the engineering and scientific software series

86/PC experts–PL/M-86

Compiler and Language Reference Guide

Caine, Farber &
Gordon, Inc.

Warren Point
International Limited

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure of the software described herein is governed by the terms of a license agreement or, in the absence of an agreement, is subject to restrictions stated in paragraph (b)(3)(B) of the Rights in Technical Data and Computer Software clause in DAR 7-104.9(a) or in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause in FAR 52.227-7013, as applicable.

Comments or questions relating to this manual or to the subject software are welcomed and should be addressed to:

Caine, Farber & Gordon, Inc.
1010 East Union Street
Pasadena, CA 91106
USA
Tel: (818) 449-3070
Telex: 295316 CFG UR

Warren Point International Limited
Babbage Road, Stevenage
Hertfordshire SG1 2EQ
ENGLAND
Tel: Stevenage (0438) 316311
Telex: 826255 DBDS G

ISBN 1-55714-008-1
Order Number: 9301-50

First printing, March, 1988

Copyright © 1982, 1984, 1986, 1988 by Caine, Farber & Gordon, Inc.
All Rights Reserved.

86/PC and 86/PL are trademarks of Caine, Farber & Gordon, Inc. Experts-PL/M and Experts-PL/M-86 are trademarks of Caine, Farber & Gordon, Inc. and Warren Point International Limited. UNIX is a trademark of AT&T Bell Laboratories. VAX, VMS, and Ultrix are trademarks of Digital Equipment Corporation. TNIX is a trademark of Tektronix, Inc. MCS is a trademark of Intel Corporation. MS and XENIX are trademarks of Microsoft Corporation.

Table of Contents

Chapter 1 Introduction 1

1.1 Supported Environments 1
1.2 Installing 86/PC 1
1.3 Features and Capabilities 1
1.4 Organization of this Manual 2

Chapter 2 General Information 3

2.1 Invoking 86/PC Under UNIX and PC-DOS 3
2.1.1 Normal Invocation Options 3
2.1.2 Preprocessor Control Options 5
2.1.3 Compiler Debugging Invocation Options 5
2.1.4 Argument Files 6
2.1.5 Redirecting the Standard Error File 6
2.1.6 Return Codes 6
2.2 Invoking 86/PC Under VMS 6
2.2.1 Normal Invocation Options 7
2.2.2 Preprocessor Control Options 9
2.2.3 Completion Status 10
2.3 Overall Operation of 86/PC 10
2.4 The 86/PC Compile-Time Control Language 10
2.4.1 Compile-Time Expressions 11
2.4.1.1 Compile-Time Variables 11
2.4.1.2 Compile-Time Constants 11
2.4.2 The "INCLUDE" Control 11
2.4.3 The "SET" Control 11
2.4.4 The "RESET" Control 12
2.4.5 Conditional Compilation 12
2.4.6 Listing Controls 12
2.4.7 Other Controls 13
2.4.8 Other Controls 13
2.5 86/PL Source Format 13
2.5.1 Blanks and Comments 13
2.5.2 Statement Recognition 13
2.6 Object Module Format 14
2.7 Run-Time Support 14

Chapter 3 Introduction to the Meta-Language 15

Chapter 4 Modules and Procedures 17

4.1 Module Definitions 17
4.2 Main Programs 17
4.2.1 Main Program Statement Labels 17
4.3 Procedure Declarations 18
4.4 Procedure Parameters 18
4.5 Procedure Types 18
4.6 Procedure Scope 18
4.6.1 Public and Internal Procedures 19
4.6.2 External Procedures 19
4.7 Procedure Class 19
4.7.1 Reentrant Procedures 19
4.7.2 Interrupt Procedures 19

Chapter 5 DECLARE Statements 21

5.1 Factored Declarations 21
5.2 The LABEL Attribute 21
5.3 The LITERALLY Attribute 22
5.4 The At Attribute 22
5.5 The DATA and INITIAL Attributes 22
5.5.1 Signed Constants 23
5.5.2 Restricted Expressions 23
5.6 Element Attributes 23
5.6.1 The EXTERNAL Attribute 23
5.6.2 The PUBLIC Attribute 23
5.6.3 The BASED Attribute 24
5.6.4 The Dimension Attribute 24
5.6.5 The Basic Type Attributes 24
5.6.6 The STRUCTURE Attribute 25

Chapter 6 Executable Statements 27

6.1 DO Groups 27
6.1.1 The DO Statement 27
6.1.2 The WHILE Statement 27
6.1.3 The Iterative DO Statement 28
6.1.4 The CASE Statement 28
6.1.5 The UNDO Statement 28
6.2 The IF Statement 28
6.3 IF Blocks 29
6.3.1 Block If Statement 29
6.3.2 The ELSEIF Statement 29
6.3.3 The ELSE Statement 30
6.3.4 The ENDIF Statement 30
6.4 Simple Statements 30
6.4.1 The Assignment Statement 30
6.4.2 The CALL Statement 30
6.4.3 The GOTO Statement 31
6.4.4 The Null Statement 31
6.4.5 The RETURN Statement 31
6.4.6 Special Statements 31
6.5 Endings 32
6.6 Label Definitions 32

6.7 Compatible Types . . . 32
6.8 Conditional Expression . . . 32

Chapter 7 Expressions . . . 33

7.1 Operators . . . 33
7.2 Relational Operators . . . 34
7.3 Constant Operands . . . 34
7.4 Embedded Assignments . . . 35
7.5 Addresses . . . 35
7.6 References . . . 35
7.6.1 Function Reference . . . 35
7.6.2 Assignment Target Reference . . . 36
7.6.3 Restricted Reference . . . 36
7.6.4 Inexact Reference . . . 36
7.6.5 Explicitly Based Reference . . . 36
7.6.6 Identifiers . . . 36
7.7 Constants . . . 37

Chapter 8 Builtin Identifiers and Functions . . . 39

8.1 Size of Variables . . . 39
8.2 Type Conversion . . . 39
8.3 Shift and Rotate . . . 40
8.4 Referencing Subfields . . . 40
8.5 Constructing a Pointer . . . 41
8.6 The Stack Pointer and Stack Base . . . 41
8.7 Decimal Adjustment . . . 41
8.8 Absolute Value . . . 41
8.9 Square Root and PI . . . 42
8.10 Time Delays . . . 42
8.11 String Operations . . . 42
8.11.1 String Move . . . 42
8.11.2 String Set . . . 42
8.11.3 String Translation . . . 42
8.11.4 String Find and Skip . . . 43
8.11.5 String Compare . . . 43
8.12 Flag Values . . . 43
8.13 Input and Output . . . 44
8.14 Multiprocessor Synchronization . . . 44
8.15 Addressing Interrupt Procedures . . . 44
8.16 Setting the 8087 Mode . . . 44
8.17 The Memory Array . . . 44

Appendix A 86/PL and PL/M-86 Differences . . . 45

A.1 Extensions to PL/M-86 . . . 45
A.1.1 Reserved Words . . . 45
A.1.2 Declare Statement . . . 45
A.1.3 The Interrupt Attribute . . . 46
A.1.4 Restricted Expressions . . . 46
A.1.5 Explicitly Based Variables . . . 46
A.1.6 Builtin Functions as Assignment Targets . . . 47
A.1.7 The IF Block . . . 47
A.1.8 The UNDO statement . . . 47

A.2 Unsupported PL/M-86 Features 48

Appendix B Error Messages 49

B.1 Warnings 49
B.2 Errors 49
B.3 Severe Errors 49
B.4 Fatal Errors 49
B.5 List of Error Messages 49

Appendix C Formal Definition of Meta-Language 59

Appendix D Multiply and Divide for Double Words 61

D.1 Routines for Intel 8086 Assembler 61
D.2 Routines for Tektronix 8086 Assembler 63

Appendix E Installing on VAX/VMS 65

E.1 Supported Operating Environment 65
E.2 Restoring the Tape 65
E.3 Defining Logical Names 65
E.4 Installing the Native Commands 66

Appendix F Installing on UNIX Systems 67

F.1 Supported Operating Environment 67
F.2 Binary Installation 67
F.3 Selecting 86/PC Final Output Default 67
F.4 Tailoring With Environment Variables 68
F.4.1 Global Tailoring Changes 68
F.4.2 Local Tailoring Changes 68
F.4.3 Specifying Maximum Number of Arguments 68
F.5 Tailoring With an Initialization File 69
F.5.1 Examples 69

Appendix G Installing on PC-DOS 71

G.1 Supported Operating Environment 71
G.2 Restoring the Diskette 71
G.3 Making the Temporary Directory 71
G.4 Installing 86/PC in the Search Path 72
G.5 Selecting 86/PC Final Output Default 72
G.6 Tailoring With Environment Variables 72
G.6.1 Global Tailoring Changes 72
G.6.2 Local Tailoring Changes 73
G.6.3 Specifying Maximum Number of Arguments 73
G.7 Tailoring With an Initialization File 73
G.7.1 Examples 73

Appendix H Installing on Tektronix 856x 75

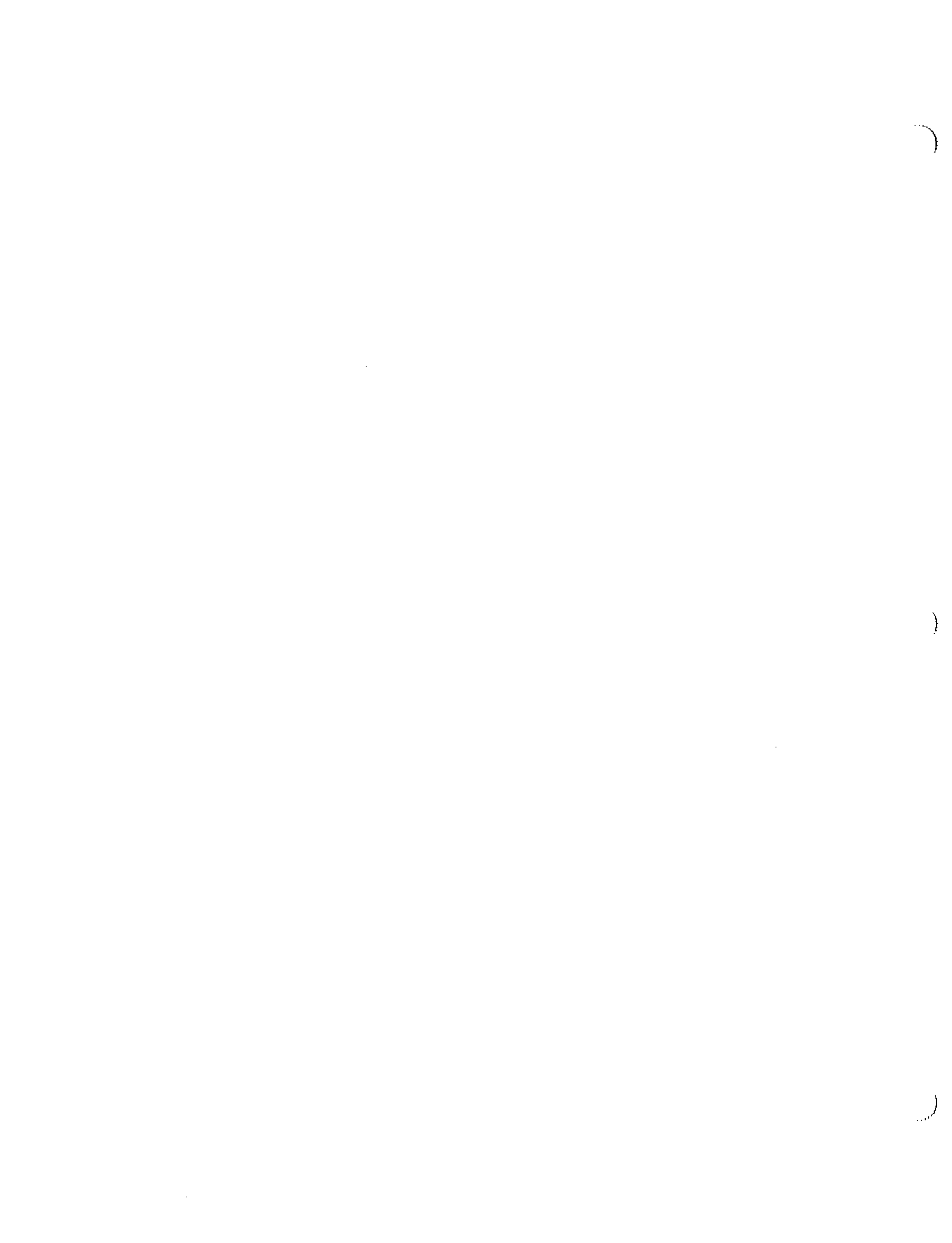
H.1 Supported Operating Environment 75
H.2 Restoring the Diskette 75
H.3 Selecting 86/PC Final Output Default 75
H.4 Tailoring With Environment Variables 76
H.4.1 Global Tailoring Changes 76
H.4.2 Local Tailoring Changes 76
H.4.3 Specifying Maximum Number of Arguments 76
H.5 Tailoring With an Initialization File 77

