain,mathlib
```

This command links the files MATHMAIN and MATHLIB into a command file named MATH**.** It also tells LINK68 to include the symbol table in MATH, and place the temporary files on drive B.

```
A>link68 [com[linkit.inp
```

This command tells LINK68K to read the command line from the file LINKIT. INP. Note that closing brackets are not needed. The file LINKIT. INP might contain the following commands:

```
link68 [ab, tex[500], d[2aOO], b[3000]] screen = \ too long scrnsl[l],
iolib[al]
```

This command creates the file SCREEN from the files SCRNS1 and IOLIB. The command tells LINK68 to create SCREEN as an absolute command file with the Text segment starting at 500H, the Data segment starting at 2AOOH, and the uninitialized Data segment starting at 3000H. It also tells LINK68 to include local symbols from SCRNS1 and all the modules in IOLIB.

Table 4-2 lists the LINK68 options, their abbreviations, and defau1t **s.**

Table 4-2 LINK68 Command-line Options

Option	Abbrev.	Purpose	Default
ABSOlUTE	AB	generates an absolute file	generates relocatable file
ALLMODS	AL	loads all modules	loads only the modules referenced
BSSBASE[n]	B[n]	sets base address of bss segment	first even word after Data segment
COMMAND	C	gets command line from a file	
DATABASE[n]	D[n]	sets base of the Data segment	first even word after Text segment
IGNORE	IG	ignores 16-bit address overflow	stop; issue error message
INCLUDE	IN	loads a module	
LOCALS	LO	puts local symbols in symbol table	no local symbols
NOLOCALS	NO	turns off LOCALS	
SYMBOLS		puts symbol table in command file	no symbol table
TEMPFILES[d:]	TEM[d:]	puts temporary files on drive d	currently logged-in disk
TEXTBASE[n]	TEX[n]	sets base of Text segment	0H
UNDEFINED	U	ignores undefined symbols and continue	lists undefined symbols and stop

End of Section 4

Using the Disassembler

DIS68 is a utility program that enables you to disassernble the machine code produced by the compiler into a series of assembly-language instructions. This can be very useful when debugging a program at the machine-code level.

In order to disassemble a program, you must compile the source code using the Pd and X command-line options.

The Pd option tells the compiler to generate a print (listing) file with filetype LIS.

The **X** option tells the compiler to generate an extended relocatable object-code file. This extended file contains the assembly-language code emitted by the compiler, and source-code line number information in encoded form. The X option also tells the compiler not to erase the temporary files needed by DI S68.

DIS68 combines the extended relocatable object-code file, the LIS file, and the temporary files (TRL and TSY**)** to produce a file showing the assembly-language code generated for each line of source code.

Figure 5-1 illustrates the operation of DIS68.

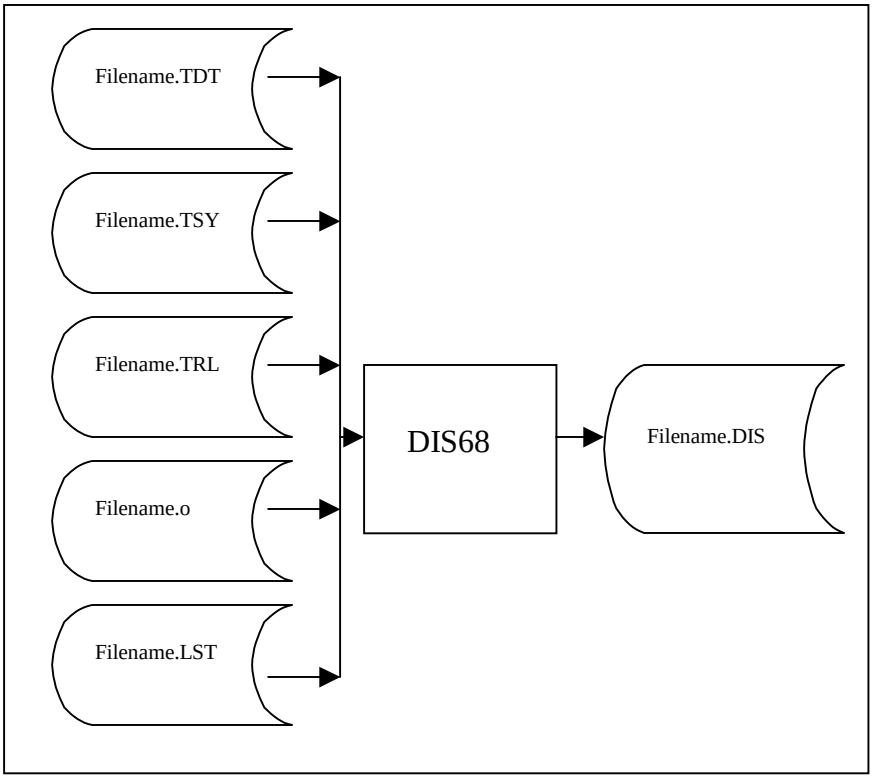


Figure 5-1 DIS68 Operation

Invoking DIS68

DIS68 is automatically invoked when you compile a program with the command-line options Pd and X**.** The compiler chains to the disassembler at the end of the compilation. The object-code file, the listing file, and the temporary files must all be on one logical disk drive.

You can use the Dd command-line option to specify the location of the disassembled listing. The

location can be a disk file or a Pascal/MT+ logical device, CON: or LST:. The default destination is CON:. If you specify a disk file, DIS68 supplies the default filetype DIS.

For examp1e, the command

```
A>mt68 mathlib $xp
```

compiles, then disassembles the file MATHLIB and sends the disassembled listing to the console. The command

```
A>mt68 mathlib $xpbd:
```

compiles, then disassembles the file MATHLIB, and sends the disassembled listing to the file MATHLIB.DIS on drive B.

Errors

DIS68 generates an error message whenever it detects an error in the relocatable object-code file. Since the relocatable object-code file should not have any errors, continuing at this point produces more errors because the sequence is off. To correct error s, recompile the program and be sure you are disassembling Pascal/MT+ code only.

Sample Disassembly

The listings shown below show the source code and the disassembly of a program called PPRIME, which counts prime number**s.**

Listing 5-1 PPRIME.PAS

```
PROGRAM
pprime; (* Uses sieve of Eratosthenes *)

CONST
   size = 8190;

VAR
   flags    :ARRAY[0 .. size] OF BOOLEAN;
   i,k   :INTEGER;
   prime    :INTEGER;
   count    :INTEGER;
   iteration   :INTEGER;


BEGIN (* Main Program *)
   count:= 0;
   writeln( 'Do 10 iterations' );

   FOR iteration := 1 TO 10 DO
      BEGIN
         count := 0;
         FILLCHAR(flags,SIZEOF(flags), CHR(TRUE));
         FOR i := 0 TO size DO
            IF flags[i] THEN
               BEGIN
                  prime := i + i + 3;
                  k := i + prime;
                  WHILE k <= size DO
                     BEGIN
                        flags[k]: = FALSE;
                        k := k + prime;
                     END;
                     count := count + 1;
               END
      END;
      WRITELN(count,' Primes');
END. (* Main Program *)
```

Listing 5-2 PPRIME.DIS

```
* Address    Opcode  Mneumonic Operands

00000000    .globl   _win     * external
00000000    .globl   _crl     * external
00000000    .globl   _sfb     * external
00000000    .globl   _ini     * external
00000000    .globl   _wrs     * external
00000000    .globl   _hlt     * external
00000000    .globl   flags         * external
00004000    .globl   1        * external
00003FFE    .globl   k        * external
00004002    .globl   prime         * external
00004006    .globl   iteratio      * external
00004004    .globl   count         * external
00000000    .globl   output        * external
00000000    .globl   fillchar * external

00000000 6000  bra   0004
00000004 4EB9  jsr   _ini

*  1  0  PROGRAM  pprime; (* Uses sieve of Eratosthenes *)
*  2  0
*  3  0  CONST
*  4  1     size = 8190;
*  5  1
*  6  1  VAR
*  7  1     flags    :ARRAY[0 .. size] OF BOOLEAN;
*  8  1     i,k   :INTEGER;
*  9  1     prime    :INTEGER;
*  10 1     count    :INTEGER;
*  11 1     iteration   :INTEGER;
*  12 1
*  13 1  BEGIN (* Main Program *)
*  14 1     count:= 0;

0000000A 33FC     move.w      #$0000,count

*  15 1     writeln( 'Do 10 iterations' );
```

6. Section

Program Structure at Runtime

Command File Structure

LINK68 creates a command file in the standard CP/M-68K format. Each command file has a 28-byte header. The header contains the size and starting address for each of the following components in the command file:

- A Text (code) segment containing the program's instructions.
- A Data segment containing the program's initialized data such as arithmetic and string constants.
- A Block storage segment (bss) for any uninitialized data generated by the program when it runs. This space is not allocated until the operating system load the command file.
- An optional symbol table that defines any symbols referenced by the program.
- Optional relocation information that specifies the relative relocation of each word within each program segment, if required.

At the beginning of each module is a **jump** table that contains jumps to each procedure or function in the module. The main module also has a jump to the beginning of the code (first instruction).

Run-time Memory Map

Figure 6-1 shows the memory layout at run-time after CP/M-68K loads a Pascal/MT+ program into the Transient Program Area (TPA).

An area reserved for stack space is immediately below the operating system. First, there is a lexical stack used by the run-time system to keep track of the lexical level of procedure blocks. Below the lexical stack is the user stack (see Figure 6-1).

The default size for the lexical stack is 512 bytes, and the default size of the user stack is 1024 bytes. You can change both values by altering the file named **CPMINI,** which is included on the distribution disks.

Free memory is the area from the end of the bss segment to the top of available memory. The heap grows upward from the low end of free memory, and the user stack grows downward from the high end of available memory.

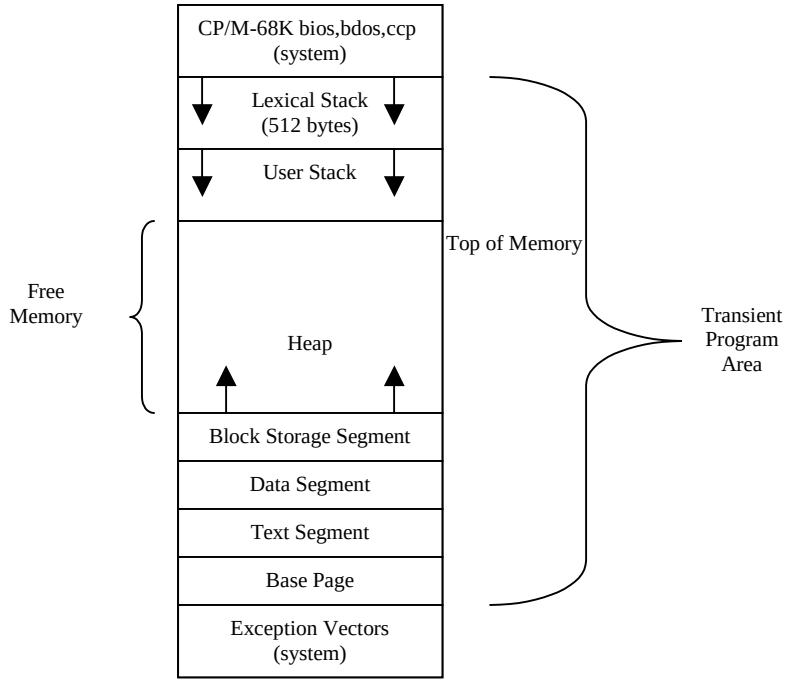


Figure 6-1 Memory Layout in Transient Program Area

Stack

The compiler always generates recursive code. Return addresses and local variables for all procedures are stored on the user stack. If recursion is deeply nested and the default stack size is too small, the program can overwrite the top of memory. Generally this is not a problem unless the heap is also very large. In this case, if recursion continues or the heap continues to grow, it is possible for the user stack to overwrite the heap or vice versa.

Note: The run-time system does not perform any checks on memory allocation bounds. If the user stack overwrites the heap, the program halts with unpredictable results.

Heap

The heap is the area of free memory from which storage for certain variables is dynamically allocated and deallocated at run-time. Refer to Section 9 for more in formation about managing the heap.

End of Section 6

7. Section

Writing Segmented Programs

One major advantage of Pascal/MT+ is the ability to write a large, complex program as a series of small, independent modules. You can code, test, debug, and maintain each module separately, and thereby greatly simplify the overall task of program design. The process of breaking a program into separate units is called segmenting.

Pascal/MT+ provides three methods for segmenting programs: modules, overlays, and chaining.

- Modules are separately compiled program sections. You can link modules together to build overlays, libraries, or entire programs.
- Overlays are sets of modules, linked together as a unit, that load into memory from disk when a procedure or function in one of the modules is referenced from somewhere else in the program. Overlays need to be in memory only when a routine in the overlay is called; otherwise, they remain on the disk. Overlays have hexadecimal filetypes, for example, PROG.01F .
- Chaining allows one program to call another and leave data in memory that can be shared by the new program.

You can use these three features in any combination to produce modular programs that are easier to maintain and take up less memory than monolithic programs.

If you are not an experienced Pascal/MT+ programmer, you should start by writing programs without overlays.

Modules

The Pascal/MT+ system lets you do modular programming with little preliminary planning. You can develop programs until they become too large to compile and then split them into modules. The **E** compiler source-code option lets you make variables, functions, and procedures local to the module.

There are two main differences between modules and programs:

- A module must contain at least one procedure or function. However, a module does not have a main body of statements other than those contained in procedures and functions.
- In a module, the reserved words MODULE and MODEND replace the reserved words PROGRAM and END.

The following example shows a typical module.