H.5.1 Examples 77

Appendix I Source Installation on UNIX Systems 79

I.1 Supported Operating Environment 79
I.2 Restoring the Delivery Tape 79
I.2.1 Tape Format 79
I.2.2 Restoring the Tape 79
I.2.3 Compiling the Sarin Utility 80
I.2.4 Extracting Source Files From the Archives 80
I.2.4.1 Structure of the Source Archives 81
I.2.4.2 Processing the Source Archives 81
I.3 Installing the 86/PC Compiler 82
I.3.1 Restoring the 86/PC Delivery Tape 82
I.3.2 Modifying the 86/PC Shell Scripts 82
I.3.2.1 Modifying pcdefs.sh 83
I.3.2.2 Modifying pccompile.sh 84
I.3.2.3 Modifying pclink.sh 84
I.3.2.4 Modifying pcinstall.sh 85
I.3.2.5 Modifying pcprint.sh 85
I.3.3 Using the 86/PC Shell Scripts 85
I.3.3.1 Compiling the Source 85
I.3.3.2 Linking the Object 85
I.3.3.3 Installing 86/PC 85
I.3.3.4 Listing 86/PC 85

Index 87



1. Introduction

The 86/PC™ Experts–PL/M-86™ is a compiler which accepts the 86/PL language as input and generates code for the Intel 8086, 80186, and 80286 microprocessors. The 86/PL language is a superset of the PL/M-86 language and most PL/M-80 and PL/M-86 source modules should compile and execute with little or no modification.

1.1 SUPPORTED ENVIRONMENTS

The compiler operates on many different machines and operating systems including the VAX™ under VMS™ and Ultrix™, the Tektronix 8560 under TNIX™, and most other UNIX™ systems.

1.2 INSTALLING 86/PC

Different methods are required for installing 86/PC under the various supported environments. See Appendix E through Appendix I for installation instructions.

1.3 FEATURES AND CAPABILITIES

This manual describes the 86/PL programming language and the operation of the 86/PC compiler. It is intended as a reference guide and *not* as a tutorial. It is assumed that the reader is already familiar with programming in the Intel PL/M-80 or PL/M-86 languages.

The 86/PL language contains a number of extensions to PL/M-86, including:

- Relaxation of most restrictions on reserved words;
- Relaxation of restrictions on the ordering and factoring of items in DECLARE statements;
- Introduction of structures within structures;
- Introduction of explicitly based references;
- Use of the HIGH, LOW, SELECTOR$OF and OFFSET$OF builtins as assignment targets;
- Introduction of the SQRT and PI builtins for reals;
- Introduction of a fully-delimited IF block construct;
- Introduction of an UNDO statement for premature loop exits; and
- Introduction of a new scope for external data and procedures so that external items declared in an included file may be redeclared within a module.

The 86/PC compiler supports the SET, RESET, and conditional compilation controls of

the PL/M-86 compiler. The INCLUDE compiler control is also supported, except that the path name of a file to be included must correspond to the syntax of a host path name. The other PL/M-86 compiler controls are not supported.

See Appendix A for a complete description of the differences between 86/PL and PL-M/86.

1.4 ORGANIZATION OF THIS MANUAL

The remainder of this manual is divided into several parts:

- Chapter 2 provides general information on the compiler, including how to invoke it in various operating environments;
- Chapter 3 provides an informal introduction to the metalanguage which is used throughout the manual. A formal definition is provided in Appendix C.
- Chapter 4 through Chapter 8 provide a detailed description of the 86/PL language.
- Appendix A discusses the differences between 86/PL and PL/M-86.
- Appendix B discusses the error messages that may be produced by the compiler.
- Appendix D provides listings of routines needed for multiply and divide of double words.
- Appendix E through Appendix I discuss installation of the compiler on the various supported machines and operating systems.

2. General Information

This chapter provides general information to users of the 86/PL language and the 86/PC compiler. It includes discussions of the compiler invocation procedure, the format of the object module, and the compile-time control language.

2.1 INVOKING 86/PC UNDER UNIX AND PC-DOS

Under the UNIX and PC-DOS operating systems, the 86/PC compiler is invoked by:

```
86pc [option]... file...
```

The normal compiler operation is to compile each file and place the resulting object module into a file with the same name as the source file with any ".*" suffix replaced with ".q". If a source file does not have a suffix, the object file name is formed by postpending ".q".

The object files are, in general, not immediately executable. They should be ultimately linked with any required libraries and then bound to addresses reasonable for the final environment of the executable program.

The normal operation of the compiler may be modified by the use of various options as described in the following sections.

2.1.1 Normal Invocation Options

The options used in normal invocations of 86/PC are:

-l Generate a source listing and place it on the standard output file.

-a Generate a symbolic, assembler-like, listing and place it on the standard output file.

-x Generate a source listing and a cross reference listing and place both on the standard output file.

-s Perform syntax checking but do not generate code or produce a ".q" file. This option causes only the preprocessor, phase 1, and phase 2 (Section 2.3) to be run.

-M*string* Change the compiler's model of the machine. The letters in *string* indicate:

c	separate code segments
d	separate data segments
s	separate data and stack
m	separate data and memory
r	constants with the code (rom)
p	four byte pointers
a	alternate linkage
1	target machine is an 80186
2	target machine is an 80286

Therefore no "-M" option is the SMALL control, "-Mcp" is the MEDIUM control, "-Msmp" is the COMPACT control, and "-Mcdsmrp" is the LARGE control. The addition of "r" is equivalent to using the ROM control.

-O*string* Change the level of optimization. The letters in *string* indicate:

n	turn off common subexpression optimization
a	assume AT variables do not change with each fetch
b	assume BASED stores do not alter any base pointer
p	use short method to compare pointers
i	ignore all implicit interactions

The option "-Opi" together with the "-J" option corresponds to the OPTIMIZATION=3 control.

-J Cause the optional jump optimization phase to be invoked. This will result in smaller, faster programs in many cases but will increase the compilation time.

-S Generate a symbolic, assembler-like, listing. The listing is placed in the corresponding ".S" file.

-L Generate local symbol and line number records in the object file for possible use by a run-time debugging system.

-d Generate local symbol and line number records in the object file for possible use by a run-time debugging system. Use the line numbers given in the source listing.

-i Generate an Intel standard object module and place it in the corresponding '.q' file. This may be established as the default when the compiler is installed.

-t Generate a Tektronix LAS object module and place it in the corresponding '.q' file. This may be established as the default when the compiler is installed.

-p*nnn* *nnn* is an integer giving the number of lines per printed page. If this option is not specified, a value of 66 will be used.

-Xs*aaa* Specifies, as *aaa*, the default suffix to use for source file names that are not given with a suffix.

-Xp*aaa* Specifies, as *aaa*, the suffix to be used on object files in place of the default ".q".

-Xl*aaa* Specifies that any listing produced will be directed to a file, instead of to the standard output. The file will have the same name as the corresponding source file, but with a suffix of *aaa*.

-Xi*aaa* Specifies, as *aaa*, the suffix to be used on generated preprocessor output files, instead of the default ".i".

-XS*aaa* Specifies, as *aaa*, the suffix to be used on symbolic output files produced as a result of the -S option, instead of the default ".S".

-Xt*aaa* Specifies, as *aaa*, the *prefix* to be used on all temporary file names, instead of the default "\tmp\" under PC-DOS or "/usr/tmp/" under UNIX. As an example, "-Xt./" will cause temporary files to be created in the current directory (i.e., the one in use when 86PC is invoked).

2.1.2 Preprocessor Control Options

The normal action of the compiler preprocessor phase (Section 2.3 and Section 2.4) can be modified by:

-D*name* Define *name* as a compile-time variable and assign it the value "-1". The first attempt to redefine the variable with a SET control (Section 2.4.3) will be ignored.

-D*name*=*expression*

Define *name* as a compile-time variable and assign it the value of *expression*. *Expression* can be any valid compile-time expression (Section 2.4.1). The operands of the expression must be constants or the names of compile-time variables defined in preceding "-D" options. The first attempt to redefine the variable with a SET control (Section 2.4.3) will be ignored.

-I*list* Specify directories to be searched for an INCLUDE file (Section 2.4.2) if the file is not found in the directory of the source file. The *list* is a colon-separated list of directory paths.

-E Don't compile the source files. Instead, just run them through the preprocessor and place the output on the standard output file.

-P Don't compile the source files. Instead, just run them through the preprocessor and, for each, put the output into a corresponding ".i" file.

2.1.3 Compiler Debugging Invocation Options

These options may be useful when debugging the 80/PC compiler. Normally they should not be used.

-B*string* Prepend *string* to the name of each compiler phase before executing it, thus allowing alternate versions of the compiler to be executed.

-T Display interesting things about the compiler progress on the standard error file.

-TT Same as the "-T" option but don't actually call the compiler phases.

-V Display the compiler version number on the standard error file and immediately exit.

-K Do *not* delete the compiler intermediate files which remain at the end of the compilation.

2.1.4 Argument Files

Any command line argument may have the form

```
@argfile
```

where *argfile* is a file containing more arguments. This is particularly useful in cases where more arguments are required than will fit on the original command line.

2.1.5 Redirecting the Standard Error File

Error messages are written on the standard error file, which is usually the display screen. This may be changed by using a command line (or argument file) argument of the form

```
^errfile
```

where *errfile* is the name of the file to receive error messages. If the argument has the form

```
^^errfile
```

the messages will be appended to the file.

2.1.6 Return Codes

The compiler returns the following codes to its invoker. See Appendix B for more detailed descriptions of these codes.

0 Compilation completed with no errors.

1 Compilation completed with warnings.

2 Compilation completed with errors.

3 Compilation terminated with a severe error.

4 Compilation terminated with a fatal compiler error.

2.2 INVOKING 86/PC UNDER VMS

Under the VMS operating system, the 86/PC compiler is invoked by:

```
86PC [options] file-name

Command Qualifiers:            Defaults:

/[NO]CROSS_REFERENCE           /NOCROSS_REFERENCE
/[NO]DEBUG=(options)           /NODEBUG
/DEFINE=(name-list)
/INCLUDES=(directories)
/[NO]LIST[=file-spec]          /NOLIST
/[NO]MACHINE_CODE              /NOMACHINE_CODE
/MODEL=(options)
/[NO]OBJECT[=file-spec]        /OBJECT
/[NO]OPTIMIZE=(options)        /OPTIMIZE=SUBEXPRESSIONS
/[NO]PREPROCESS_ONLY           /NOPREPROCESS_ONLY
/SYNTAX
```

The normal compiler operation is to compile the named file and place the resulting object module into a file with the same name as the source file but with a file type of "Q86". The default file type for the source file is "P86".

The object files are, in general, not immediately executable. They should be ultimately linked with any required libraries and then bound to addresses reasonable for the final environment of the executable program.

The normal operation of the compiler may be modified by the use of various options as described in the following sections.

2.2.1 Normal Invocation Options

/CROSS_REFERENCE
/NOCROSS_REFERENCE

> Controls whether or not a cross-reference listing will be generated. If so, it will appear at the end of the listing file. For /CROSS_REFERENCE to operate, /LIST must also be in effect. The default is /NOCROSS_REFERENCE.

/DEBUG[=option]
/NODEBUG

> Specifies the type of debugging output to be placed in the generated object module. The options are:
>
> | | |
> |---|---|
> | LINE_NUMBERS | generate debug records which refer to input line numbers. Such numbers will *not* be assigned to lines which are contained in files which are input by the INCLUDE compiler control. |
> | STATEMENT_NUMBERS | generate debug records which refer to statement numbers as printed on the listing. |
>
> The two options are mutually exclusive. The default qualifier is /NODEBUG. /DEBUG without an option is equivalent to /DEBUG=STATEMENT_NUMBERS.

/LIST[=file-spec]
/NOLIST

By default, the compiler does not produce a listing. If /LIST is specified, the compiler produces a source listing file with the same name as the input source file but with a file type of "LIS". This may be overridden by giving a file-spec.

/MACHINE_CODE
/NOMACHINE_CODE

/MACHINE_CODE will cause the compiler to produce a symbolic, assembler-like listing on the listing file where it will follow the source listing. This listing is provided for information only and is not intended to be a complete assembly-language program. The default is /NOMACHINE_CODE.

/MODEL=(option,...)

This qualifier changes the compiler's model of the target machine. The possible options, which may appear in any order, are

CODE	generate separate code segments
DATA	generate separate data segments
STACK	generate separate data and stack segments
MEMORY	generate separate data and memory segments
ROM	put the constants with the code
P4	generate four-byte pointers
186	target machine is an 80186
286	target machine is an 80286

If the MODEL qualifier is not used, the compiler will generate the so-called *small* model. The other commonly used models may be specified as

/MODEL=(code p4)	*Medium* model
/MODEL=(stack,memory,p4)	*Compact* model
/MODEL=(code,data,stack,memory,rom,p4)	*Large* model

/OBJECT[=file-spec]
/NOOBJECT

Controls whether or not the compiler produces an object module. The default is /OBJECT which produces an object model that has the same file name as the source file and a file type of "Q86".

/OPTIMIZE[=(options)]
/NOOPTIMIZE

Controls whether or not the compiler optimizes the compiled program to generate more efficient code. The options, which may appear in any order, are

[NO]SUBEXPRESSIONS	specifies elimination of common subexpressions
[NO]AT_VARIABLES	specifies that uses of AT variables are to be optimized

[NO]BASED_VARIABLES	specifies that uses of BASED variables are to be optimized by assuming that no based store will modify *any* base pointer. Note that the compiler will *always* assume that a based store will not change its own base pointer.
[NO]JUMPS	specifies that the compiler is to attempt to remove dead-end and duplicate code sequences and to change long jumps to short jumps where possible
[NO]POINTERS	specifies that pointer comparisons are to be optimized by assuming that independent comparison of frames and offsets is sufficient to compare two pointers.
[NO]DANGEROUS	.ixDANGEROUS option .ixNODANGEROUS option specifies that various other optimizations are to be performed. These involve assumptions that may not always be valid and, thus, may lead to incorrect code.

The default is /OPTIMIZE=SUBEXPRESSIONS.

/SYNTAX

Specifies that the compiler is to perform syntax checking, only. Code generation will not be performed an an object module will not be produced.

2.2.2 Preprocessor Control Options

The normal action of the compiler preprocessor phase (Section 2.3 and Section 2.4) can be modified by:

/DEFINE=(name[=expression],...)

Defines each *name* as a compile-time variable and assigns it the value of the *expression* (or the value "-1" if an expression is not given). The first attempt to redefine the variable with a SET control (Section 2.4.3) will be ignored. The expression can be any valid compile-time expression (Section 2.4.1). The operands of the expression must be constants or the names of compile-time variables defined previously in the /DEFINE qualifier.

/INCLUDES=(directory,...)

Specify directories to be searched for an INCLUDE file (Section 2.4.2) if the file is not found in the directory of the source file. The directories are searched in the order given.

/PREPROCESS_ONLY
/NOPREPROCESS_ONLY

Specifies that the source file is not to be compiled but is to be run through the preprocessor with the output placed on the listing file. The default is /NOPREPROCESS_ONLY.

2.2.3 Completion Status

On completion, the compiler returns a standard VMS completion status of *success*, *warning*, or *severe/fatal*. See Appendix B for more detailed descriptions of these codes.

2.3 OVERALL OPERATION OF 86/PC

The 86/PC compiler consists of a driver, named "86pc", and a number of phases which perform the actual compilations. The phases used in a normal compilation, in the order executed, are:

- 86pp — The preprocessor which handles the compile-time control language described in Section 2.4.
- 86p1 — The initial syntax analyzer and declarations processor.
- 86p2 — The final syntax analyzer, semantics processor, and run-time storage allocator.
- 86pcg — The code generator.
- 86pjo — The optional jump optimizer.
- 86pifo — The final output generator for Intel object.
- 86ptfo — The final output generator for Tektronix object.
- 86pfo — The final output generator.
- 86psym — The symbolic lister.
- 86pxrf — The cross-reference lister.

2.4 THE 86/PC COMPILE-TIME CONTROL LANGUAGE

If the first character of a line is a "$", the line is known as a *control* line. This is true even if the line is within a comment or a quoted string. Such lines are used to request inclusion of additional source files and to control conditional compilation.

The general format of a control line is:

```
$ [control]...
```

where *control* has the general format:

```
keyword [operand]
```

The *keywords* are as described below and the operand format depends on the particular keyword. Letters appearing in keywords may be entered in either upper-case or lower-case.

2.4.1 Compile-Time Expressions

Several controls allow compile-time expressions in their operands. These are expressions which combine compile-time variables and numeric constants with the operators:

```
+ - * / NOT AND OR XOR < <= = <> >= >
```

The meaning of the operators and their precedence is the same as for other 86/PL expressions (Chapter 7) and parentheses may be used to modify the precedence.

2.4.1.1 Compile-Time Variables

Compile-time variables have the same form as other 86/PL identifiers (Rule 92). They contain signed, 16-bit quantities and are accessible only with compiler controls.

2.4.1.2 Compile-Time Constants

Compile-time constants have the form of a "number" (Rule 97) as described in Section 7.7. They can be represented in binary, octal, decimal, or hexadecimal notation.

2.4.2 The "INCLUDE" Control

The INCLUDE control has the form

```
$INCLUDE (path)
```

where *path* is a path to a file name. If necessary, the file is searched for first in the directory of the primary source file, and then in the directories given in the "-I" ("/INCLUDES") invocation option (Section 2.1.2, Section 2.2.2). If the file is found, it is included in the source at this point.

The INCLUDE control must be the rightmost control on a control line. Included files may, themselves, contain INCLUDE controls.

2.4.3 The "SET" Control

The SET control has the form

```
$SET (setspec[,setspec]...)
```

where each *setspec* has the form

```
compile-time-variable[=compile-time-expression]
```

The variable is defined (or redefined) to have the value of the expression. Any variables used in the expression must have been previously defined by a SET control or by the "-D" ("/DEFINE") invocation option (Section 2.1.2, Section 2.2.2). If the expression (and the equal sign) are absent, a value of *-1* will be assigned to the variable. The first $SET

of a variable that has been set by a "-D" ("/DEFINE") invocation option is ignored.

2.4.4 The "RESET" Control

```
$RESET (var[,var]...)
```

Each variable is defined (or redefined) to have the value of zero. The first $RESET of a variable that has been set by a "-D" ("/DEFINE") invocation option is ignored.

2.4.5 Conditional Compilation

Conditional compilation is performed by the controls described in this section. When used, each of these controls must appear alone on a control line.

The general form of a conditional compilation block is

```
$IF expression
...
...
...
$ELSEIF expression
...
...
...
$ELSEIF expression
...
...
...
$ELSE
...
...
...
$ENDIF
```

The ELSEIF and ELSE portions are optional and there can be a number of ELSEIF portions. Conditional compilation blocks may be nested.

An expression in the IF and ELSEIF controls is considered true if the low bit of its value is one; otherwise, it is considered *false*.

2.4.6 Listing Controls

The listing controls are TITLE, SUBTITLE, LIST, NOLIST and EJECT. The listing controls are ignored unless the "-l" ("/LIST"), "-a" ("/MACHINE_CODE"), or "-x" ("/CROSS_REFERENCE") options create a listing on the standard output.

The TITLE and SUBTITLE control have the form

```
$TITLE('string')
$SUBTITLE('string')
```

where string is a sequence of up to 60 ASCII characters.

More than one SUBTITLE is allowed. Any SUBTITLE control after the first causes a page eject.

The LIST, NOLIST and EJECT controls have the form

```
$LIST
$NOLIST
$EJECT
```

LIST resumes listing the source. NOLIST suppresses listing of the source. EJECT causes a page eject in the source listing.

2.4.7 Other Controls

All other controls are ignored by 80/PC so that source files intended for PL/M-80 can be processed by 80/PC without change.

2.4.8 Other Controls

All other controls are ignored by 86/PC so that source files intended for PL/M-86 can be processed by 86/PC without change.

2.5 86/PL SOURCE FORMAT

An 86/PL source program is composed of a sequence of *lines*, each of which must be ended by a newline character.

2.5.1 Blanks and Comments

A comment in 86/PL consists of a sequence of characters prefixed with the combination "/*" and suffixed with the combination "*/". The sequence of characters may not include the combination "*/".

A comment may be used wherever a blank is permitted, except within strings.

2.5.2 Statement Recognition

There are no reserved words in 86/PL. However, a statement beginning with one of the following statement keywords is assumed to be the statement which begins with that word:

DO	IF	PROCEDURE	ENABLE
END	ELSEIF	DECLARE	DISABLE
GO	ELSE	CALL	HALT
GOTO	ENDIF	RETURN	CAUSEINTERRUPT
UNDO			

2.6 OBJECT MODULE FORMAT

The object module produced by the 86/PC compiler uses the format of the Intel MCS-8086 Relocatable Object Module Formats. Some versions of the compiler are optionally able to produce Tektronix LAS format object modules.

2.7 RUN-TIME SUPPORT

The only external calls generated by 86/PC are for multiply, divide, and MOD for the DWORD data type. The compiler uses LQ_DWORD_MUL for multiply and LQ_DWORD_DIV for both divide and MOD. The user must supply LQ_DWORD_MUL and LQ_DWORD_DIV when the compiler generates calls to them. See Appendix D for example listings.

3. Introduction to the Meta-Language

This manual presents the complete syntax for the 86/PL language using a formal meta-language. The syntax is permissive in that some constructs that are formally allowed by the syntax are disallowed in practice as described in the text of this manual.

The meta-language used to describe the syntax of 86/PL is a modification of BNF. It is described informally in this chapter and formally in Appendix C.

A *grammar* in the meta-language consists of a sequence of *rules*, each terminated by a period.

Non-terminal symbols are composed of letters, decimal digits, and dashes. *Literals* are represented as quoted strings or as a sequence of upper-case letters. Within a literal, an upper-case letter and the corresponding lower-case letter are considered equivalent.

Each rule of grammar has the general form:

    v = s1 | s2 | ... | sn.

where *v* is a non-terminal symbol and the *s*'s are arbitrary strings of non-terminal symbols and literals. The interpretation of such a rule is that *v* is to be replaced by one of the alternatives *s1*, or *s2*, or ...or *sn*.

This form can be extended by a number of simplifying constructs:

1. A rule such as

        a = b ';' | c ';'.

   may be written as

        a = { b | c } ';'.

   In general,

        v = s1 t1 s2 | s1 t2 s2 | ... | s1 tn s2.

   (where the *t*'s are non-null strings of non-terminal symbols and literals) may be replaced by

        v = s1 { t1 | t2 | ...| tn } s2.

2. A rule such as

        a = b | b c.

   may be written as

a = b [c].

in general,

v = s1 s2 | s1 t1 s2 | ...| s1 tn s2.

may be replaced by

v = s1 [t1 | t2 | ...| tn] s2.