```
MODULE sample_mod;

VAR
   mainfile : EXTERNAL TEXT;

PROCEDURE echo (st: STRING; times: INTEGER);

VAR
   i  : INTEGER BEGIN
   FOR i:=” 1 TO times DO
      WRITELN (mainfile,st)
   END;

MODEND.
```

Modules can have free access to global variables, functions, and procedures in any other module. If you want to keep variables, functions, and procedures local to a module, use the E- compiler source-code option.

Use the reserved word EXTERNAL to declare variables, functions, and procedures that are allocated in other modules or in the main program. EXTERNAL tells the compiler not to allocate space in the module. You can declare externals only at the global (outermost) level of a module or program.

For variables, put the reserved word EXTERNAL between the colon and the type in a global declaration. For example,

```
VAR
   i,j,k    : EXTERNAL INTEGER; (* in another module *)
   r  : EXTERNAL RECORD ; (* in another module *)
   x,y   : INTEGER;
   st    : STRING;
END;
```

For procedures and functions declared in other modules, put the reserved word EXTERNAL before the word FUNCTION or PROCEDURE. These external declarations must come before the first normal procedure or function declaration in the module or program. External routines cannot have procedures and functions as parameters.

Note: The compiler does not type check declarations between modules. Therefore, ensure that the number and type of parameters match the declarations in the module where the space is allocated. For functions, the type of the returned value must match.

In Pascal/MT+, external names are significant to seven characters only. Internal names are significant to eight characters.

Listing 7-1 shows the outline of a main program, and Listing 7-2 shows the outline of a module. The main program references variables and subprograms in the module; the module references variables and subprograms in the main program.

Listing 7-1 Main Program Example

```
PROGRAM external_demo;

(* label, constant and type declarations go here *)

VAR
i,j : INTEGER; (* available in other modules *) k,l : EXTERNAL INTEGER;
(* located elsewhere *)

EXTERNAL PROCEDURE sort(VAR q : list; len :INTEGER); EXTERNAL FUNCTION
iotest : INTEGER;

PROCEDURE procl;

 BEGIN
   IF iotest =1 THEN(* normal external function call *)
   .
   .
   .

   END;

   BEGIN
    sort(...) (* normal external procedure call *)
   END.
```

Listing 7-2 Module Example

```
MODULE module_demo;

(* label, constant and type declarations go here *)

VAR
   i,j EXTERNAL INTEGER; (* use those from main program *)
   k,l   INTEGER; (* define these here *)

EXTERNAL PROCEDURE prod; (* from main program *)
   PROCEDURE SORT(...);    (* define sort here *)

   .
   .
   .

FUNCTION iotest INTEGER; (* define iotest here *)
   .
   .
   .

(* maybe other procedures and functions here *)

MODEND.
```

Overlays

Using overlays, you can link programs so that parts of them automatically load from the disk as they are needed. Thus, a whole program does not have to fit in memory simultaneously. You can use overlays to store infrequently used modules and module groups that need not be co-resident.

Terminology

The following terms are used when describing overlays:

- Root module: the portion of the program that is always in memory. Root modules have the 68K filetype. A root module consists of a main program, the run-time routines it requires, and, optionally, the run-time routines that the overlays require.
- Overlay area: an area of memory where the overlay manager loads overlays. Plan the location and size of the overlay areas and specify them at link-time.
- Overlay static variables: global variables, or variables local to a run-time or assembly-language routine in the overlay. All Pascal/MT+ modules are recursive. Recursion reduces the amount of static data. It does not necessarily eliminate it because run-time code linked with the overlay might contain static data. When you link the overlay, the linker determines the amount of space required for static variables.

General Overlay Scheme

LINK68 supports **a** simple tree-structured overlay scheme with a maximum of 255 overlays. You can create overlays to a depth of five levels below the root module. Only one overlay on a given level can be memory-resident at a time. LINK68 places all global static data in the root module, no matter where it is originally defined.

An overlay can reference any symbol in another overlay that is one level above it in the tree, or in an overlay on any level below. An overlay cannot reference any symbol in an overlay on the same level or

in an overlay that is more than one level above itself.

Figure 7-1 shows a typical overlay scheme. In this scheme, overlays A and B can both reference symbols in the root, but overlay A cannot reference symbols in B because they cannot be coresident. Overlays B and C can reference symbols in each other and the root, but not in overlay A.

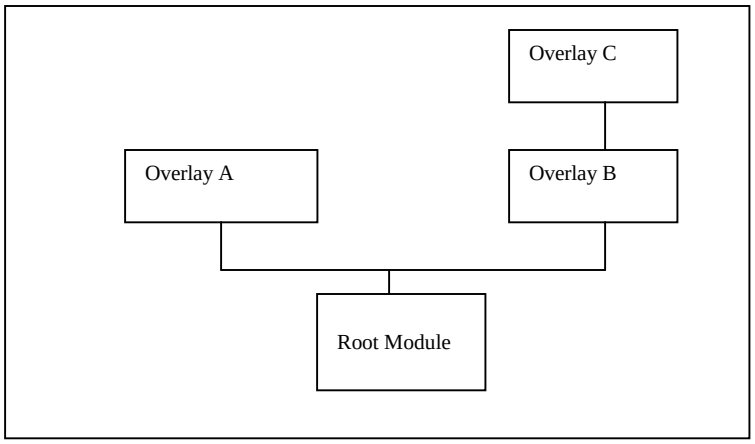


Figure 7-1 Typical LINK68 Overlay Sheme

Overlay File Format

An overlay file has the same format as a regular 68K command file. The first word in the header is always 6OlAH. An overlay file can be either absolute or relocatable. An overlay file can have any filetype. The default filetype is 068.

If you use the SYMBOLS option, LINK68 places the overlay's symbols in the root module.

If you use the common directive in AS68 to specify a common area shared by separate overlay modules, LINK68 resolves all common areas with the same name to the same address in the root module's bss segment. If more than one overlay file specifies static storage with the same name, LINK68 uses the largest size for allocation.

The bss size for the root module is set to contain the area into which the overlays are loaded. The symbol _end is resolved to the top of the overlay area.

Linking Overlays

You determine a specific overlay scheme by the manner in which you link the programs. That is, overlays do not require any special construct or syntax in the source code. However, you must ensure that the root module contains the overlay manager and loader.

The general form of the command line for linking overlays is

LINK68 <root>, <ovlmgr>,(<overlay-l>[, <overlay-2>[, .. <overlay-n>]])

where <ovlmgr> is the overlay manager in the run—time system and <overlay—i> through <overlay—n> are the overlay modules. The overlay specifications are always last in the command line.

For example, the following command creates the overlay scheme shown in Figure 7-2:

```
A>link68 myfile =parta,ovlmgr,(partb1,partb2)
```

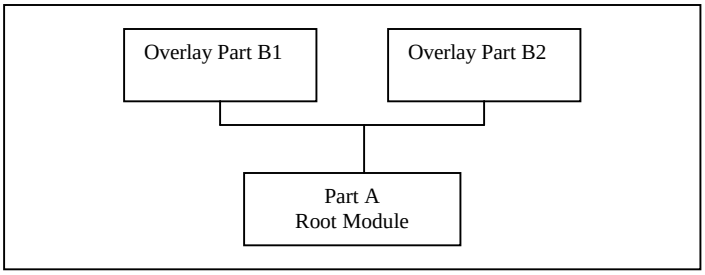


Figure 7-2 Overlay Scheme Example 1

You can nest overlays by nesting the enclosing parentheses in the command line. For example, the following command creates the overlay scheme shown in Figure 7—3:

```
A>link68 myfile =parta,ovlmgr,(partbl, (partb2))
```

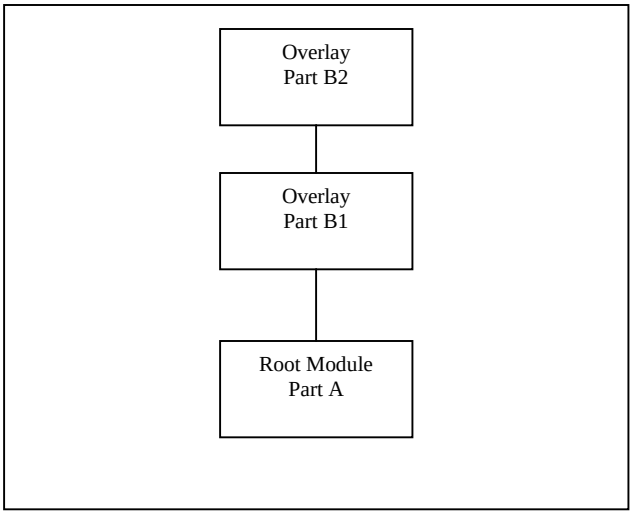


Figure 7-3 Overlay Scheme Example 2

Chaining

Chaining allows one program to call another program into memory and transfer control to that program. Chaining is an implementation—dependent feature that is not be available on all implementations of Pascal/MT+ (see Appendix E, "Writing Portable Programs").

To chain to another program, you execute a call to the CHAIN procedure, and pass the name of the file variable as a single string parameter.

Sharing Data

There are two ways that chained programs can share data: shared global variables and absolute variables.

Using the shared global variable method, you must guarantee that at least the first section of global variables is the communication area. The remainder of the global variables do not need to be the same in each program. You must also declare the shared variables identically in each of the chained

programs so that they have the same location and size.

Using the absolute variable method, you typically define a record that is used as a communication area; then place the record at the same absolute location in each module.

Maintaining the Heap

No special facilities are needed to maintain the heap across the chain. However, files cannot remain open across a chain. If you want to leave something open, you must use overlays, not chaining.

Listings 7—3a and 7—3b list two example programs that communicate with each other using absolute variables. The first program chains to the second program, which prints the results of the first program's execution.

Listing 7-3 Chain Demonstration Program 1