3. A rule such as

a = b | a b.

may be written as

a = b*.

which may be read as "a is to be replaced by one or more occurrences of b". For convenience, the sequence

[t*]

may be replaced by

[t]*

and the sequence

[{ t1 | t2 | ...| tn} *]

may be replaced by

[t1 | t2 | ...| tn]*

4. Modules and Procedures

The 86/PL unit of compilation is known as a *module*. It may contain declarations, procedures, and possibly a main program. Modules, procedures, and main programs are discussed in this chapter.

4.1 MODULE DEFINITIONS

The syntax of a module is:

1. *module* = *identifier* ':' *DO* ';' *module-body ending.*

2. *module-body* = *[declare-statement | procedure-declaration]** *[executable-statement]*.*

The statements from the DO statement through the ending are within a new naming scope. This naming scope is the module level and many concepts such as initial data, public and external data, and reentrant and interrupt procedures can only be used in statements at the module level.

The identifier is the *module name* and is used to name the resulting object module. If a name appears in the ending (Rule 67) of the module, the compiler will verify that it is the module name.

A source file may contain only one module.

4.2 MAIN PROGRAMS

If the module body contains executable statements (Rule 38) not contained within procedures (Rule 3), the module is a *main program*. The entry point of a main program is the first executable statement in the module body. All executable statements except the RETURN statement (Rule 61) may appear in a main program. A HALT statement (Rule 65) implicitly follows the last statement of a main program.

4.2.1 Main Program Statement Labels

Statement labels (Rule 68) in a main program differ in three ways from statement labels in procedures:

- they may be declared PUBLIC;
- they generate code to reinitialize the stack pointer and the stack base registers; and
- they may be the target of a GOTO statement (Rule 59) from outside the main program.

4.3 PROCEDURE DECLARATIONS

The syntax of a procedure declaration is:

3. *procedure-declaration* = *identifier* ':' *procedure-head*
   procedure-body ending.

4. *procedure-body* = [*declare-statement* | *procedure-declaration*]*
   [*executable-statement*]*.

5. *procedure-head* = PROCEDURE [*parameter-list*]
   [*procedure-attribute*]* ';'.

6. *procedure-attribute* = *basic-type* | *procedure-scope* |
   procedure-class.

The statements from the procedure head through the ending are within a new naming scope.

If a name appears in the ending (Rule 67) of the procedure, the compiler will verify that the named procedure is the one being ended.

4.4 PROCEDURE PARAMETERS

The syntax of the optional procedure parameter list is:

7. *parameter-list* = '(' *identifier* [',' *identifier*]* ')'.

Procedure parameters appear in the parameter list and then in declare statements (Rule 10) which must appear among the declare statements in the procedure body. The declare statements for procedure parameters give a basic type (Rule 33) and no other attributes.

4.5 PROCEDURE TYPES

Procedures are either untyped or have one of the basic types (Rule 33).

Untyped procedures are frequently referred to as *subroutines*. They are invoked by a CALL statement (Rule 58). The return from an untyped procedure is a RETURN statement without the optional expression. Such a return implicitly follows the last statement of an untyped procedure.

Typed procedures are frequently referred to as *functions*. They are invoked by a function reference (Rule 86) within an expression. The return from a typed procedure is a RETURN statement with an expression compatible with the type of the procedure. If the return does not have the optional expression, a warning is issued.

Unless the end of the function is preceded by a GOTO statement or a RETURN statement, a return without an expression is generated.

4.6 PROCEDURE SCOPE

The syntax of the procedure scope attributes is:

8. *procedure-scope* = *EXTERNAL* | *PUBLIC*.

Procedures which are neither external nor public are internal. A single procedure definition can not have both the EXTERNAL and PUBLIC attributes.

4.6.1 Public and Internal Procedures

Procedures with the PUBLIC attribute or with no scope attribute are actual procedures and should have at least one executable statement (Rule 38) in their procedure bodies.

4.6.2 External Procedures

Procedures with the EXTERNAL attribute are dummy procedures that declare the procedure type and parameters. The procedure bodies of external procedures may contain only declarations for the procedure parameters.

External procedures may appear only at the module level. Like all external declarations, external procedures may be redeclared as actual procedures at the module level. No compatibility checks are made when a procedure is redeclared in this way.

4.7 PROCEDURE CLASS

The syntax of the procedure class attributes is:

9. *procedure-class* = *REENTRANT* | *INTERRUPT [number]*.

A procedure may have both the interrupt and the reentrant attributes.

4.7.1 Reentrant Procedures

Procedures with the REENTRANT attribute have their local storage allocated on the stack. This allows more than one activation of the reentrant procedure to execute concurrently. Reentrant procedures must be at the module level and may not have other procedures nested within them.

4.7.2 Interrupt Procedures

Procedures with the INTERRUPT attribute can be invoked from the processor interrupt vector of the machine. The prologue of an interrupt procedure disables interrupts and stores all the processor registers. The epilogue restores all the registers, enables interrupts, and returns. Interrupt procedures are untyped and have no parameters.

If the optional number is used, an interrupt vector is generated by the compiler.

5. DECLARE Statements

The syntax of the DECLARE statement is:

10. *declare-statement* = DECLARE *declaration-list* ';'.

11. *declaration-list* = *declaration-item* [',' *declaration-item*]*.

12. *declaration-item* = *identifier any-attribute** | *factored-declaration-item*.

13. *any-attribute* = *label-attribute* | *literally-attribute* | *at-attribute* | *initialization-attribute* | *element-attribute*.

Data items, labels, and literallys are declared by means of the DECLARE statement. All identifiers except procedure names, labels, and the predefined array, MEMORY, must be declared in a DECLARE statement before they are used.

5.1 FACTORED DECLARATIONS

The syntax of a factored declaration item is:

14. *factored-declaration-item* = '(' *basic-declaration* [',' *basic-declaration*]* ')' *any-attribute**].

15. *basic-declaration* = *identifier* [*element-attribute*]*.

Factoring of declarations has two purposes – convenience, and forcing contiguous allocation of storage. When items in the factored list are allocated storage, that storage will be contiguous and in the order of the items in the list.

5.2 THE LABEL ATTRIBUTE

The LABEL attribute is:

16. *label-attribute* = LABEL.

The LABEL attribute declares an identifier to be a label. When an identifier is used in a label definition (Rule 68) it is implicitly declared, so explicit label declaration is not normally needed. However, the label declaration is needed to associate the PUBLIC and EXTERNAL attributes with a label. The LABEL attribute must be factored if any attributes are factored. The LABEL attribute is incompatible with any attributes except PUBLIC and EXTERNAL.

5.3 THE LITERALLY ATTRIBUTE

The syntax of the LITERALLY attribute is:

17. *literal-attribute* = *LITERALLY string.*

An identifier declared with the LITERALLY attribute is actually a parameterless macro. Whenever the compiler encounters an identifier declared with this attribute, the associated string (Rule 95) is substituted for the identifier. Since the compiler resumes its scan from the beginning of the substituted string, that string may also contain identifiers declared with the LITERALLY attribute.

The LITERALLY attribute is not compatible with any other attribute.

5.4 THE AT ATTRIBUTE

The syntax of the AT attribute is:

18. *at-attribute* = *AT '(' restricted-expression ')'.*

The AT attribute can only be applied to variables which are not based or external. The restricted expression (Rule 25) cannot be the address of a procedure or label. If the restricted expression gives the address of an external variable, then the variable with the AT attribute cannot be public. If the AT attribute is applied to a factored list, the first variable is placed at the location given by the restricted expression and the other variables follow. A variable with the AT attribute is assumed to have a new value every time it is referenced so it will not normally be optimized. However, the *a* setting of the -O invocation switch (Section 2.1.1), or the *AT* option of the */OPTIMIZE* qualifier (Section 2.2.1), allow optimization of AT variables.

5.5 THE DATA AND INITIAL ATTRIBUTES

The syntax of the DATA and INITIAL attributes is:

19. *initialization-attribute* = *data-attribute* | *initial-attribute.*

20. *data-attribute* = *DATA ['(' initialization-item-list ')'].*

21. *initial-attribute* = *INITIAL '(' initialization-item-list ')'.*

22. *initialization-item-list* = *initialization-item [',' initialization-item]*.*

23. *initialization-item* = *string* | *signed-constant* | *restricted-expression* | *function-reference.*

The DATA attribute without an initialization item list can only be applied to external variables. It is needed when the external item is allocated in the code segment. Initialization attributes can only be applied to variables which are not based (Rule 31) or external. If the initialization attribute is applied to a factored list, the initialization items are used to fill the variables in the list until they are used up.

If the initialization item is a string (Rule 95), each element is filled with the next bytes in the string. For instance, if the current element is a word, the next two bytes will be used to fill the element. When the string contains too few bytes to fill the element, those that remain will be placed left adjusted in the element.

The DATA attribute specifies that the variable cannot be changed during execution. When separate data or ROM is specified with the -*M* option, or with the */MODEL* qualifier,

variables with the DATA attribute are allocated in the code segment. The INITIAL attribute specifies that the variable is assigned to the data segment and can be changed at execution. The INITIAL attribute can only appear at the module level (Section 4.1).

The only functions that may be used in an initialization are the builtin functions that give constant values (Section 7.3) and the builtin functions INTERRUPT$PTR, SELECTOR$OF, and OFFSET$OF.

5.5.1 Signed Constants

A signed constant has the syntax:

24. *signed-constant = ['+' | '-']* number.*

Signed constants are used to initialize dword and real variables.

5.5.2 Restricted Expressions

A restricted expression has the syntax:

25. *restricted-expression = address [{ '+' | '-'} expression] |*
    expression.

Restricted expressions are used in the AT attribute, DATA attribute, and the INITIAL attribute. The expression must have the form of a constant operand (Section 7.3). The address (Rule 81) may be the address of a long constant (Rule 82).

5.6 ELEMENT ATTRIBUTES

The syntax of an element attribute is:

26. *element-attribute = storage-class | member-attribute.*

27. *storage-class = public-attribute | external-attribute | based-attribute.*

28. *member-attribute = dimension | structure | basic-type.*

5.6.1 The EXTERNAL Attribute

The EXTERNAL attribute is:

29. *external-attribute = EXTERNAL.*

The EXTERNAL attribute can only be used at the module level.

The scope of externals is between that of builtin names and module level names. This means that external names can be redefined by declarations at the module level. A program made up of many modules can therefore have a definition file containing external definitions for all identifiers shared between the modules. This file can be included in all the modules, even one where there is a corresponding public definition, without causing an error.

5.6.2 The PUBLIC Attribute

The PUBLIC attribute is:

30. *public-attribute = PUBLIC.*

The PUBLIC attribute makes an identifier available to other modules. The PUBLIC attribute can be used only at the module level. An identifier that is declared EXTERNAL or AT an external cannot have the PUBLIC attribute.

5.6.3 The BASED Attribute

The syntax of the based attribute is:

31. *based-attribute* = *BASED* { *restricted-reference* | '*'} .

The BASED attribute specifies that the declared item is located at the address given by its base. No storage is allocated for based items.

The restricted reference (Rule 89) gives an implied base which must be a previously defined word, dword, pointer, or selector. The implied base will be used when an actual reference to the based item does not have an explicit base (Rule 91).

An implied base of '*' means that the variable must always be referenced with an explicit base.

5.6.4 The Dimension Attribute

The syntax of the dimension attribute is:

32. *dimension* = '(' { *number* | '*'} ')'.

The dimension attribute specifies that the declared item is an array and usually gives the number of elements in the array.

A dimension of '*' is legal if the identifier is external or based or if it has an unfactored INITIAL attribute or DATA attribute. The actual dimension of an external or based array is unimportant. If an array is initialized, the value of the '*' is set to the the number of items in the initialization list. If an item in the initial list is a string, the dimension will be made large enough to hold the bytes in the string. For instance, if the string is five bytes and the array is a word array, three words will be allocated in the array to hold the string.

Note that the dimension attribute need not immediately follow the dimensioned identifier.

5.6.5 The Basic Type Attributes

The basic types are:

33. *basic-type* = *BYTE* | *WORD* | *ADDRESS* | *DWORD* | *INTEGER* |
    REAL | *POINTER* | *SELECTOR*.

. The BYTE type specifies an unsigned number of 8 bits. This is a number from 0 to 255.

The WORD type specifies an unsigned number of 16 bits. This is a number from 0 to 65535

The ADDRESS type is exactly the same as the WORD type.

The DWORD data type specifies an unsigned number of 32 bits. This is a number from 0 to 4294967295.

The INTEGER type specifies a signed number of 16 bits. This is a number from -32768 to 32767.

The REAL data type specifies a real number of 32 bits. Real numbers are in the 8087 short-real format.

The POINTER type specifies an address made up of a 16-bit offset and a 16-bit base. A pointer is, therefore, normally 32 bits long. When all data is addressable from the *data* segment register, a pointer can be just the 16-bit offset. This is controlled by the -M invocation option or the /MODEL qualifier.

The SELECTOR data type ELECTOR data type specifies an 8086 paragraph number and is 16 bits long.

5.6.6 The STRUCTURE Attribute

The syntax of the STRUCTURE attribute is:

34. *structure* = *STRUCTURE* '(' *member-list* ')'.

35. *member-list* = *member* [',' *member*]*.

36. *member* = *member-identifier member-attribute** |
    '(' *member-identifier* [',' *member-identifier*]* ')'
    *member-attribute**.

37. *member-identifier* = *identifier* | '*'.

If the member identifier is a '*', an unnamed space is left in the structure. The size of this space is determined by the member attributes (Rule 28).

6. Executable Statements

The syntax of an executable statement is:

38. *executable-statement* = *do-group* | *if-statement* | *if-block* | *simple-statement*.

6.1 DO GROUPS

The syntax of a DO group is:

39. *do-group* = *group-head-statement [declaration]** *[undo-statement* | *executable statement]** *ending*.

40. *group-head-statement* = *[label-definition]* { *do-statement* | *while-statement* | *iterative-do-statement* | *case-statement* } .

The statements from the group head through the ending are within a new naming scope.

Any group may be prefixed with a label which gives the group name to be referred to in an UNDO statement or an END statement. If the label definition (Rule 68) preceding a group contains multiple labels, the last one is the group name.

If the group name is used in an END statement, the compiler will verify that the named group is, in fact, the one being closed.

6.1.1 The DO Statement

The syntax of the DO statement is:

41. *do-statement* = DO ';'.

The DO statement initiates a group but serves no other purpose.

6.1.2 The WHILE Statement

The syntax of the WHILE statement is:

42. *while-statement* = DO WHILE *conditional-expression* ';'.

The WHILE statement initiates a group. The statements within the group are executed repeatedly while the conditional expression (Rule 69) remains true. The test takes place before each execution of the statements within the group.

6.1.3 The Iterative DO Statement

The syntax of the iterative DO statement is:

43. *iterative-do-statement* = *DO restricted-reference '=' expression-1*
    TO expression-2 [BY expression-3] ';'.

This statement initiates a group that is an iterative loop. The restricted reference (Rule 89) is the loop index. The expressions (Rule 70) must have a type compatible with the type of the loop index. If the optional BY clause is absent, expression-3 is assumed to be the constant 1.

The loop index may be either integer or unsigned (byte or word). Integer and unsigned iterative loops operate in a significantly different manner. For both types of iterative loops, the start, expression-1, is first evaluated and assigned to the loop index.

For an unsigned iterative loop, the limit, expression-2, is evaluated before each iteration of the loop. If the loop index exceeds the limit, the loop is terminated. The step expression is evaluated after each iteration of the loop and is added to the loop index. If there is an overflow, the loop is terminated.

For an integer iterative loop, both the limit and the step expressions are evaluated before each iteration of the loop. If the step is positive the loop is terminated if the loop index is greater than the limit. If the step is negative the loop is terminated if the loop index is less than the limit. After each iteration of the loop the step which was evaluated at the beginning of the loop is added to the loop index.

6.1.4 The CASE Statement

The syntax of the CASE statement is:

44. *case-statement* = *DO CASE expression ';'.*

In operation, consider that each statement in the body of the group is numbered sequentially from zero. The expression is evaluated and the correspondingly numbered statement is executed. Control is then transferred to the statement following the end of the group.

If the value of the expression is negative or greater than the number of statements in the body of the group minus one, the results are unpredictable.

6.1.5 The UNDO Statement

The syntax of the UNDO statement is:

45. *undo-statement* = *[label-definition] UNDO [identifier] ';'.*

If the identifier is absent, control passes out of the immediately containing DO group. If the identifier is present, control passes out of the containing DO group with the corresponding name.

6.2 THE IF STATEMENT

The syntax of the IF statement is:

46. *if-statement* = *if-clause executable-statement | if-clause*
    balanced-statement ELSE executable-statement.

47. *if-clause* = *[label-definition] IF conditional-expression THEN.*

48. *balanced-statement* = *if-clause balanced-statement ELSE balanced-statement* | { *if-block* | *do-group* | *simple-statement*} .

Note that the IF statement itself is not ended by a semicolon. If the conditional expression (Rule 69) is true, the executable statement following the THEN is executed and, if there is an ELSE, the executable statement following the ELSE is not executed. If the conditional expression is false, the executable statement following the THEN is skipped and, if there is an ELSE, the executable statement following the ELSE is executed.

6.3 IF BLOCKS

The syntax of an IF block is:

49. *if-block* = *block-if-statement [executable-statement]* [block-elseif]* [block-else] endif-statement.*

50. *block-elseif* = *elseif-statement [executable-statement]*.*

51. *block-else* = *else-statement [executable-statement]*.*

The IF block provides the same capability as the IF statement but does so with separate statements as the block delimiters. This use of statements as the block delimiters is like the use of the DO and END statements as group delimiters. However, an IF block does not create a new naming scope.

6.3.1 Block If Statement

The syntax of the block IF statement is:

52. *block-if-statement* = *[label-definition] IF conditional-expression ';'.*

The block IF statement introduces an IF block. Note that this statement has a semicolon in the place that an IF statement would have a THEN.

If the conditional expression (Rule 69) is false, control passes to the following block delimiter for this IF block. The following block delimiter may be an ELSEIF statement, an ELSE statement, or an ENDIF statement.

If the conditional expression is true, control falls through to the following statements which are executed up to the following block delimiter. Control then passes to the ENDIF statement for this IF block.

6.3.2 The ELSEIF Statement

The syntax of the ELSEIF statement is:

53. *elseif* = *[label-definition] ELSEIF conditional-expression ';'.*

Note that an ELSEIF statement is only legal within an IF block.

If the conditional expression is false, control passes to the following block delimiter for this IF block. The following block delimiter may be an ELSEIF statement, an ELSE statement, or an ENDIF statement.

If the conditional expression is true, control falls through to the following statements which are executed up to the following block delimiter. Control then passes to the ENDIF statement for this IF block.

6.3.3 The ELSE Statement

The syntax for the ELSE statement is:

54. *else-statement* = *[label-definition] ELSE ';'.*

Note that an ELSE statement is only legal within an IF block and is not the same thing as the ELSE keyword in the IF statement.

If control reaches the ELSE statement the following statements are executed up to the ENDIF statement for this IF block. Control then passes to the ENDIF statement for this IF block.

6.3.4 The ENDIF Statement

The syntax of the ENDIF statement is:

55. *endif statement* = *[label-definition] ENDIF ';'.*

Note that an ENDIF statement is only legal within an IF block. Control passes from the ENDIF statement to the statements following the IF block.

6.4 SIMPLE STATEMENTS

The syntax of simple statements is:

56. *simple-statement* = *assignment-statement | call-statement |*
goto-statement | null-statement |
return-statement | special-statement.

6.4.1 The Assignment Statement

The syntax of the assignment statement is:

57. *assignment-statement* = *[label-definition] target-reference*
[',' target-reference] '=' expression ';'.*

The right side expression (Rule 70) is assigned to all the target references. The types of the target references must be compatible with each other and with the right side expression.

6.4.2 The CALL Statement

The syntax of the CALL statement is:

58. *call-statement* = *[label-definition] CALL { identifier |*
restricted-reference} ['(' expression-list ')'] ';'.

If the identifier form is used, the call is a direct call. The identifier must be the name of an untyped procedure (Section 4.5). If the restricted reference form is used, the call is an indirect call and the restricted reference must contain the address of an untyped procedure. The restricted reference (Rule 89) must be to a word or pointer variable.

The expression list supplies arguments to the procedure. All arguments are passed by value. If the call is direct, the argument expressions must match the parameters of the procedure declaration in number and the expression types must be compatible. If the call is indirect, the arguments are assumed to match, in number and type, the parameters of the called procedure.

6.4.3 The GOTO Statement

The syntax of the GOTO statement is:

59. *goto-statement* = *[label-definition]* { *GOTO* | *GO TO*} *identifier* ';'.

The GOTO statement performs an unconditional transfer to a label. The identifier must be a label in the procedure containing the GOTO statement or a label in a main program (Section 4.2). The identifier must also be in the same or an enclosing naming scope.

A transfer to a label in a main program resets the stack pointer and stack base.

6.4.4 The Null Statement

The syntax of the null statement is:

60. *null-statement* = *[label-definition]* ';'.

The null statement performs no operation whatsoever. However, it is counted as a statement and, thus, may be found useful in DO CASE groups (Rule 44) and after the THEN or ELSE keywords in IF statements (Rule 46).

6.4.5 The RETURN Statement

The syntax of the RETURN statement is:

61. *return-statement* = *[label-definition] RETURN [expression]* ';'.

The form of the RETURN statement with the optional expression (Rule 70) is used to return from a typed procedure. A typed procedure should logically end with such a return. The expression must be compatible with the type of the procedure.

The form without the expression is used to return from an untyped procedure. An untyped procedure implicitly ends with such a return, but may contain other such returns.

6.4.6 Special Statements

The syntax of the special statements is:

62. *special-statement* = *disable-statement* | *enable-statement* | *halt-statement* | *cause-interrupt-statement*.

63. *disable-statement* = *[label-definition] DISABLE* ';'.

64. *enable-statement* = *[label-definition] ENABLE* ';'.

The DISABLE statement and the ENABLE statement generate the equivalent machine instructions.

65. *halt-statement* = *[label-definition] HALT* ';'.

The HALT statement generates an ENABLE and then a HALT instruction.

66. *cause-interrupt-statement* = *[label-definition] CAUSE$INTERRUPT* '(' *expression* ')' ';'.

The CAUSE$INTERRUPT statement generates an interrupt (INT) instruction. The expression must be a constant operand (Section 7.3) from zero to seven (0-255).

6.5 ENDINGS

The syntax of an ending is:

67. *ending = [label-definition] END [identifier] ';'.*

Endings are used to close modules, procedures, and DO groups. If the identifier appears in the ending, it will be used to verify that the named module, procedure or group is the one being closed.

6.6 LABEL DEFINITIONS

The syntax of label definitions is:

68. *label-definition = identifier ':'*.*

Only executable statements may have statement labels.

6.7 COMPATIBLE TYPES

In assignment statements (Rule 57) and other similar situations, the right side must have a type that can be converted into the type of the left side. Bytes, words and dwords are compatible: bytes and words are converted to words or dwords by extending them with zeros; words and dwords are converted to bytes or words by truncation.

Words and dwords are compatible with pointers: words and dwords are converted to pointers by setting the pointer offset to the word or the least significant word of the dword and the pointer base to the current data segment. Selectors are compatible with pointers: selectors are converted to pointers by setting the base of the pointer to the selector and the offset of the pointer to zero. Pointers are compatible with pointers. When the right side is a constant then the constant represents a 20 bit 8086 memory address.

Selectors are only compatible with selectors. When the right side is a constant, the constant represents an 8086 paragraph number. Integers are only compatible with integers. When the right side is a constant, the context of the constant operand is integer. Reals are only compatible with reals and with real constants.

6.8 CONDITIONAL EXPRESSION

The syntax of a conditional expression is:

69. *conditional-expression = expression.*

Even though a conditional expression has the same syntax as an expression, it may not be evaluated in the same way.

The expression is treated as if it were made up of simpler expressions connected by the AND, OR and NOT operators. The simpler expressions are evaluated only until the truth of the expression is determined.