```
PROGRAM chain_1; (* Program #1 in chain demonstration *) TYPE
   comm_area =    RECORD
                  i,j,k    INTEGER
               END;
VAR
   globals ABSOLUTE [$8OOO] comm_area;
   (* this address is arbitrary; *)
   (* it may not work on your system *)
   chain_file FILE;
   title string;

BEGIN (* Main program #1 *)
title := 'F:CHAIN2.68K';
WITH globals DO
   BEGIN
      i:= 3;
      j:= 3;
      k:= i * j;
   END;
IF IORESULT = 255 THEN
      BEGIN
         WRITELN('Unable to open CHAIN2.68K');
         EXIT
      END;
   CHAIN(chain file)
END. (* End chain_1 *)
```

Listing 7-3b. chain Demonstration Program 2

```
PROGRAM chain_2; (* Program #2 in chain demonstration *)

TYPE
   comm_area = RECORD
   i,j,k INTEGER
END;
VAR
   globals ABSOLUTE [$8000] comm_area;

BEGIN (* Main program #2 *)
   WITH globals DO
      WRITELN('Result of ',i,' times ',j, 'is =', k)

END. (* End chain_2; return to operating system *)
```

8. Section

Interfacing Pascal/MT + with Assembly Language Routines

This section describes the conventions for interfacing Pascal/MT+ programs with code written in assembly language.

Interface Conventions

Both the AS68 assembler and the Pascal/MT+ compiler generate entry-point and external-reference records in the same relocatable object-code format. These records contain external symbol names. The relocatable object-code format allows up to seven characters in an external name.

To access assembly-language variables or routines from a Pascal/MT+ program, you must follow these conventions:

- Declare them .globl in the Data segment of the assembly-language module.
- Declare them EXTERNAL in the Pascal/MT+ program.

To access Pascal/MT+ global variables and routines from an assembly—language routine, you must perform the following steps:

- Declare the name .globl in the Data segment of the assembly-language program.
- Declare the variable or routine at the global level in the Pascal/MT+ program.
- Compile the program using the E+ source-code option to generate entry-point records.

The following example shows how an assembly-language module references a variable that is declared in a Pascal/MT+ module.

```
(* program test.s *)

.globl pqr (* external variable from pascal program *)
.text

test:
   .
   .
   move.w   pqr,d7   (*get contents of pascal integer *)
   .
   .

end

(* Pascal program fragment *)

VAR (* in globals *)
   PQR   INTEGER; (* accessible by as68 routine *)
```

Parameter Passing

When you call an assembly-language routine from Pascal/MT+ or a Pascal/MT+ routine from assembly language, parameters pass on the stack.

On entry to the routine, the top of the stack is a double word containing the return address. The parameters are below the return address, in reverse order from declaration.

Each parameter requires at least one 16-bit word of stack space. A character or Boolean passes as a 16-bit word with a high-order byte of 00. VAR parameters pass by address.

Address operands and pointers use two words of stack space. The address represents the byte of the actual variable with the lowest memory address.

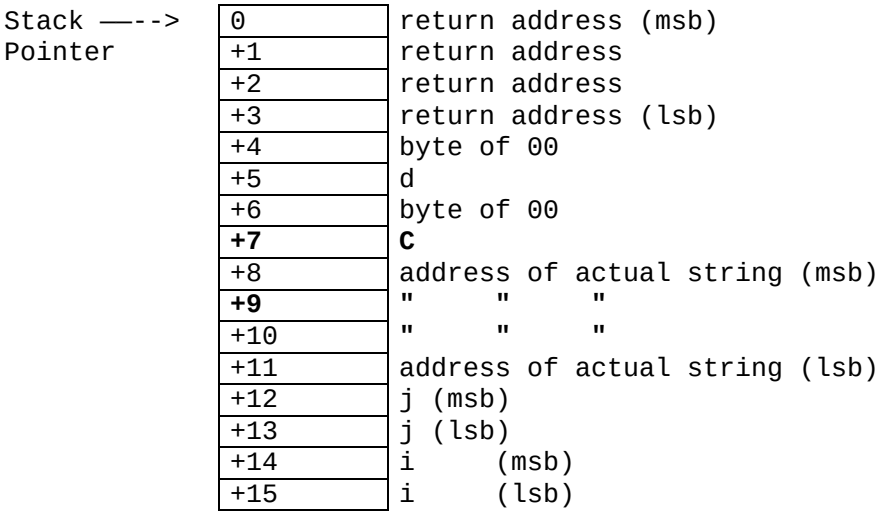
Nonscalar parameters, except sets, always pass by address. If the parameter is a value parameter, the compiler generates code that calls _MVL to move the data.

The _SS2 routine handles set parameters. If passed by value, the actual value of the set goes on the stack. Sets are stored on the stack with the least significant byte on top and the most significant byte on bottom.

Figure 8-1 shows how a typical parameter list appears on the stack on entry to a procedure. If the procedure is declared as

```
PROCEDURE demo(i,j INTEGER; VAR q STRING; c,d CHAR);
```

then the stack as appears as shown below.



lsb = least significant byte
msb = most significant byte

Figure 8-1 Stack Containing a Parameter List

The assembly-language program must remove all parameters from the stack before returning to the calling routine.

Nonreal function values return in the D7 register. Real values return on the stack. They are placed below the return address before the function returns. Therefore, they remain on the top of the stack when the calling program reenters after the return.

Assembly-language functions can return the simple types BOOLEAN, CHAR, INTEGER, LONGINT, or REAL. Assembly-language functions can also return pointers and enumerated types, but cannot return the structured types STRING, RECORD, or arrays.

Interface Example

Listings 8-1 and 8-2 illustrate the interface between a Pascal/MT+ program and two assembly—language routines.

The Pascal program performs the PEEK and POKE functions found in BASIC. The assembly-language module simulates the PEEK and POKE. PEEK returns the byte found at the address passed to it, and POKE puts the bytes in the specified address.

Listing 8-1 Pascal/MT+ PEEK_POKE Program

```
PROGRAM peek_poke;

TYPE
   byte_ptr = ^BYTE;

   pointerkludge = RECORD
                   CASE BOOLEAN OF
                      TRUE  (p : byte_ptr);
                      FALSE:      (q : LONGINT)
               END;
VAR
   choice INTEGER;
   bbb BYTE;
   ppp pointerkludge;

EXTERNAL PROCEDURE poke(b : BYTE; p byte-ptr); EXTERNAL FUNCTION peek(p
: byte_ptr) : BYTE;

BEGIN (* Main Program *)
 REPEAT

   WRITE( 'Which address?');
   READLN(ppp.q);
   WRITE('l) Peek   2) Poke   3) Exit');
   READLN(choice);
   IF choice = 1 THEN
      WRITELN(ppp.q, ' contains ',peek(ppp.p))
   ELSE

      IF choice =2 THEN
         BEGIN
            WRITE('Enter byte of data: ');
            READLN(bbb);
            POKE(bbb, ppp.p)
         END
  UNTIL  choice   = 3
END. (* Main Program *)
```

Listing 8-2. Assembly-Language PEEK and POKE Routines

```
*  PEEK and POKE Routines
*

*

.globl   peek  * Entry point for peek routine
.globl   poke  * Entry point for poke routine
.text       * Tell assembler we are writing code
*

*

*  PEEK — Address to peek is on stack. Return result in D7.
*

*

peek:
   move.l   (a7)+,a0 * pop return address
   move.l   (a7)+,al * pop address to peek
   moveq #0,d7    * clear function return
   move.b   (al),d7        * get byte from memory
   jmp   (aQ)        * return
*

*

*  POKE - Byte to poke is on top of stack as the lower byte *   of a
word.
*  Address to poke follows on stack.
*

*

poke:
   move.l   (a7)+,a0 * pop return address
   move.w   (a7)+,d7       * byte to store
   move.l (a7)+,al   * address to poke
   move.b d7,(al)    * poke byte
   jmp (aO)          * return
*

   .end
```

9. Section

Controlling the Run-time Environment

This section describes several Pascal/MT+ features that let you control your program's run—time environment. The features provide the ability to

- manage the heap as a standard heap, or as a stack
- access the operating system through direct function calls
- insert machine code into the Pascal/MT+ source code using INLINE
- declare variables with absolute addresses
- address the processor's I/O ports
- perform range and error checking

Heap Management

Pascal/MT+ supports two alternative methods for managing the heap: as a standard heap `or as a stack.`

Using the FULLHEAP Routines

You can manage the heap using the ISO—standard routines NEW and DISIOSE as they are implemented in the library named FULLHEAP. When you use the FULLHEAP routines,

- NEW assumes a standard heap and dynamically allocates data to the smallest space that can hold the requested item.
- DISPOSE frees the memory allocated to the requested item.

Using the PASLIB Routines

You can also manage the heap using the NEW and DISPOSE routines as they are implemented in the PASLIB run-time library. When you use the PASLIB routines,

- NEW treats the heap area as an ordinary stack. NEW puts the dynamic data on top of the stack which grows upward from the end of the bss segment.
- DISPOSE performs no function, but is included in the symbol-table.
- You can simulate UCSD Pascal's MARK and RELEASE routines by using the built-in routines MRK and RLS, as shown in this example:

```
MODULE ucsd_heap;

EXTERNAL FUNCTION MRK : LONGINT;
EXTERNAL FUNCTION _RLS (l : LONGINT);

PROCEDURE mark(VAR p : LONGINT);
BEGIN
   p := _MRK
END;
```

```
PROCEDURE release(p : LONGINT);
BEGIN
_RLS(p)
END;

MODEND.
```

LMAXAVAIL and LMEMAVAIL

You can use the predefined functions LMAXAVAIL and LMEMAVAIL to determine the amount of heap space remaining at any given time. LMAXAVAIL and LMEMAVAIL are not included in PASLIB. You must explicitly declare them in your program as

```
EXTERNAL FUNCTION LMEMAVAIL : LONGINT;
EXTERNAL FUNCTION LMAXAVAIL : LONGINT;
```

When used in conjunction with the NEW and DISPOSE routines in FULLHEAP, LMAXAVAIL returns the size of the largest contiguous block of unallocated free memory. LMEMAVAIL returns the total of all currently unallocated blocks of memory.

When used in conjunction with the `NEW` and DISPOSE routines in PASLIB, LMAXAVAIL and LMEMAVAIL return identical values.

You should always use LMAXAVAIL and LMEMAVAIL instead of the standard PASLIB routines MEMAVAIL and MAXAVAIL, which return true values only if the total amount of heap space is less than 32,767 bytes. If the heap space is greater than 32,767 bytes, both MEMAVAIL and MAXAVAIL return 7FFFH.

_HERR

HERR (Heap Error) is a predefined BOOLEAN variable used by NEW to return the result of an allocation request. Always use HERR in conjunction with NEW, because the heap management system does not signal an error if there is no space available when you make an allocation request.

Direct Operating System Access

You can make direct function calls to the CP/M—68K operating system by using the BDOS routine which you declare in a Pascal/MT+ program as follows:

```
EXTERNAL FUNCTION_BDOS (<func>:INTEGER;<parm>PTR):INTEGER;
```

<func> is the BDOS function number. Refer to your operating system documentation for the list of functions. <parm> is a generic pointer. You can use the ADDR function to generate the value.

The following example demonstrates direct access to the operating system in a function definition. The function KEYPRESSED returns TRUE if a key is pressed, and FALSE if not.