The statements which use conditional expressions check only the least significant bit of an expression for true (1) or false (0).

7. Expressions

The syntax of an expression is:

70. *expression* = *basic-expression* | *embedded-assignment.*

71. *basic-expression* = *logical-factor* [{ OR | XOR } *logical-factor*]*.

72. *logical-factor* = *logical-secondary* [AND *logical-secondary*].

73. *logical-secondary* = [NOT]* *logical-primary.*

74. *logical-primary* = *sum* [*relop sum*].

75. *relop* = ‘<’ | ‘<=’ | ‘=’ | ‘<>’ | ‘>=’ | ‘>’.

76. *sum* = *term* [{ ‘+’ | ‘–’ | PLUS | MINUS} *term*]*.

77. *term* = *secondary* [{ ‘*’ | ‘/’ | MOD} *secondary*]*.

78. *secondary* = [‘+’ | ‘–’]* *primary.*

79. *primary* = *constant* | *address* | *reference* | ‘(’ *expression* ‘)’.

Note that by the rule for logical primary, a sequence such as X < Y < Z is not legal since the “relop sum” sequence cannot be repeated.

7.1 OPERATORS

The table below gives the operators recognized in expressions. The table is ordered from the highest operator precedence to the lowest with groups of operators with the same precedence on consecutive lines. The highest precedence operators are those executed first. The columns labelled "b w d i r ps" (byte, word, dword, integer, real, pointer or selector) shows which operands are legal for each operator, and gives the type of the result. A “–” means that operands of that type are not legal.

op	b	w	d	i	r	ps	name
+	b	w	d	i	r	–	unary plus
-	b	w	d	i	r	–	unary minus (negation)
*	w	w	d	i	r	–	multiplication
/	w	w	d	i	r	–	division
MOD	w	w	d	i	–	–	remainder
+	b	w	d	i	r	–	addition
-	b	w	d	i	r	–	subtraction
PLUS	b	w	d	–	–	–	add with carry
MINUS	b	w	d	–	–	–	subtract with carry
<	b	b	b	b	b	b	less than
<=	b	b	b	b	b	b	less than or equal to
=	b	b	b	b	b	b	equal to
<>	b	b	b	b	b	b	not equal to
>=	b	b	b	b	b	b	greater than or equal to
>	b	b	b	b	b	b	greater than
NOT	b	w	d	–	–	–	bitwise NOT (one's complement)
AND	b	w	d	–	–	–	bitwise AND
OR	b	w	d	–	–	–	bitwise OR
XOR	b	w	d	–	–	–	bitwise exclusive OR

The operand type for binary operators is determined by combining the types of the two operands as shown by the next table. As before, a "–" means that the combination is not legal.

	b	w	d	i	r	p	s
byte	b	w	d	–	–	–	–
word	w	w	d	–	–	–	–
dword	w	w	d	–	–	–	–
integer	–	–	–	i	–	–	–
real	–	–	–	–	r	–	–
pointer	–	–	–	–	–	p	–
selector	–	–	–	–	–	–	s

7.2 RELATIONAL OPERATORS

The relational operators

< <= = <> >= >

compare all the legal combinations shown in the above tables. The result of the relational operation is a byte with the value true (0FFh) or *false* (0).

7.3 CONSTANT OPERANDS

A constant operand is

- a constant;

- the builtin functions SIZE, LENGTH, or LAST;
- the builtin functions LOW, HIGH, UNSIGN, and INT when their operand is a constant;
- the builtin function DOUBLE when its operand is a byte constant; or
- an operation with constant operands.

The type of a constant operand depends on its context.

If a constant operand is used in a place where only an integer would be legal, the value of the constant operand is treated as if it were an integer. When the integer operand is a number, its value must be 0 to 32767. If the integer operand is really an operation between constant operands, then the operation becomes an integer operation and the context of its operands is integer.

7.4 EMBEDDED ASSIGNMENTS

The syntax of an embedded assignment is:

80. *embedded-assignment* = *target-reference* ':=' *basic-expression.*

The type of an embedded assignment is the type of the basic expression. The type of the target-reference must be compatible (Section 6.7) with the type of the basic expression.

7.5 ADDRESSES

The syntax of an address is:

81. *address* = {'@'|'.'} {*inexact-reference* | *long-constant*}.

82. *long-constant* = '(' expression [',' expression]* ')'.

A long constant acts like a DATA initialization (Rule 20) of a byte array.

7.6 REFERENCES

The syntax of a reference is:

83. *reference* = *basic-reference* | *explicit-based-reference* | *function-reference.*

84. *basic-reference* = *elementary-reference* ['.' *elementary-reference*]*.

85. *elementary-reference* = *identifier* ['(' *expression* ')'].

7.6.1 Function Reference

The syntax of a function reference is almost like the syntax of an elementary reference.

86. *function-reference* = *identifier* ['(' *expression-list* ')'].

87. *expression-list* = *expression* [',' *expression*]*.

The identifier is the name of a typed procedure and the expressions in the expression list are the arguments. The arguments must be compatible with the formal parameters of the function.

7.6.2 Assignment Target Reference

The syntax of a target reference is:

88. *target-reference* = *reference.*

If the target reference is a function reference (Rule 86), the function can only be one of the builtin pseudo-functions:

OUTPUT STACKPTR LOW HIGH

STACKBASE OFFSET$OF SELECTOR$OF

See Chapter 8 for further information.

7.6.3 Restricted Reference

The syntax of a restricted reference is:

89. *restricted-reference* = *identifier ['.' identifier]*.*

Restricted references appear in the BASED attribute, the iterative DO statement and the indirect CALL statement.

7.6.4 Inexact Reference

The syntax of an inexact reference is just like the syntax of a basic reference:

90. *inexact-reference* = *basic-reference.*

An inexact reference can be a reference to an array or to a structure as well as to a simple variable. An inexact reference to a member of an array of structures need not have an index for the structure. If the index is missing, it is assumed to be zero. Inexact references appear in addresses (Rule 81) and in the SIZE, LENGTH, and LAST builtin functions (Chapter 8).

7.6.5 Explicitly Based Reference

The syntax of an explicitly based reference is:

91. *explicit-based-reference* = { *basic-reference* |
    '(' expression ')'} { *'->' basic-reference*} *.

Each basic reference following the arrow must be to a variable with the BASED attribute (Rule 31). If the based item was declared with an implicit base, that base is ignored.

The base itself cand be a word, dword, pointer, or selector. When the base expression is an absolute number, that number is treated as a pointer constant. A variable based on an absolute is like a variable AT an absolute (Rule 18). However, it is optimized like any other based reference.

7.6.6 Identifiers

An identifier is:

92. *identifier* = *letter [letter* | *decimal-digit* | *'_'* | *'$']*.*

93. *letter* = *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | K | L | M | N | O |
    P | Q | R | S | T | U | V | W | X | Y | Z.

An identifier may consist of a maximum of 31 characters. Dollar signs ("$"), which may be used freely for readability, are not saved as part of the identifier and do not count toward the maximum.

An upper-case letter and its lower-case form are considered equivalent.

7.7 CONSTANTS

A constant is:

94. *constant* = *string* | *number*.

95. *string* = '' *string-character** ''.

96. *string-character* = '''' | *printing-character-other-than-apostrophe*.

97. *number* = *binary-number* | *octal-number* | *decimal-number* | *hex-number* | *floating-point-number*.

98. *binary-number* = *binary-digit [binary-digit*|'$']* B.

99. *binary-digit* = 0 | 1.

100. *octal-number* = *octal-digit [octal-digit*|'$']* {O|Q}.

101. *octal-digit* = *binary-digit* | 2 | 3 | 4 | 5 | 6 | 7.

102. *decimal-number* = *decimal-digit [decimal-digit*|'$']* [D].

103. *decimal-digit* = *octal-digit* | 8 | 9.

104. *hex-number* = *decimal-digit [hex-digit*|'$'] H.

105. *hex-digit* = *decimal-digit* | A | B | C | D | E | F.

106. *floating-point-number* = *decimal-number* '.' *[decimal-number] [E[+|-]decimal-number]*.

Dollar signs ("$") may be freely used between digits in numbers for readability.

8. Builtin Identifiers and Functions

The compiler recognizes builtin identifiers that are equivalent to typed procedures, untyped procedures and pseudo-functions. Pseudo-functions are like untyped procedures but appear on the left of assignment statements. The compiler also recognized the builtin array variable MEMORY.

In the following description, the builtin identifiers are given as they would appear in the context of simple assignment or call statements:

- Untyped procedures appear in call statements.
- Typed procedures appear on the right of an assignment and the left indicates the result expected from the procedure.
- Pseudo-functions appear on the left of an assignment statement and the right gives the type of the values that can be assigned to them.

Most builtin procedures and pseudo-functions have arguments. Arguments can generally be expressions. The few exceptions are constants or references. Where arguments are described as byte or word, either byte, dword, or word expressions may be used since a word or dword can always be truncated to a byte and a byte can be extended with zeros to be a word or dword.

8.1 SIZE OF VARIABLES

LENGTH, LAST and SIZE are functions that yield constants. The constants are like numbers in that values from 0 - 255 have type byte and values from 256 - 65535 have type word. Their argument has the syntax of an inexact reference (Rule 90).

```
con = SIZE (ref)
con = LENGTH (ref)
con = LAST (ref)
```

The SIZE function gives the size in bytes of the referenced item.

The LENGTH function gives the number of elements in the referenced item. If it is not an array then the length is one.

The LAST function gives the index of the last element in the referenced item. If it is not an array then the index is zero.

8.2 TYPE CONVERSION

The INT, SIGNED, DOUBLE, UNSIGN, FIX, and FLOAT functions change the type of their argument.

```
integer = INT (word)
integer = SIGNED (word)
word = DOUBLE (byte)
dword = DOUBLE (word)
word = UNSIGN (integer)
integer = FIX (real)
real = FLOAT (integer)
```

8.3 SHIFT AND ROTATE

The *bits* argument and *count* argument may be either bytes or words. The result type will be of the same type as the first argument.

```
bits = SHL (bits,count)
bits = SHR (bits,count)
bits = ROL (bits,count)
bits = ROR (bits,count)
bits = SCL (bits,count)
bits = SCR (bits,count)
integer = SAL (integer,count)
integer = SAR (integer,count)
```

SHL and SHR shift bytes,words, or dwords. Bits shifted out go into the carry. Zeroes are shifted in.

ROL and ROR rotate bytes, words, or dwords. Bits rotated out go into both the other end of the item and into the carry.

SCL and SCR also rotate bytes, words, or words but they include the carry in the bits rotated. The bits shifted out of the item go into the carry and the bits shifted out of the carry go into the other end of the item.

SAR shifts an integer to the right. Bits shifted out go into the carry. Bits shifted in are the same as the sign bit.

SAL shifts an integer to the left. It operates just like SHL.

8.4 REFERENCING SUBFIELDS

HIGH and LOW reference the two bytes of a word or the two words of a dword. SELECTOR$OF and OFFSET$OF reference the base and offset parts of a pointer. HIGH, LOW, SELECTOR$OF and OFFSET$OF can be used as a normal functions or as pseudo-functions. When HIGH or LOW is used as a pseudo-function, its argument must be a reference to a word or dword variable. When SELECTOR$OF or OFFSET$OF is used as a pseudo-function, its argument must be a reference to a pointer variable.

```
byte = HIGH (word)
word = HIGH (dword)
byte = LOW (word)
word = LOW (dword)
selector = SELECTOR$OF (pointer)
word = OFFSET$OF (pointer)
```

HIGH used as a function returns the high half of its argument. LOW used as a function returns the low half of its argument. SELECTOR$OF used as a function returns the

base part of its argument. OFFSET$OF used as a function returns the offset part of its argument.

```
HIGH (word ref) = byte
HIGH (dword ref) = word
LOW (word ref) = byte
LOW (dword ref) = word
SELECTOR$OF (pointer ref) = selector
OFFSET$OF (pointer ref) = word
```

HIGH used as a pseudo-function assigns the value of the right side expression to the high half of the referenced word or dword and leaves the low half unchanged.

LOW used as a pseudo-function assigns the value of the right side expression to the low half of the referenced word or dword and leaves the high half unchanged.