```
FUNCTION keypressed : BOOLEAN;

BEGIN
   keypressed := (_BDOS (ll,ADDR(keypressed)) <> 0)
END;
```

Listings 9-1 and 9-2 illustrate calls to BDOS Functions 6 and 23 respectively.

Listing 9-1 Calling _BDOS Function 6

```
PROGRAM BDOS_6; (* Use BDOS Function 6 for console I/o *)

(* Since the BDOS call requires a pointer parameter *)
(* we must define a record that allows us to pass an *)
(* INTEGER as a pointer type. In the record, the *)
(* FALSE tagfield occupies the same memory as the two *)
(* INTEGERS in the TRUE tagfield.

TYPE
   ptr = ^CHAR;

VAR
   i  :INTEGER;
   ch :CHAR;
   pchar :ptr;

   EXTERNAL FUNCTION _BDOS(func INTEGER; parm ptr) INTEGER;

(* The main program echoes any input character *)
(* at the console until you input a colon *)

BEGIN (* Main Program *)
 new(pchar);
 REPEAT
   pchar^ :=   chr(255);

   REPEAT (* Read a character *)
      ch := CHR(_BDOS(6,pchar));
   UNTIL ch <> CHR(0);
   IF ch <> ':' THEN
      BEGIN (* convert ch to INTEGER, pass as a pointer *)
         pchar^ := ch;
         i:=   BDOS(6,pchar); (* Write a character *)
      END;
 UNTIL ch = ':'
END. (* Main Program *)
```

Listing 9-2 Calling BDOS Function

```
PROGRAM BDOS_23;(* Use BDOS Function 23 to rename files *)

TYPE
   ptr^ = INTEGER;
   fcblk = PACKED ARRAY [0. .36] OF CHAR;

VAR
   oldname,newname   :STRING;
   fl,f2             : fcblk;
   i              :INTEGER;

EXTERNAL FUNCTION _BDOS(func INTEGER; parm : ptr) INTEGER; EXTERNAL
PROCEDURE _PARSE(VAR f : fcblk; S STRING);

(* _PARSE converts a string into internal *)
(* CP/M filename format *)

BEGIN (* Main Program *)
   WRITE('Enter old filename:' );(* Get the old filename *)
   READLN(oldname);
   _PARSE( fl, oldname);

   WRITE('Enter new filename:' );(* Get the new filename *)
   READLN(newname);
   _PARSE ( f2, newname);

(* Create the FCB required by BDOS call 23 *)

   MOVE( f2, f1[16] ,12);

(* Now call the rename function passing pointer to FCB *)
(* containing the old and new filenames *)

   IF BDOS(23,ADDR(fl)) = 255 THEN
      WRITELN('Rename failed. ',oldname, ' not found.')
   ELSE
      WRITELN('File ' ,oldname, 'renamed to ' , newname);

END. (* Main Program *)
```

INLINE

INLINE is a built-in feature that lets you insert code or data in the middle of a Pascal/MT+ procedure or function. You can insert small machine-code sequences and constant tables into a Pascal/MT+ program without using externally-assembled routines.

The syntax for INLINE has the form

```
INLINE( <argument> {/<argument>/... <argument>})
```

where <argument> must be either a constant or a variable reference that evaluates to a constant. <argument> can be of type BOOLEAN, CHAR, INTEGER, LONGINT, REAL, or STRING.

Note that a string in single apostrophes does not generate a length byte, but simply the data for the string.

Variables evaluate to a long address. All jumps are relative to the current position in the code segment.

Literal constants of type integer are allocated one byte if the value falls in the range 0 to 255. Named and declared integer constants always get two bytes.

Listing 9—3 demonstrates how you can use INLINE to construct compile-time tables. Note that the ADDR of TABLE must be added to its offset. This is because ADDR does not give the address of TABLE, due to additional code that recursion management produces. An extra eight bytes of code is generated.

Note also that the procedure TABLE must be in the same module as the statement that takes the ADDR of TABLE.

Listing 9-3 Using INLINE to Construct Compile-time Tables

```
PROGRAM demo_inline;

CONST
   element = 3; {Third array element}

TYPE
   Id_field = ARRAY [1. .8] OF CHAR;
   Id_ptr = ^id_field;
   pointerkludge = RECORD
                   CASE BOOLEAN OF
                   TRUE :(p : id_ptr);
                   FALSE: (l : longint);
                END;
VAR
   table_ptr : id_ptr;
   p : pointerkludge;
   offset : integer;

PROCEDURE table;

BEGIN
   INLINE( 'Digital ' /'Research' /'Software');
END;

BEGIN (* Main Program *)

p.p := ADDR(table);
p.l := p.l + #14;
offset := sizeof(table_ptr^) * (element – 1);
p.l := p.l + long(offset);
table_ptr := p.p;
WRITELN(table_ptr^); (* Should write 'Software' *)

END. (* Main Program *)
```

Absolute Variables

You can declare a variable with an absolute address if you know the address at compile time. The syntax for declaring ABSOLUTE variables is

```
<variable name> : ABSOLUTE [<address>]
```

The following examples are valid declarations of ABSOLUTE variables:

```
int : ABSOLUTE [$8000] INTEGER;
```

```
screen: ABSOLUTE [$CO] ARRAY[0..15, 0..63] OF CHAR;
```

The compiler does not allocate space in your Data segment for absolute variables. Ensure that no compiler—allocated `variables conflict with the ABSOLUTE variables.`

Manipulating I/O Ports

Pascal/MT+ supports direct manipulation of the processor s input and output ports through two features:

- INP and OUT
- INPORT-W and OUTPORT_W

INP and OUT

INP and OUT are two predeclared arrays of type BYTE that can be subscripted with INTEGER port number constants in the range 0 to 255. The syntax is

```
<variable> := INP[<INTEGER constant>]
OUT[<INTEGER constant>] := <variable>
```

OUT can be used only on the left side of an assignment statement. If it is not convenient to use a literal constant, you can put the values you want to send out in a CASE statement. For example,

```
CASE <expression> OF

   $Nl : OUT[$Nl] :=<value 1>
   $N2 : OUT[$N2] :=<value 2>
   $N3 : OUT[$N3] :=<value 3>
```

where $Nl,$N2, etc. are literal constants.

If you assign values from INP to an INTEGER type, use the following construct to zero the high-order byte:

```
<variable> :=(INP[$N] & $FF)
```

The following examples illustrate INP and OUT:

```
OUT[0] :=$88;
j := INP[portnum];
```

INPORT_W and OUTPORT_W

You can also manipulate the processor's I/O ports using the function INPORT_W and the procedure OUTPRT_W. Although they are present in PASLIB, you must explicitly declare them as follows:

```
EXTERNAL FUNCTION INPORT_W(<portnumber>:INTEGER):WORD; EXTERNAL PROCEDURE
OUTPRT_W(<portnumber>:INTEGER;data:WORD);
```

The following examples illustrate INPORT W and OUTPORT W:

```
inchar := INPORT_W(portnum);
OUTPRT_W(portnum,outchar);
OUTPRT_W($004F, outchar);
```

Range and Error Checking

The Pascal/MT+ system supports two types of run-time checking: range and error (exception)

checking. By default, the compiler disables range checking and enables error checking.

Range checking

Range checking monitors array subscripts and subrange assignments. It does not check when you read into a subrange variable.

When range checking is enabled, the compiler generates calls to CHK for each array subscript and subrange assignment. The _CHK routine leaves a Boolean value on the stack and error code number 4 (see "Error Checking" in this section). The compiler generates calls to _ERR after the _CHK call. If an error occurs, _ERR asks you whether it should continue or abort.

When range checking is disabled and an array subscript falls outside the valid range, you get unpredictable results. For subrange assignments, the value truncates at the byte level.

Error checking

By default, the run—time system checks for the following error conditions:

- integers and real numbers divided by 0
- real number under flow and overflow
- string overflow

The run-time error checking routines set internal Boolean flags. At run—time, these flags are loaded onto the stack along with an error code. Then, the predefined routine ERR is called to test the Boolean flag.

If there is no error, the flag is FALSE, so _ERR returns to the compiled code and continues execution. If an error occurs, the flag is TRUE and _ERR takes appropriate action. Table 9-1 summarizes the error codes associated with the _ERR routine.

Table 9-1 ERR Routine Error Codes

Value	Meaning
1	Divide by zero check
2	Heap overflow check (unused)
3	String overflow check (unused)
4	array and subrange check
5	Floating point underflow
6	Floating point overflow

The various error conditions produce the following results:

- For floating—point underflow, _ERR does not print an error message, and the result of the operation is 0.0.
- For floating—point overflow, _ERR prints the error message

```
FLOATING-POINT OVERFLOW
```

  The result of the operation is a large number.

- For division by zero, _ERR prints the error message

```
DIVIDE BY ZERO DETECTED
```

  The result is the representation of the largest—possible number.

- For heap overflow, _ERR takes no action and does not print an error message. You should always test the value of _HERR to detect heap overflow.
- For string overflow, _ERR prints the error message

```
STRING OVERFLOW (TRUNCATED)
```

and the string is truncated.

User-supplied Error Handlers

You can write your own _ERR routine instead of using the one supplied with the system. To use your own version of ERR instead of the one in PASLIB, link your routine ahead of PASLIB to resolve the reference to _ERR.

Declare your version of ERR as follows:

```
PROCEDURE_ERR(<error> : BOOLEAN; <error number> : INTEGER);
```

Your version should check the <error> variable and exit if it is FALSE. If the value is TRUE, decide what action to take. Your version should also use the same values of <error number> listed in Table 9-1.

I/O Error Handling

The run-time routine BDOS does not handle I/O errors. However, it returns the CP/M-68K error code in IORESULT. You can rewrite _BDOS, using the supplied assembly-language source, to make more extensive checks for disk I/O errors.

End of Section 9

10. Section

Writing ROM-based Code

The Pascal/MT+ system can generate code for use with or without an operating system. This section presents some guidelines for writing programs in a ROM-based system.

Note: The guidelines presented here are just a suggestion; Digital Research does not provide detailed application support for ROM-based applications.

Programs That Use I/O

There are three ways you can write a ROM-based program that performs I/O:

- Use redirected I/O for all READ and WRITE statements. This replaces the run-time character I/O routines with user-written I/O routines. Refer to the Language Reference Manual.
- Rewrite the GET routine because the read-integer and read-real routines call it. Also, rewrite the run-time subroutines _RNC (read-next-character) and _WNC (write-next-character).
- If you want the program to run in a totally stand-alone environment, you must write an assembly-language module that simulates the CP/M-68K BDOS in your PROM. This routine can jump around the standard code that simulates the BDOS and can simulate the following BDOS functions:
  - Function 1: Console Input
  - Function 2: Console Output
  - Function 5: List Output

The function number is in the DO register; the data for output is in Dl.

To simulate Function 1, return the data in the DO register. All registers are free to use, and the stack contains nothing but the return address.

Rewriting the _INI Routine

In a ROM-based environment, you might also want to rewrite the INI routine to shorten or eliminate the INPUT and OUTPUT FIB (File Information Block) storage, which is needed for TEXT file I/O compatibility.

Make sure any changes to INPUT and OUTPUT are also handled in RST (read a string from a file) and _CWT (wait for EOLN to be TRUE on a file).

If your program does not do READLN or WRITELN calls and does not use the heap or overlays, you can rewrite the _INI procedure in your program as

```
PROCEDURE _INI;
BEGIN
END;
```

Note: The distribution disks include source-code outlines for the _INI, _RNC, _WNC, and GET routines that you can customize for your ROM-based environment.

Linking Altered Routines

If you alter any of the standard run—time routines to run in a ROM-based environment, remember to link them before PASLIB to resolve the references. For example,

```
A>link68 userproq,mywnc,myrnc,myget,myini,paslib.L68
```

End of Section 10

11. Section

Sample Pascal/MT ± Programs

This section contains sample programs that illustrate various features of Pascal/MT+. The best way to learn any programming language is to study working examples. You should study the programs in this and other sections, and cross check with the material in the Language Reference Manual when necessary. Once you understand the operation of a program, you can modify or enhance it, and thereby gain further experience with Pascal/MT+.

File Transfer

Listing 11-1 shows the main body of a file transfer program. The main program calls one of four different transfer procedures that illustrate different ways to implement such a file transfer.

Listing 11-2 shows the transfer program using the BLOCKREAD and BLOCKWRITE procedures. This program uses untyped files and a large 2K byte buffer to transfer the data.

Note that the program only works for files whose size is an even multiple of 2K bytes. Thus, if the size of the source file is 9K, the last 1K is not written because the variable result is nonzero after the call to BLOCKREAD. Using a 128-byte buffer guarantees that all the data is transferred.

Listing 11-3 shows the transfer program using the GNB and WNB routines for byte-level access to the file.

Listing 11-4 shows the transfer program using the SEEKREAD and SEEKWRITE procedures for performing random access I/O.

Note that IORESULT returns a 1, indicating end-of-file if the source file does not fill the sector, as in BLOCK I/O. In this case, the 2K bytes of window variable for <file a> do not fill the sector, and the last portion of data that does not fill the 2K buffer is never written to the destination file.

Listing 11-5 shows the transfer program using the GET and PUT procedures. This method is slower than the buffered methods.

Comparison Table

Table 11—1 shows a comparison of the code size, data size, and execution speed for each file transfer program. The sizes are in decimal bytes, the speed is in seconds, and the size of the file is 8K bytes. Each program was run on a 10MHz Motorola MC68000 processor with no wait states, using both a dual floppy disk and a hard disk system.

Note: Your system might not produce the same values reflected in Table 11-1. However, the relative size and speed differences should be the same.

Table 11-1 Comparison of I/O Methods

Statistics	Transfer Method			
	BLOCK I/O	GNB/WNB	SEEK I/O	GET/PUT
Compiled Code	678	716	718	666
Compiled Data	2258	2260	4306	214
Total Code	6428	6448	9170	6204
Total Data	4332	4334	6380	2288
Total Size	10760	10782	15550	8942
Speed				
(Floppy Disks)	8.0	10.0	8.0	64.0
(Hard Disk)	2.0	5.0	3.0	12.0

Program Listings

Listing 11-1 Main Program Body for File Transfer Programs

```
BEGIN (* Main Program *)

WRITE('Name of Source File ? ');
READLN(name);
ASSIGN(file_a,name);
RESET(file_a);
IF IORESULT = 255 THEN
 BEGIN
   WRITELN('Sorry, cannot open ',name);
   EXIT
 END;

WRITE('Name of Destination File ? ');
READLN(name);
ASSIGN(file_b,name);
REWRITE(file_b);
IF IORESULT = 255 THEN
 BEGIN
   WRITELN( Sorry, cannot open ,name);
   EXIT
 END;

(* Call specific TRANSFER procedure *)

transfer( file_a, file_b)

END. (* Main Program *)
```

Listing 11-2 File Transfer with BLOCKREAD and BLOCKWRITE

```
PROGRAM file_transfer_1;

 (* Transfer file_a to file_b using BLOCKREAD and BLOCKWRITE *)

CONST
Buffer_size = 2047;

TYPE
paoc = ARRAY[O. .buffer_size] OF CHAR;
fyle = FILE;

VAR
   file_a, file_b : fyle;
   name        : STRING;
   buffer         : paoc;

PROCEDURE transfer(VAR source : fyle; VAR destination : fyle);

VAR
   result,i : INTEGER;
   quit : BOOLEAN;

BEGIN (* Body of TRANSFER procedure *)

   i:=1;

   REPEAT
     BLOCKREAD(source,buffer,result,SIZEOF(buffer), i);
      IF result = 0 THEN
    BLOCKWRITE(destination,buffer,result,SIZEOF(buffer), i)
      ELSE quit:=TRUE;
   UNTIL quit;

   CLOSE(destination, result);
   IF result = 255
      THEN WRITELN('Error closing destination file')

END; (* TRANSFER procedure *)

(* Body of Main Program in Listing l1-l *)
```

Listing 11-3 File Transfer with GNB and WNB

```
PROGRAM file_transfer_2;

(* Transfer file_a to file_b using GNB and WNB     *)


CONST
   buffer size = 2047;

TYPE
   paoc = ARRAY[1. .buffer_size] OF CHAR;
   text file FILE OF paoc;
   char_file = FILE OF CHAR;

VAR
   file_a : text file;
   file_b : char file;
   name : STRING;

PROCEDURE transfer(VAR source: text file;
               VAR destination : char_file);
VAR
   ch : CHAR;
   result : INTEGER;
   stop_it : BOOLEAN;

BEGIN (* Body of TRANSFER procedure *)
   stop it := FALSE;
   WHILE (NOT EOF(source)) AND (NOT stop_it) DO
    BEGIN
      ch := GNB(source);

      IF WNB(destination,ch) THEN
       BEGIN
         WRITELN( 'Error writing character');
         stop_it := TRUE;
       END;

   CLOSE(destination, result);
   IF result = 255 THEN
      WRITELN('Error closing')

END; (* TRANSFER procedure *)

(* Body of Main Program in Listing 11—1 *)
```

Listing 11-4 File Transfer with SEEKREAD and SEEKWRITE

```
PROGRAM file_transfer_3;

(* Transfer file_a to file_b using SEEKREAD and SEEKWRITE *)

CONST
   buffer size = 2047;

TYPE
   paoc = ARRAY[O..buffer size] OF CHAR;
   text_file FILE OF paoc;
   char_file = FILE OF paoc;

VAR
   file_a : text_file;
   file_b : text_file;
   name : STRING;

PROCEDURE transfer( VAR source: text_file;
                VAR destination : text_file);
VAR
  result,i : INTEGER;
  stop_it : BOOLEAN;
  ch : CHAR;

BEGIN (* Body of TRANSFER procedure *)
 ch :=   'A';
 result := 0;
 i := 0;

WHILE result <> 1 DO
 BEGIN
   SEEKREAD( source, i);
   result := IORESULT;
   IF result = 0 THEN
      BEGIN
         destination^ := source^;
         SEEKWRITE(destination,i);
      END;
   i := i +1;
 END;

 CLOSE(destination, result);
 IF result = 255 THEN
   WRITELN('Error closing destination file')

END; (* TRANSFER procedure *)

(* Body of Main Program in Listing 11-1 *)
```

Listing 11-5 File Transfer with GET and PUT

```
PROGRAM file_transfer_4;

(* Transfer file_a to file_b using GET and PUT *)

TYPE
   char_file = FILE OF CHAR;

VAR
   file_a, file_b : char file;
   name : STRING;

PROCEDURE transfer(VAR source: char_file;
               VAR destination : char_file);

VAR
   result   : INTEGER;

BEGIN (* Body of TRANSFER procedure *)
   WHILE NOT EOF(source) DO
    BEGIN
      destination^ = source^
      PUT(destination);
      GET(source);
    END;

   CLOSE(destination, result);
   IF result = 255 THEN
      WRITELN('Error closing destination file')

END; (* TRANSFER procedure *)