SELECTOR$OF used as a pseudo-function assigns the value of the right side expression to the base part of the referenced pointer and leaves the offset part unchanged.

OFFSET$OF used as a pseudo-function assigns the value of the right side expression to the offset part of the referenced pointer and leaves the segment part unchanged.

8.5 CONSTRUCTING A POINTER

The BUILD$PTR builtin function constructs a pointer from a words and a selector.

```
pointer = BUILD$PTR (selector,word)
```

8.6 THE STACK POINTER AND STACK BASE

STACKPTR and STACKBASE reference the hardware stack pointer and stack base registers. STACKPTR and STACKBASE can be used as normal functions or as pseudo-functions.

```
word = STACKPTR
word = STACKBASE
STACKPTR = word
STACKBASE = word
```

STACKPTR used as a function returns the current value of the stack pointer register. STACKBASE used as a function returns the current value of the stack base register.

STACKPTR used as a pseudo function assigns the word value of the right side expression to the stack pointer register. STACKBASE used as a pseudo function assigns the word value of the right side expression to the stack base register.

8.7 DECIMAL ADJUSTMENT

The DEC function performs a decimal adjust on its argument.

```
byte = DEC (byte)
```

8.8 ABSOLUTE VALUE

The IABS and ABS functions returns the absolute value of their arguments.

```
integer = IABS (integer)
real = ABS (real)
```

8.9 SQUARE ROOT AND PI

The SQRT function returns the square root of its argument. The PI function returns the value of pi accurate to 19 decimal digits.

```
real = SQRT (real)
real = PI
```

8.10 TIME DELAYS

The TIME procedure causes a time delay proportional to the value of its argument.

```
call TIME (word)
```

8.11 STRING OPERATIONS

String operations operate on bytes or words as indicated by their names. They all have a length argument which gives the maximum number of items to process.

The length argument can be a byte, word, or dword. The addresses of the bytes or words to process are given by the source and destination arguments. The source and destination can be pointers or types that can be converted to pointers.

8.11.1 String Move

The string move procedures move bytes or words from their source to their destination. The reverse forms of the string move procedures start at the last item in their source and destination instead of the first.

```
call MOVB (source,destination,length)
call MOVW (source,destination,length)
call MOVRB (source,destination,length)
call MOVRW (source,destination,length)
```

The MOVE procedure moves bytes. It operates just like MOVB but its arguments are in a different order.

```
call MOVE (length,source,destination)
```

8.11.2 String Set

The string set procedures move the value of their first argument into every item in their destination string.

```
call SETB (byte,destination,length)
call SETW (word,destination,length)
```

8.11.3 String Translation

The XLAT procedure translates the bytes in the source string and places them in the destination string. The table argument gives the address of a byte array of up to 256 bytes. The translation is performed using each byte in the source string as an index to a byte in the table.

```
call XLAT (source,destination,length,table)
```

8.11.4 String Find and Skip

The second argument of the find and skip functions is a byte or word which is compared to the items of the source string.

The find functions compare each item until an equivalent one is found then they return the index of that item.

The skip functions compare each item until a different one is found then they return the index of that item.

The reverse find and skip functions search starting with the last item in the string.

If the entire string is checked without satisfying the condition, an index of 0FFFFh is returned.

```
word = FINDB (source,byte,length)
word = FINDW (source,word,length)
word = FINDRB (source,byte,length)
word = FINDRW (source,word,length)
word = SKIPB (source,byte,length)
word = SKIPW (source,word,length)
word = SKIPRB (source,byte,length)
word = SKIPRW (source,word,length)
```

8.11.5 String Compare

The left and right arguments are pointers or words.

Items from the left string are compared to items from the right string until they are not equal; then the index of the unequal items is returned. If the left and right strings are the same, then an index of 0FFFFh is returned.

```
word = CMPB (left,right,length)
word = CMPW (left,right,length)
```

8.12 FLAG VALUES

The flag functions return the values of the machine flags.

```
byte = CARRY
byte = ZERO
byte = SIGN
byte = PARITY
```

The FLAGS function references the hardware flags register. FLAGS can be used as a normal function or as a pseudo-function.

```
word = FLAGS
FLAGS = word
```

FLAGS used as a function returns the current value of the flags register. FLAGS used as a pseudo-function assigns the word value of the right side expression to the flags register.

8.13 INPUT AND OUTPUT

The argument to INPUT and OUTPUT is a byte or word specifying one of the hardware ports. If the argument is a byte constant, more compact code is generated. INPUT is a byte function which reads a byte from one of the hardware ports. INWORD is a word function which reads a word from a pair of the hardware ports. OUTPUT and OUTWORD are pseudo-functions that may only appear on the left of an assignment. OUTPUT writes the byte value of the right side expression to one of the hardware ports. OUTWORD writes the word value of the right side expression to a pair of the hardware ports.

```
byte = INPUT (word)
word = INWORD (word)
OUTPUT (word) = byte
OUTWORD (word) = word
```

8.14 MULTIPROCESSOR SYNCHRONIZATION

The LOCKSET builtin function provides a software lock for multiprocessor synchronization. The first argument to LOCKSET is a word or pointer address of a byte variable. The second argument to LOCKSET is a byte. LOCKSET stores the its second argument in the byte variable and returns the old value of the variable. The exchange of the new value for the old value is performed as one uninterruptable operation using the XCHG instruction.

```
byte = LOCKSET (pointer, byte)
```

8.15 ADDRESSING INTERRUPT PROCEDURES

The SET$INTERRUPT procedure and the INTERRUPT$PTR function are used to address the interrupt entry point of procedures declared with the interrupt attribute. The first argument of SET$INTERRUPT is an interrupt number. It must be a constant between 0 and 255. The argument of INTERRUPT$PTR and the second argument of SET$INTERRUPT must be the name of an interrupt procedure.

```
call SET$INTERRUPT (con, name)
pointer = INTERRUPT$PTR (name)
```

SET$INTERRUPT assigns a pointer to the interrupt entry point of the named interrupt procedure to an interrupt vector. INTERRUPT$PTR returns a pointer to the interrupt entry point of the named interrupt procedure.

8.16 SETTING THE 8087 MODE

The SET$REAL$MODE procedure sets the mode word of the 8087.

```
call SET$REAL$MODE (word)
```

8.17 THE MEMORY ARRAY

MEMORY is an external byte array of unknown size. It can be used like any other array variable except that it cannot be the argument to SIZE, LENGTH, and LAST.

A. 86/PL and PL/M-86 Differences

86/PL is a superset of PL/M-86 and most differences are extensions to the PL/M-86 language. However, there are some restrictions in the areas where the compiled code interacts with the 8086 interrupts.

A.1 EXTENSIONS TO PL/M-86

86/PL extends PL/M-86 by relaxing restrictions and by adding new features.

A.1.1 Reserved Words

86/PL has no reserved words, not even EOF. A statement which begins with one of the following words:

DO	IF	PROCEDURE	ENABLE
END	ELSEIF	DECLARE	DISABLE
GO	ELSE	CALL	HALT
GOTO	ENDIF	RETURN	CAUSEINTERRUPT
UNDO			

is assumed to be the statement which begins with that word. All other statements are assignment statements.

A.1.2 Declare Statement

Attributes, the array specifier, the based specifier, and the type specifier can be in any order. All attributes except LABEL, LITERALLY, AT, INITIAL, and DATA may appear within a factored list.

The number in the array specifier can be replaced by a '*' if the array is based or external. The star signifies an unknown dimension for the array.

The variable in the based specifier can be replaced by a '*'. The star signifies a based declaration with no implicit base defined.

One consequence of the new rules for the order of attributes is that the declaration of a based array can be written as:

```
DECLARE x(*) BASED y BYTE;
```

instead of

```
DECLARE x BASED y (10) BYTE;
```

which looked like x was based on the 10'th element of y.

A member of a structure can have the STRUCTURE data type. The name of a structure member can be '*'. Such a member occupies space but cannot be referenced.

Externals have their own scope between the scope of builtin procedures and the module scope. This means that a variable can be declared first EXTERNAL and then redeclared PUBLIC without generating an error. For example, a module which contains:

```
DECLARE xxx BYTE EXTERNAL ; /* from an include file */
```

followed later by:

```
DECLARE xxx BYTE PUBLIC ;
```

will not generate an error in 86/PL but will in PL/M-86. It is therefore possible in 86/PL but not in PL/M-86 to include, in every module of a program, a single file which contains all the external declarations for the program.

A.1.3 The Interrupt Attribute

The interrupt number in the interrupt attribute is optional. If an interrupt number is not given, an entry is not made in the interrupt vector.

A.1.4 Restricted Expressions

A restricted expression (Rule 25) is an expression formed from constant expressions and constant location references.

A constant expression may contain any operators except PLUS and MINUS. The operands in a constant expression must be constants, constant expressions or builtin functions which are constant. The SIZE, LENGTH and LAST builtin functions are always constant. The HIGH, LOW, DOUBLE, INT, and UNSIGN builtin functions with constant arguments are also constants.

The INTERRUPE$PTR, SELECTOR$OF and OFFSET$OF builtin functins are constant location references when their argument is a constant or a relocatable address.

A location reference may be formed with the '.' operator or the '@' operator. The location reference may be to a variable or a long constant.

This less restrictive definition of a restricted expression allows some very useful declarations such as:

```
DECLARE xxx(*) BYTE DATA( LENGTH(xxx)-1, 'string')
```

which is a character string preceded by its length, or the declaration:

```
DECLARE xxx(2) POINTER DATA( @('s1',0), @('s2',0) )
```

which is a pointer array initialized to the addresses of character strings.

A.1.5 Explicitly Based Variables

An explicit base may be specified in a variable reference. An explicitly based variable can have the form:

```
reference -> based-reference
```

or for more flexibility:

```
(expression) -> based reference
```

Since in the first example the reference part can itself be explicitly based, a reference of the form:

```
reference -> based-reference -> based-reference
```

is legal and has the effect of following a chain through memory.

A.1.6 Builtin Functions as Assignment Targets

The builtin functions HIGH, LOW, SELECTOR$OF and OFFSET$OF may be used as assignment targets. A statement of the form:

```
HIGH(word-reference) = expression ;
```

assigns the byte value of expression to the high byte of the word reference. Likewise, a statement of the form:

```
LOW(reference) = expression ;
```

assigns the byte value of expression to the low or only byte of reference.

A.1.7 The IF Block

A series of statements of the form:

```
IF expression ;
    one or more statements
ELSEIF expression ;
    one or more statements
ELSE ;
    one or more statements
ENDIF ;
```

is equivalent to:

```
IF expression THEN
    DO ;
        one or more statements
    END ;
ELSE
    IF expression THEN
        DO ;
            one or more statements
        END ;
    ELSE
        DO ;
            one or more statements
        END ;
```

Note that in the IF block, ELSEIF, ELSE, and ENDIF are separate statements and not part of the syntax of the IF statement.

A.1.8 The UNDO statement

The UNDO statement jumps to the statement which immediately follows a DO block. If UNDO is used with no argument, the immediately containing DO block is the one jumped out of. If there is an argument as in:

```
UNDO xxxxx ;
```

then the DO block is the containing one with the name given by the argument.

A.2 UNSUPPORTED PL/M-86 FEATURES

The calculation of the stack size is not supported.

The NOINTVECTOR control is not supported but no interrupt vector entry is made for interrupt procedures that do not have an interrupt number.

The 8087 emulator is not supported. The 8087 support functions

```
INIT$REAL$MATH$UNIT
GET$REAL$ERROR
SAVE$REAL$STATUS
RESTORE$REAL$STATUS
```

are not supported.

B. Error Messages

The 86/PC compiler will detect a number of error situations and issue appropriate error messages. The various types of error messages and their associated return codes or completion status codes are described in this Appendix.

B.1 WARNINGS

Warnings are generated by 86/PC when a potential error has been encountered, even though an unambiguous and probably correct choice of actions is made. Under UNIX or PC-DOS, a code of one is returned. Under VMS, a *warning* completion status is returned.