(* Body of Main Program in Listing 11-1 *)
```

A. Appendix

Compilation and Run-time Error Messages

This appendix contains a list of the error messages output by the compiler and run—time system. The compilation errors have the same numbering sequence as described in the Pascal User Manual and Report, second edition, by Kathleen Jensen and Niklaus Wirth (New York: Springer—Verlag, 1978).

In most cases, the error description is self—explanatory and the user response is obvious. In certain cases where the error can occur in more than one context, suggested user responses are given. In each case, you must correct the error and recompile the program.

Compilation Errors

Table A—l. Compiler Error Messages

Message Meaning
ERROR # 3 ‘PROGRAM’ EXPECTED The compiler expects the reserved word ‘PROGRAM’ in this context.
ERROR # 5 ‘:‘ EXPECTED The compiler expects the token 1:1 in this context. This error can be caused by using an equal sign (~) in a VAR declaration.
ERROR # 6 ILLEGAL SYMBOL (POSSIBLY MISSING ‘;‘ ON LINE ABOVE) The compiler does not allow the symbol in this context.
ERROR # 11 ‘[’ EXPECTED The compiler expects the token ‘[‘ in this context.
ERROR # 15 INTEGER EXPECTED The compiler expects an integer value in this context.
ERROR # 16 ‘=’ EXPECTED The compiler expects the token in this context. This error can be caused by using a colon (:) in a TYPE or CONST declaration.

ERROR # 17
'BEGIN' EXPECTED

The compiler expects the reserved word 'BEGIN' in this context.

ERROR # 18
ERROR IN DECLARATION PART

The compiler encountered an error in the declaration. This error can be caused by an illegal backward reference to a type in a pointer declaration.

ERROR # 50
ERROR IN CONSTANT

The compiler encountered a syntax error in a literal constant. This error can occur when using recursion, or improperly using INP and
OUT.

ERROR # 55
'TO' OR 'DOWNTO' EXPECTED IN FOR STATEMENT

The compiler expects the reserved word 'TO' or 'DOWNTO' in this context.

ERROR # 58
ERROR IN <FACTOR> (BAD EXPRESSION)

The compiler encountered a syntax error in the expression.

ERROR # 101
IDENTIFIER DECLARED TWICE

The compiler encountered an identifier that is already declared.

ERROR # 102
LOW BOUND EXCEEDS HIGH BOUND

For subrange types, the low bound must be less than or equal to the high bound.

ERROR # 103
IDENTIFIER IS NOT OF THE APPROPRIATE CLASS

The compiler encountered a variable name used as a type, or a type used as a variable name.

ERROR # 104
UNDECLARED IDENTIFIER

The compiler encountered an identifier that has not been declared.

ERROR # 105 SIGN NOT ALLOWED

Signs are not allowed on non-INTEGER or non-REAL constants.

ERROR # 106
NUMBER EXPECTED

The compiler expects a number in this context. This error can occur as the compiler checks for numbers in an expression after all other possibilities have been exhausted.

ERROR # 107
INCOMPATIBLE SUBRANGE TYPES

Types must be compatible for subrange comparison and assignment. For example, 'A' .. 'Z' is not compatible with 0. .9.

ERROR # 108
FILE NOT ALLOWED HERE

Comparison and assignment of FILE types is not allowed.

ERROR # 109
TYPE MUST NOT BE REAL

The compiler does not allow the type REAL in this context.

ERROR # 110
<TAGFIELD> TYPE MUST BE SCALAR OR SUBRANGE

The tagfield in a CASE—variant record must be a scalar or subrange type.

ERROR # 1ll
INCOMPATIBLE WITH <TAGFIELD> PART

The type of the selector in a CASE—variant record is not compatible with the type of the tagfield.

ERROR # 113
INDEX TYPE MUST BE A SCALAR OR A SUBRANGE

The type of an array index must be declared as a scalar or subrange.

ERROR # 115
BASE TYPE MUST BE A SCALAR OR A SUBRANGE

The base type of a set must be a declared as a scalar or subrange.

ERROR # 116
ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER

There is an error in the type of a variant when using NEW or DISPOSE.

ERROR # 117
UNSATISFIED FORWARD REFERENCE

A forwardly declared pointer was never defined.

ERROR # 119
FORWARD DECLARED PROCEDURE CANNOT RESPECIFY PARAMETERS

Self-explanatory.

ERROR # 120
FUNCTION RESULT TYPE MUST BE A SCALAR, SUBRANGE, OR POINTER

The function is declared with a return value of some nonscalar type such as STRING. This is not allowed in Pascal/MT+.

ERROR # 121
FILE VALUE PARAMETER NOT ALLOWED

FILE types must be passed as VAR parameters.

ERROR # 122
FORWARD DECLARED FUNCTION CANNOT RESPECIFY RESULT TYPE

Self-explanatory

ERROR # 125
ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER

The compiler encountered an error in the type of a parameter to a procedure. This error can be caused by not having the parameters in the proper order for built—in procedures. It can also be caused by attempting to read or write pointers, enumerated types, etc.

ERROR # 126
NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION

The number of parameters passed to the procedure does not match the number specified in the procedure’s declaration.

ERROR # 127
ILLEGAL PARAMETER SUBSTITUTION

The type of a parameter passed to the procedure does not match the corresponding formal parameter in the procedure’s declaration.

ERROR # 129
TYPE CONFLICT OF OPERANDS

The operands in the expression have incompatible types.

ERROR # 130
EXPRESSION IS NOT OF SET TYPE

The context of the expression requires the type SET.

ERROR # 131
TESTS ON EQUALITY ALLOWED ONLY

SET types can only be compared for equality; no other comparison is allowed.

ERROR # 134
ILLEGAL TYPE OF OPERAND(S)

The operands are not valid for this operator.

ERROR # 135
TYPE OF OPERAND MUST BE BOOLEAN

The operands to AND, OR, and NOT must be BOOLEAN.

ERROR # 136
SET ELEMENT TYPE MUST BE SCALAR OR SUBRANGE

An element of a set must be a scalar of subrange type.

ERROR # 137
SET ELEMENT TYPES MUST BE COMPATIBLE

All the elements of a set must be of a compatible type.

ERROR # 138
TYPE OF VARIABLE IS NOT ARRAY

A subscript was specified for a variable that was not declared as ARRAY OF ...

ERROR # 139
INDEX TYPE IS NOT COMPATIBLE WITH THE DECLARATION

The type of the expression that specifies an array subscript is incompatible with the array type.

ERROR # 140
TYPE OF VARIABLE IS NOT RECORD

This error occurs when there is an attempt to access a non-RECORD data structure with the dot operator '.' or the 'WITH' statement.

ERROR # 141
TYPE OF VARIABLE MUST BE FILE OR POINTER

This error occurs when the pointer reference character follows a variable that is not of type pointer or FILE.

ERROR # 143
ILLEGAL TYPE OF LOOP CONTROL VARIABLE

The control variable in an iterative loop can be only be a locally declared, non—REAL scalar value.

ERROR # 144
ILLEGAL TYPE OF EXPRESSION

The expression used as a selector in a CASE statement must be of non—REAL, scalar type.

ERROR # 145
TYPE CONFLICT

The selector in a CASE statement is not the same type as the selecting expression.

ERROR # 147
LABEL TYPE INCOMPATIBLE WITH SELECTING EXPRESSION

The selector in a CASE statement is not the same type as the selecting expression.

ERROR # 148
SUBRANGE BOUNDS MUST BE SCALAR

The lower and upper bounds of a subrange must be scalar types.

ERROR # 149
INDEX TYPE MUST NOT BE INTEGER

An array bound cannot be declared type INTEGER or LONGINT, it must be a subrange type.

ERROR # 151
ASSIGNMENT TO FUNCTION IS NOT ALLOWED

A value cannot be assigned to a function.

ERROR # 152
NO SUCH FIELD IN THIS RECORD

The compiler cannot find the specified field in the record.

ERROR # 155
CONTROL VARIABLE CANNOT BE FORMAL OR NONLOCAL

The control variable in a FOR loop must be locally declared.

ERROR # 156
MULTIDEFINED CASE LABEL

A label in a CASE statement has been defined more than once.

ERROR # 158
NO SUCH VARIANT IN THIS RECORD

The compiler cannot find the specified variant in the record.

ERROR # 159
REAL OR STRING TAGFIELDS NOT ALLOWED

The tagfield in a CASE-variant record must be a scalar or subrange type.

ERROR # 162
PARAMETER SIZE MUST BE CONSTANT

This error occurs when using NEW or DISPOSE with a variant that is not a constant.

ERROR # 165
MULTIDEFINED LABEL

This error occurs when more than one statement is assigned the same label.

ERROR # 168
UNDEFINED LABEL

This error occurs when a declared label was not used to label a statement.

ERROR # 169
ERROR IN BASE SET

The base type of a set must be a scalar or subrange type.

ERROR # 170
VAR PARAMETER EXPECTED

This error occurs when an array is passed as a value parameter.

ERROR # 174
PASCAL FUNCTION OR PROCEDURE EXPECTED

The compiler expects a function or procedure at this lexical level.

ERROR # 183
EXTERNAL DECLARATION NOT ALLOWED AT THIS NESTING LEVEL

This error occurs when an EXTERNAL variable is declared anywhere except at the outermost (global) level.

ERROR #206
IL**1**EGAL REAL NUMBER

The integer part of a REAL constant exceeds the valid range.

ERROR # 250
TOO MANY SCOPES OF NESTED IDENTIFIERS

There is a limit of 15 nesting levels at compile time. This includes WITH and procedure nesting. Simplify the program and recompile.

ERROR # 251
TOO MANY NESTED PROCEDURES OR FUNCTIONS

There is a limit of 15 nesting levels at run-time. This error can also occur when more than 200 routines are in one compiled module. Simplify and recompile.

ERROR # 253
PROCEDURE (OR PROGRAM BODY) TOO LONG

A procedure generated code that overflowed the internal procedure buffer. The limit is 4096 bytes. Reduce the size of the procedure and recompile.

ERROR # 397
TOO MANY FOR OR WITH STATEMENTS IN A PROCEDURE

There is a limit of 16 FOR or WITH statements in a single procedure. Simplify and recompile.

ERROR # 398
IMPLEMENTATION RESTRICTION

Normally used for arrays and sets that are too big to be manipulated or allocated.

ERROR # 407
SYMBOL TABLE OVERFLOW

There is not enough space left in the symbol table. Use the Kn compiler option to eliminate unused entry points, or segment the program into smaller modules.

ERROR # 496
INVALID OPERAND TO INLINE

Usually due to reference that requires address calculation at run—time.

ERROR # 500
NON ISO-STANDARD FEATURE BEING USED

This is a warning only and does not prevent the program from compiling.

ERROR # 998
ERROR IN CONDITIONAL COMPILATION PARAMETER

There is an error in one or more conditional compilation parameters.

ERROR # 999
COMPILER UNABLE TO CONTINUE DUE TO PREVIOUS ERRORS

It is possible for a program to be syntactically correct and still have semantic errors that can confuse the compiler. The compiler stops early with this error number. Look carefully at the line on which the compilation halts. Make some corrections and recompile.

Run-time Errors

Table A—2 lists the error messages reported by the run—time system.

Table A—2. Run—tine Error Messages

STRING OVERFLOW (TRUNCATED) This error occurs when a string constant is assigned to a variable whose declared length is insufficient to hold the constant.
SUBSCRIPT/SUBRANGE OUT OF BOUNDS This error occurs when a subscripted array reference or a subrange reference is not within the declared bounds.
FLOATING POINT OVERFLOW This error occurs when a REAL number becomes larger than the largest possible number that can be represented in internal floating—point form.

End of Appendix A

B. Appendix

LINK68 Error Messages

LINK68 returns two types of error messages: diagnostic and logic. Both types of error messages have the following form:

```
LINK68:  <Error Message>
```

A diagnostic error prevents your program from linking. You should make the appropriate correction to your program and try again.

A logic error is a non—recoverable error in the internal logic of LINK68. If you receive one of these messages, contact the place you purchased your system for assistance. You should provide the following:

- The version of the operating system you are using.
- A description of your system's hardware configuration.
- Sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, also provide a disk with a copy of the program.

Diagnostic Error Messages

Table B—l list the LINK68 diagnostic errors in alphabetic order with explanations and suggested user responses.

Table B-l. LINK68 Diagnostic Error Messages

Message	Meaning
LINK68:ILLEGAL CHARACTER: '<char>'	The character <char> is not a legal character in the command line. Correct the error and relink.
LINK68: SYNTAX ERROR, EXPECTED: <item>	There is a syntax error in the command line. LINK68 expected to encounter <item>. Correct the error and relink.
LINK68: UNEXPECTED END OF COMMAND STREAM	LINK6S unexpectedly encountered the physical end of the command stream before the logical end. Check the command line for proper syntax and options.
LINK68: UNRECOGNIZED OR MISPLACED OPTION NAME: '<option>"	The option given by <option> is not a valid LINK68 option, or it is misplaced in the command line. Correct the error and relink.
LINK68: HEAP OVERFLOW -- NOT ENOUGH MEMORY	There is not enough memory for LINK68 to continue processing.

`LINK68: IMPROPERLY FORMED HEX NUMBER: "<num>"`

The hexadecimal number h contains an invalid digit. Correct the error and relink.

`LINK68: PARSE END BEFORE COMMAND STREAM END`

LINK68 has unexpectedly encountered the logical end of the command line before the physical end. Check the command line for proper syntax and options.

`LINK68: CANNOT OPEN <filename> FOR INPUT`

The file indicated by the variable <filename> is invalid, or the file does not exist. Check the filename before you reenter the LINK68 command line.

`LINK68: NESTED COMMAND FILES NOT ALLOWED`

LINK68 does not allow you to nest command files. Correct the error and relink.

`LINK68: TOO MANY OVERLAYS`

LINK68 allows a maximum of 255 overlays. Examine your program and simplify the overlay scheme. Reassemble or recompile the source code before relinking.

`LINK68: COMMAND LINE TOO LONG`

The command line does not fit on one line. Correct the error by using a command file and relink.

`LINK68: OVERLAYS NESTED TOO DEEPLY`

LINK68 allows only 5 levels of overlays. Examine your program and simplify the overlay scheme. Reassemble or recompile the source code before relinking.

`LINK68: CANNOT SET DATA OR BSS BASE WHEN USING OVERLAYS`

The BSSBASE and DATABASE options are not allowed when linking overlays. Correct the error end relink.

`LINK68: ILLEGAL REFERENCE TO OVERLAY SYMBOL "<symbol-name>" FROM MODULE <module-name>`

The module indicated by <module—name> contains a illegal reference to the symbol indicated by <symbol—name>.

`LINK68: "<symbol-name>" DOUBLY DEFINED IN <filename>`

The symbol <symbol—name> is defined twice. The variable <filename> indicates the file where the second definition occurs. Rewrite the source code and provide a unique definition for each symbol. Reassemble or recompile the file before relinking.

`LINK68: FILE FORMAT ERROR IN <filename>`

The file indicated by the variable <filename> is not an object file or the file has been corrupted. Ensure that the file is an object file, output by the assembler or compiler. Reassemble or recompile the file before relinking.

LINK68:	INVALID SYMBOL FLAG IN <filename>
	LINK68 does not recognize the symbol flags indicated by the variable <filename>. The file is not an object file or it has been corrupted. Ensure that the file is an object file, output by the assembler or compiler. Reassemble or recompile the file before relinking.
LINK68:	INVALID RELOCATION FLAG IN <filename>
	The contents of the file indicated by the variable <filename> are incorrectly formatted. The file is not an object file or it has been corrupted. Ensure that the file is an object file, output by the assembler or compiler. If the file is an object file and this error occurs, the file has been corrupted. Reassemble or recompile the file before relinking.
LINK68:	NO RELOCATION BITS IN <filename>
	The file indicated by the variable <filename> is not an object file or has been corrupted. Ensure that the file is an object file, output by the assembler or compiler. If the file is an object file and this error occurs, the file has been corrupted. Reassemble or recompile the file before relinking.
LINK68:	WRITE ERROR ON FILE: <filename>
	The disk to which LINK68 is writing is full. Erase unnecessary files, if any, or insert a new disk before you reenter the LINK68 command line.
LINK68:	READ ERROR ON FILE <filename>
	The object file indicated by the variable <filename>, does not have enough bytes. The file either is incorrectly formatted or has been corrupted. This error is commonly caused when the input to LINK68 is a partially assembled or compiled object file. The assembler, AS68, and some compilers create partial object files when they receive the disk full abort message while assembling or compiling a file. Ensure that the file is a complete object file. Reassemble or recompile the file before relinking.
LINK68:	SYMBOL TABLE OVERFLOW
	The object code contains too many symbols for the size of the symbol table. Rewrite the source code using fewer symbols. Reassemble or recompile the file before relinking.
LINK68:	UNABLE TO CREATE FILE <filename>
	Either the output file indicated by <filename> has an invalid drive code, or the disk to which LINK6B is writing is full. Check the drive code. If it is correct, the disk is full. Erase unnecessary files, if any, or insert a new disk before you reenter the LINK6B command line.
LINK68:	UNABLE TO OPEN TEMPORARY FILE <filename>
	Either the file, indicated by <filename>, has an invalid drive code, specified by the f option, or the disk to which LINK68 is writing is full. Check the drive code. If it is correct, the disk is full. Erase unnecessary files, if any, or insert a new disk before you reenter the LINK68 command line.

`LINK68: UNDEFINED SYMBOL(S)` The symbol or symbols which are listed one per line on the lines following the error message are undefined. Provide a valid definition and reassemble the source code before you reenter the LINK68 command line. If the symbols are not referenced by the program, you can use the UNDEFINED option in the command line.

Internal Logic Errors

The following list identifies the LINK68 internal logic error messages.

```
LINK68:  INTERNAL ERROR IN <procname>
LINK68:  TEXT SIZE ERROR IN <filename>
LINK68:  RELATIVE ADDRESS OVERFLOW AT Lx IN <filename>
LINK68:  SEEK ERROR ON FILE <filename>
LINK68:  SHORT ADDRESS OVERFLOW IN <filename>
LINK68:  UNABLE TO REOPEN FILE <filename>
```

End of Appendix B

C. Appendix

Run-time Library Routines

This appendix describes the run—time library routines that are specific to the implementation for the Motorola MC68000 microprocessor and the CP/M—68K operating system.

The following tables list the names of the routines and their purposes. Knowledge of what these routines do can be helpful when you are disassembling a program.

Note: You should not call these routines from your program because Digital Research does not guarantee parameter list compatibility between releases.

Table C–1. PASLIB Routines

	System Access
_BDOS	Call operating system directly
_CHN	Program chaining routine
CHAIN	Pascal interface for
_HLT	Halt routine; returns to operating system
_INI	Run—time initialization
_XJP	Table case jump routine

	String Handling Routines
Routine	Purpose
_EQD	String comparison routine for =
_NED	String comparison routine for <>
_GTD	String comparison routine for >
_LTD	String comparison routine for <
_GED	String comparison routine for >=
_LED	String comparison routine for <=
_LBA	Load concat string buffer address
_ISB	Initialize string buffer
_CNC	Concatenate a string to the buffer
_STR	String store
_RST	Read a string from a file
_WCH	Write a string to a file
POS	Run-time support for strings

	Set Manipulation Routines
Routine	Purpose
_EQS	Set equality
_NES	Set inequality
_GES	Set superset
_LES	Set subset
_SAD	Set union
_SSB	Set difference
_SML	Set intersection
_SIN	Set membership

_SIN	Build singleton set
_BST	Build subrange set
_BSR	
_EQA	Array comparison routine for =
_NEA	Array comparison routine for <>
_GTA	Array comparison routine for >
_LTA	Array comparison routine for <
_GEA	Array comparison routine for >=
_LEA	Array comparison routine for <=

	Character Manipulation Routines
Routine	Purpose
_CCH	Concatenate a character to the buffer
_RNC	Read next character from a file
_WNC	Write next character to a file
_RCH	Read a character from a file
_CHW	Write a character to a file
_CRL	Write a newline character (CR) to a file

	Bit Manipulation Routines
Routine	Purpose
TSTBIT	Test for a bit on
SETBIT	Turn a bit on
CLRBIT	Turn a bit off
	I/O and File Handling Routines
Routine	Purpose
_SFB	Set global FIB address
_DWD	Set default width and decimal places
_SIA	Reset input vector
_SOA	Reset output vector
_DIO	Set I/O vectors to default addresses
_CWT	Read until EOLN is True on a file
_RNB	Read n bytes from a file
_WNB	Write n bytes to a file
OPEN	File handling routine
BLOCKREA	File handling routine
BLOCKWRI	File handling routine
SEEKREAD	File handling routine
SEEKWRIT	File handling routine
CREATE	File handling routine
CLOSE	File handling routine
CLOSEDEL	File handling routine
GNB	File handling routine
WNB	File handling routine
PAGE	File handling routine
EOLN	File handling routine
EOF	File handling routine
RESET	File handling routine
REWRITE	File handling routine
GET	File handling routine
PUT	File handling routine
ASSIGN	File handling routine
PURGE	File handling routine

IORESULT	File handling routine
COPY	File handling routine
INSERT	File handling routine
DELETE	File handling routine

Arithmetic Routines	
Routine	Purpose
_MUL	Multiply a long integer
_RIN	Read integer from a file
_RDL	Read a long integer from a file
_WIN	Write an integer to a file
_RTL	Write a long integer to a file
_DVL	32—bit DIV software routine
_MDL	32-bit MOD software routine

Memory Manipulation Routines	
Routine	Purpose
MOVELEFT	Block move left end to left end
MOVERIGH	Block move right end to left right
_NEW	Allocate memory for NEW procedure
_DSP	Deallocate memory for DISPOSE procedure
MEMAVAIL	MEMAVAIL function
MAXAVAIL	MAXAVAIL function
LMEMAVAI	LMEMAVAIL function
LMAXAVAI	LMAXAVAIL function

Table C-2. BCDREALS Routines

Routine	Purpose
_EQR	Real comparison for =
_NER	Real comparison for <>
_GTR	Real comparison for >
_LSR	Real comparison for <
_GER	Real comparison for >=
_LER	Real comparison for <=
_XOP	Floating-point operations
_RAD	Real add
_RSB	Real subtract
_RML	Real multiply
_RDV	Real divide
_RNG	Real negate
_RAB	Real absolute value
_QQS	Store a real
_FLT	Convert integer to float
TRUNC	Built-in truncate function

ROUND	Built-in round function

Table C-3. FPREALS Routines

Routine	Purpose
_EQR	Real comparison for =
_NER	Real comparison for <>
_GTR	Real comparison for >
_LSR	Real comparison for <
_GER	Real comparison for >=
_LER	Real compasison for <=
_RAD	Real add
_RSB	Real subtract
_RML	Real multiply
_RDV	Real divide
_RNG	Real negate
_RAB	Real absolute value
_XOP	Floating-point operations
_RRL	Read a real from a file
_WRL	Write a real to a file
_QQS	Store a real
_FLT	Convert integer to float
TRUNC	Built-in truncate function
ROUND	Built-in round function
SQR	Built-in square function
SQRT	Built-in square root function
SIN	Built—in sine function
COS	Built—in cosine function
ARCTAN	Built-in arctangent function
EXP	Built-in exponential function
LN	Built-in natural log function

Table C-4. FULLHEAP Routines

Routine	Purpose
_NEW	Allocate memory from heap
_DSP	Return memory space to heap

End of Appendix C

D. Appendix

Internal Data Representation

This appendix describes how Pascal/MT+ internally represents the constants and variables declared in your programs. This information is useful when you want to interface Pascal/MT+ code with assembly language programs (see Section 8).

Each Pascal/MT+ implementation differs in the way it internally represents data. The information presented here is specific to the Motorola MC68000 microprocessor running under the CP/M-68K operating system.

Size and Range of Data types

The table below summarizes the size and range of Pascal/MT+ data types for the 68K implementation.

Table D—l. Size and Range of Pascal/MT+ Data Types

Data Type	Size	Range
BOOLEAN	2 bytes	FALSE .. TRUE
BYTE	1 byte.	0 .. 255
CHAR	1 byte.	0 .. 255
INTEGER	2 bytes	-32768 .. 32767
LONGINT	4 bytes	2^{-32} .. 2^{+32}
WORD	2 bytes	0 .. 65535
BCD REAL	10 bytes	18 total digits, 4 decimal places
FLOATING REAL	8 bytes	10^{-307} .. 10^{307}
SET	32 bytes	0 .. 255
STRING	1. .256 bytes	

Multibyte Storage

All data represented by multiple bytes is stored in memory with the high—order (most significant) byte first. That is, the high-order byte appears at the lowest address; then the other bytes appear at increasing addresses with the low-order (least significant) byte at the highest address.

BOOLEAN Representation

Pascal/MT+ represents variables of type BOOLEAN using two consecutive bytes. The high-order byte is stored first. The least significant bit (LSB) in the low-order byte determines the value. If the bit is 0, the value is TRUE; if the bit is 1, the value is FALSE.

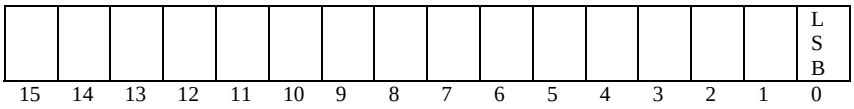


Figure D-1. BOOLEAN Representation

BYTE Representation

Pascal/MT+ represents variables of type BYTE using one byte. All the bits are considered significant.

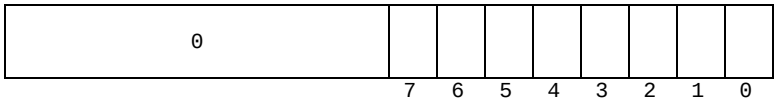


Figure D-2. BYTE Representation

CHAR Representation

Pascal/MT+ represents variables of type CHAR using one byte to contain the ASCII representation. The most significant bit (MSB) is ignored.

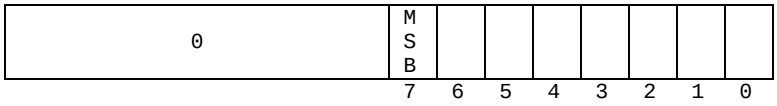


Figure D-3. CHAR Representation

INTEGER Representation

Pascal/MT+ represents variables of type INTEGER in two's complement form using two consecutive bytes. The high-order byte is stored first, and the most significant bit (MSB) is the sign bit.

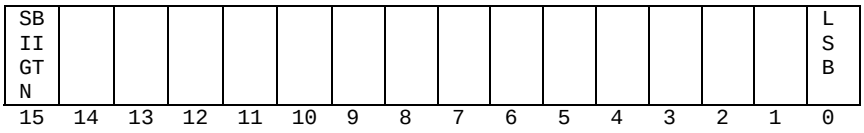


Figure D-4. INTEGER Representation

LONGINT Representation

Pascal/MT+ represents variables of type LONGINT in two's complement form using four consecutive bytes. The high—order byte is stored first, and the most significant bit (MSB) is the sign bit.

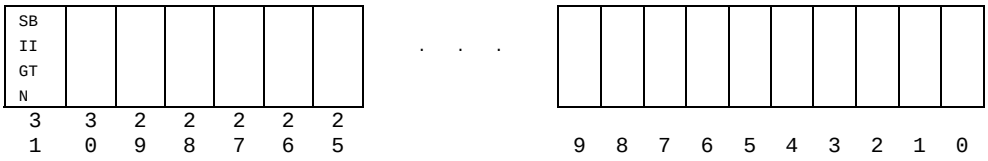


Figure D-5. LONGINT Representation

WORD Representation

Pascal/MT+ represents variables of type WORD using two consecutive bytes. The high-order byte is stored first. All the bits are considered significant.

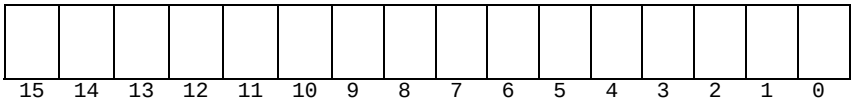


Figure D-5. WORD Representation

REAL Representation

Pascal/MT+ represents variables of type REAL using two different formats:

- Fixed-point variables use the Binary Coded Decimal (BCD) format. Fixed—point numbers are decimal numbers that have a fixed total number of digits and a fixed number of digits to the right of the decimal point.
- Floating—point variables use the Institute of Electrical and Electronic Engineers (IEEE) double precision format. Floating-point numbers are very large or very small numbers expressed in scientific notation with a mantissa and an optionally signed integer exponent.

BCD Format

The BCD format uses 10 consecutive bytes with the high-order byte stored first.

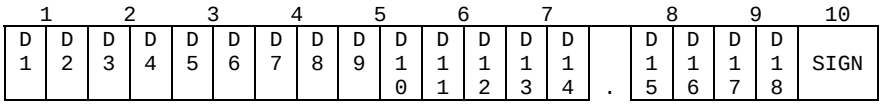


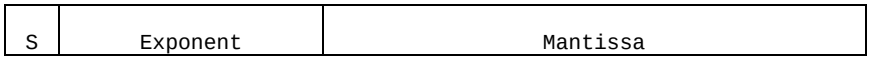
Figure D-7. BCD REAL Representation

In bytes 1 through 9, the decimal digits are packed two to **a** byte. That is, each digit occupies four bits. Byte 10 is reserved for the sign, with 0 for positive, and FF11 for negative.

There is an implicit decimal point immediately preceding byte number 8, so the BCD format can represent a number with 18 total digits and 4 digits to the right of the decimal point.

IEEE Format

Pascal/MT+ represents floating-point binary data using the IEEE double—precision format. This format uses eight consecutive bytes, with the 64 bits containing the following fields: a 52— bit mantissa, an 11—bit exponent, and a sign-bit. The least significant byte of the mantissa is stored at the highest memory address.



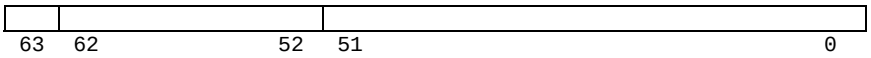


Figure D-8. Double-precision Floating-point Format

The double-precision format normalizes floating—point numbers so the most significant bit of the mantissa is always 1 for nonzero numbers. Because the most significant bit of the mantissa must be 1 for nonzero numbers, this bit is not stored. This is called using an implicit normalized bit. The binary point is considered to be immediately to the right of the normalized bit.

In the double—precision format, the exponent has a bias of 1023 (decimal) or 3FF (hexadecimal) so 400 represents an exponent of +1 while 3FE represents an exponent of -1.

Suppose a double-precision floating—point binary number appears in memory as the eight—byte value:

```
CO 43 CO 00 00 00 00 00
higher memory ——-->
You can visualize this value as a string of 64 bits in the form:
   C  0  4  3  C  0  0  0  0  0  0  0  0  0
   1100 0000 0100 0011 1100 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

The high—order bit equal to 1 indicates the sign is negative.

```
C  0  4  3 C 0  .  . . .
1100  0000 0100 0011 1100 0000 .    .    . .
```

Ignoring the sign bit yields a biased exponent value of

```
4    0    4    .    .
0100 0000 0100 .    .
^
```

s (ignored)

Subtracting the bias (3FF) from the exponent 404 gives a true binary exponent of 5.

Restoring the implicit normalized bit to the mantissa produces the bit pattern shown below:

```
3  C  0  . .
0011 1100  0000    . .
1001 1110  0000    . .
^
```

implicit normalized bit (restored)

Because the binary point is one position to the right of the implicit normalized bit, the value of the mantissa is

```
1 001 1110 0000
 ^
```

Since the true binary exponent is 5, the binary point must be shifted to right 5 places, giving a new value to the mantissa as shown below:

```
1001 11 10 0000 . .
   ^
```

To calculate the value represented by the mantissa, multiply by the true binary exponent, which is now 2 . because the binary point has been shifted to the right.

```
(2 + 2 + 2 + 2 + 2) * 2 = (32 + 4 + 2 + 1 + 1/2) * 1 = 39.5
```

Thus, the eight-byte value

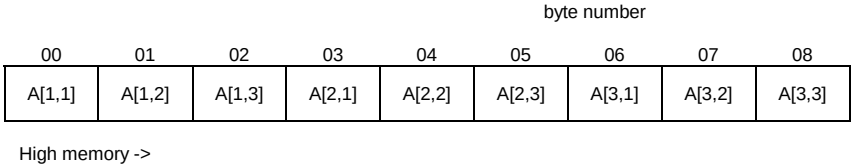
```
00 00 00 00 00 CO 43 CO
```

is the double-precision float—binary representation of the decimal number -39.5.

Array Representation

Pascal/MT+ represents variables of type ARRAY in row-major order. Figure D—9 shows the storage for the declaration:

```
A: ARRAY [1. .3, 1. .31] OF CHAR
```

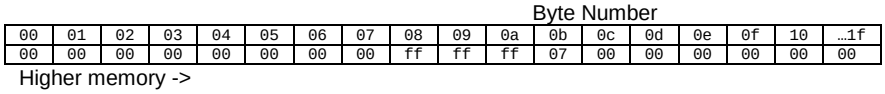


Logically, this is a one-dimensional array of vectors. In Pascal/MT+, all arrays are logically one-dimensional arrays of some type.

Set Representation

Pascal/MT+ represents variables of type SET using 32 consecutive bytes with each element of the set using one bit. The low-order bit (bit 0) of each byte is the least significant bit in the byte.

Figure D-l0 shows the storage for the set A. .Z. The first element in the set is capital A, which occupies position 65 in the ASCII collating sequence (see Appendix F). Thus, the first bit in the set is bit 65, the first bit in byte 8. The last bit in the set is bit 90, which is bit 2 in byte 11, and corresponds to capital Z.



Static Data Allocation

Pascal/MT+ allocates space for variables in the order you declare them. The exception is variables appearing in an identifier list before a type. These are allocated in reverse order. For example, given the declaration:

```
VAR
a,b,c : INTEGER
```

c is allocated first, then b, then a.

Global Variables

Pascal/MT+ stores global variables contiguously with no space left between one declaration and the next. For example, given the declaration

```
VAR
     a    : INTEGER;
     b    : CHAR;
     i,j,k       : BYTE;
```

```
l       : INTEGER;
p       : ^INTEGER;
```

Pascal/MT+ stores the variables as shown below:

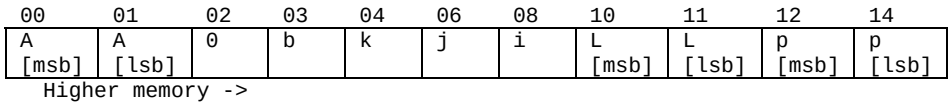


Figure D-11 Contiguous Variable Storage

Local Variables

All local variables are allocated on the stack. If a single-byte variable (BYTE or CHAR) falls on an odd byte boundary, the compiler pads the variable with one byte and aligns it on a word boundary to improve code efficiency.

End of Appendix D

E. Appendix

Writing Portable Programs

This appendix describes certain features of Pascal/MT+ that are not portable to other implementations. This does not mean that these features are not available in other implementations, but only indicates that if they are available, they are implemented differently.

If you want to write portable programs, you should avoid using the implementation-dependent features listed below, but if you do, isolate them so that they are easy to locate and modify when you port the program.

Hardware-dependent Features

All the following Pascal/MT+ features depend on detailed knowledge of a particular processor's architecture and native instruction set.

- ABSOLUTE variable addressing
- INLINE
- INTERRUPT procedures
- I/O port addressing
- Redirected I/O

System-dependent Features

All the following Pascal/MT+ features either depend on a particular implementation's run-time system or operating system's file structure. Thus, they can vary from one implementation to another.

- logical device names such as CON: and RDR:
- the values returned by IORESULT
- chaining from one program to another
- having overlays call other overlays
- dependence upon EOF for non—TEXT files. Some operating systems keep track of how much data is in the file to the exact byte, while others only keep track to the sector/block level, and the last sector/block can contain uninitialized data.
- BLQCKREAD/BLOCKWRITE depends on knowledge of the correct allocation block size in the BIOS. Use SEEKREAD/SEEKWRITE instead.
- temporary files

In general, if compliance with the ISO standard is desired, you should avoid using variant records that circumvent type checking.

End of Appendix E