B.2 ERRORS

Errors are generated by 86/PC when a statement contains one or more errors that are serious enough that the compiler cannot continue processing the statement. Under UNIX or PC-DOS, a code of two is returned. Under VMS, an error completion status is returned.

B.3 SEVERE ERRORS

These errors are the most severe errors that the user should encounter in normal operation. Severe errors are errors that the compiler cannot recover from, and they cause immediate termination of the current compilation. These errors fall into two general classes. They may be caused by some dynamic or static space overflow within the compiler, and the solution is to reduce the size and/or complexity of the program. Or, they may indicate some problem with the environment within which 86/PC runs. I/O errors generally fall into this category. Under UNIX or PC-DOS, a code of three is returned. Under VMS, a *fatal* completion status is returned.

B.4 FATAL ERRORS

Fatal errors indicate an internal 86/PC failure. They should never be encountered by the user. Under UNIX or PC-DOS, a code of four is returned. Under VMS, a *fatal* completion status is returned.

B.5 LIST OF ERROR MESSAGES

Error messages which may be issued by the compiler are shown below. The initial letter indicates the severity of the error.

E ADDRESS WRAPAROUND

F ARG COUNT REQUEST

E ARGUMENTS NEEDED

E ARRAY ATTRIBUTE IS INCOMPATIBLE

E AT ATTRIBUTE IS INCOMPATIBLE

E AT LEAST ONE CASE REQUIRED

E ATTRIBUTE CAN NOT BE USED WITHIN A PROCEDURE

E ATTRIBUTES INCOMPATIBLE WITH PARAMETER

F BAD BUS FILE (FIX TO ABS)

F BAD BUS FILE (SEG FIX)

F BAD BUS FILE (SELF FIX)

F BAD BUS FILE CONTENT (FIX SEG)

F BAD BUS FILE CONTENT (FIX-UP TO NON-MEM)

F BAD BUS FILE CONTENT (FMT OP)

F BAD BUS FILE CONTENT (SYM OPND)

F BAD MACRO TYPE WHILE WRITING

E BASE VARIABLE IS UNDECLARED

E BASED ATTRIBUTE IS INCOMPATIBLE

E BUILTIN IS NOT ADDRESSABLE

E BYTE OR WORD REQUIRED

E BYTE WORD OR INTEGER REQUIRED

S CALL STACK OVERFLOW; EXPRESSION TOO COMPLEX

E CAN NOT BE NESTED WITHIN EXTERNAL

E CAN NOT BE NESTED WITHIN EXTERNAL PROCEDURE

E CAN NOT BE NESTED WITHIN REENTRANT OR INTERRUPT

F CAN'T CREATE FORK FOR: name

F CAN'T DECLARE EXIT HANDLER

F CAN'T FIND COMPILER PHASE: name

E COMPILER CONTROL SYNTAX

E CONSTANT EXPRESSION REQUIRED

E CONSTANT OVERFLOW

E CONSTANT REQUIRED

E CONTROL IS OUT OF PLACE

S CONTROL STACK OVERFLOW

F CONTROL STACK UNDERFLOW

F COULD NOT MOVE ARGUMENT

F COULDN'T CREATE DICT FILE

F COULDN'T OPEN DICT FILE

S CREATING FILE MIGHT ERASE SOURCE FILE: name

F CURRNT HAS UNKNOWN LOCATION

F DANGLING NODE

E DATA ATTRIBUTE IS INCOMPATIBLE

S DICT OVERFLOW

E DIGIT NOT APPROPRIATE TO NUMBER BASE

E DIMENSION OF ZERO IS NOT ALLOWED

E DO EXPECTED

E DUPLICATE DECLARATION

E DUPLICATE EXTERNAL DECLARATION

E DUPLICATE LABEL DEFINITION

E DUPLICATE MEMBER DECLARATION

E DUPLICATE PARAMETER NAME

E DUPLICATE PROCEDURE NAME

S DYNAMIC MEMORY OVERFLOW

E ELEMENT REFERENCE REQUIRED

E ELSEIF FOLLOWING ELSE

E END DOES NOT MATCH ACTIVE BLOCK

E END OF ELEMENT EXPECTED

E END OF FILE EXPECTED

E END OF LINE EXPECTED

E END OF STATEMENT EXPECTED

E ENDIF EXPECTED

S EOF BEFORE END OF MODULE

E EOF IN QUOTED STRING

E EQUAL EXPECTED

E EXPLICIT ARRAY DIMENSION REQUIRED

E EXPRESSION SYNTAX

E EXTERNAL ATTRIBUTE IS INCOMPATIBLE

E EXTERNAL CAN NOT BE INITIALIZED

E FAR ATTRIBUTE IS INCOMPATIBLE

E FIXED-POINT DIVIDE OVERFLOW

E GOTO TARGET NOT DEFINED

E GOTO TARGET NOT REACHABLE

E IDENTIFIER EXPECTED

E IDENTIFIER TOO LONG, TRUNCATED

E ILLEGAL CHARACTER

E ILLEGAL CHARACTER IN QUOTED STRING

F ILLEGAL OPERATOR IN PERFORM CONSTANT OPERATION

F IMPOSSIBLE STATE TABLE ACTION!!

E INCOMPATIBLE OPERAND MODES

S INIT STACK OVERFLOW

E INITIAL ATTRIBUTE IS INCOMPATIBLE

E INITIAL VALUE DOES NOT MATCH DATA TYPE

E INTEGER REQUIRED

E INTERRUPT ATTRIBUTE IS INCOMPATIBLE

E INTERRUPT PROCEDURE CAN NOT BE TYPED

E INTERRUPT PROCEDURE CAN NOT HAVE PARAMETERS

E INTERRUPT PROCEDURE REQUIRED

E INVALID ASSIGNMENT TARGET

E INVALID BASE SPECIFIER FOR CONSTANT

E INVALID COMPILER CONTROL LINE

E INVALID COMPILER CONTROL LINE (FILENAME EXPECTED)

E INVALID CONSTANT - STRING TOO LONG

E INVALID DIGIT IN NUMBER

E INVALID EMBEDDED ASSIGN

E INVALID INDEX MODE

E INVALID INDEX VARIABLE

E INVALID INDIRECT CALL

E INVALID INTEGER OPERAND

E INVALID NUMERIC CONSTANT

E INVALID RETURN IN MAIN PROGRAM

E INVALID USE OF A LABEL

E INVALID USE OF OUTPUT

E INVALID USE OF PROCEDURE OR LABEL

E LABEL TYPE IS INCOMPATIBLE

E LEFT PARENTHESIS EXPECTED

S LEXIC STACK OVERFLOW

S LEXIC STACK OVERFLOW (ADD CASE)

S LEXIC STACK OVERFLOW (EMBEDDED ASSIGN)

S LEXIC STACK OVERFLOW (PUSH)

S LITERAL STACK UNDERFLOW

E LITERALLY TYPE IS INCOMPATIBLE

E LONG INITIAL VALUE NOT SUPPORTED

E LONG REQUIRED

F LOST SYNCHRONIZATION

F LOST SYNCHRONIZATION 1

F LOST SYNCHRONIZATION 2

F LOST SYNCHRONIZATION 3

E MAXIMUM LITERALLY NESTING EXCEEDED

E MISPLACED STATEMENT

E MISSING RIGHT PAREN

S MISSING STANDARD ERROR FILE NAME

E MODULE NAME IS NEEDED

S MORE THAN 255 VALUE NUMBERS

E MORE THAN ONE SUBSCRIPT

E MULTIPLE ARRAY ATTRIBUTES

E MULTIPLE AT ATTRIBUTES

E MULTIPLE BASED ATTRIBUTES

E MULTIPLE DATA OR INITIAL ATTRIBUTES

E MULTIPLE MODULE NAMES ARE NOT ALLOWED

E MULTIPLE PROCEDURE NAMES ARE NOT ALLOWED

E MULTIPLE PROCEDURE TYPE DEFINITIONS

E MULTIPLE TYPE DEFINITIONS

E NAME IS NOT A LABEL

E NAME IS NOT A REFERENCE

E NAME IS NOT A STRUCTURE

E NAME IS NOT A VALUE

E NAME IS NOT AN ARRAY

E NAME IS NOT AN IDENTIFIER

E NAME IS NOT BASED

E NAME IS NOT DEFINED

E NAME IS NOT MEMBER

E NO BASE VARIABLE DEFINED

E NO FILE NAME GIVEN TO INCLUDE

E NO MATCHING BLOCK

S NO SOURCE FILE GIVEN

F NODE SIZE TOO LARGE

S NODE STACK OVERFLOW

E NOT WITHIN A BLOCK

E NUMBER EXPECTED

E OPERAND MODES INCOMPATIBLE WITH OPERATOR

F ORIGIN SYNCHRONIZATION

S OUTPUT BUFFER OVERFLOW

E POINTER CONSTANT TOO LARGE

E POINTER OR SELECTOR REQUIRED

E POINTER REQUIRED

F POP STMT STRUCTURE: STACK UNDERFLOW

F POP STMT STRUCTURE: UNKNOWN BLOCK TYPE

E PREMATURE END OF FILE

F PREMATURE END-OF-FILE

E PROCEDURE NAME IS NEEDED

S PROCEDURE NESTING LIMIT EXCEEDED

E PUBLIC ATTRIBUTE IS INCOMPATIBLE

E PUBLIC IS INCOMPATIBLE WITH EXTERNAL AT

F PUSHING ILLEGAL STATEMENT STRUCTURE

E REAL REQUIRED

S REAL STACK OVERFLOW

E RECURSIVE LITERALLY

E REENTRANT ATTRIBUTE IS INCOMPATIBLE

E REFERENCE REQUIRED

E REQUIRED TOKEN MISSING: identifier

E RESTRICTED ADDRESS CAN NOT BE BASED

E RESTRICTED ADDRESS REQUIRED

E RESTRICTED CONSTANT EXPRESSION REQUIRED

E RESTRICTED EXPRESSION REQUIRED

F REUSABLE TYPDEF FAILED

E RIGHT PARENTHESIS EXPECTED

F SEGMENT WITH NO NAME

E SEGMENT WRAPAROUND

E SELECTOR ADDRESS IS NOT SUPPORTED

E SELECTOR REQUIRED

E SIMPLE VARIABLE REQUIRED

E SOURCE LINE IS TOO LONG TO PROCESS

F STATE STACK UNDERFLOW

E STATEMENT CAN NOT BE LABELED

E STRING EXPECTED

E STRING TOO LONG

E STRING TOO LONG FOR CONSTANT

S STRUCTURE NESTING LIMIT EXCEEDED

E STRUCTURE TYPE IS INCOMPATIBLE

S TEMPORARY STACK OVERFLOW

E THEN OR SEMICOLON EXPECTED

E TO REQUIRED

E TOO FEW ARGUMENTS

E TOO MANY ARGUMENTS

E TOO MANY INCLUDE DIRECTORIES

S TOO MANY LEXIC BLOCKS

E TOO MANY REAL ARGUMENTS

S TOO MANY TYPE DEFINITIONS

S TREE BUFFER OVERFLOW

S TYPDEF RECORD TOO LONG

E TYPE DEFINITION IS REQUIRED

E TYPED PROCEDURE REQUIRED

S UNABLE TO CREATE STANDARD ERROR FILE: name

S UNABLE TO REDIRECT STANDARD ERROR FILE: name

E UNCLOSED CONDITIONAL ASSEMBLY CONSTRUCTS

E UNDECLARED PARAMETER

E UNKNOWN COMPILER CONTROL

E UNKNOWN COMPILER CONTROL TYPE

F UNKNOWN INCOMING MACRO TYPE

F UNKNOWN LEAF

S UNKNOWN OPTION: option

F UNKNOWN RELOCATABILITY

F UNRECOGNIZED JUMP TYPE

E UNSIGNED CONSTANT EXPRESSION REQUIRED

E UNTYPED PROCEDURE REQUIRED

E VALUE RETURNED FROM SUBROUTINE

E WORD ADDRESS REQUIRED

E WORD OR POINTER REQUIRED

E WRONG NUMBER OF ARGUMENTS

S WSTACK OVERFLOW

F WSTACK UNDERFLOW

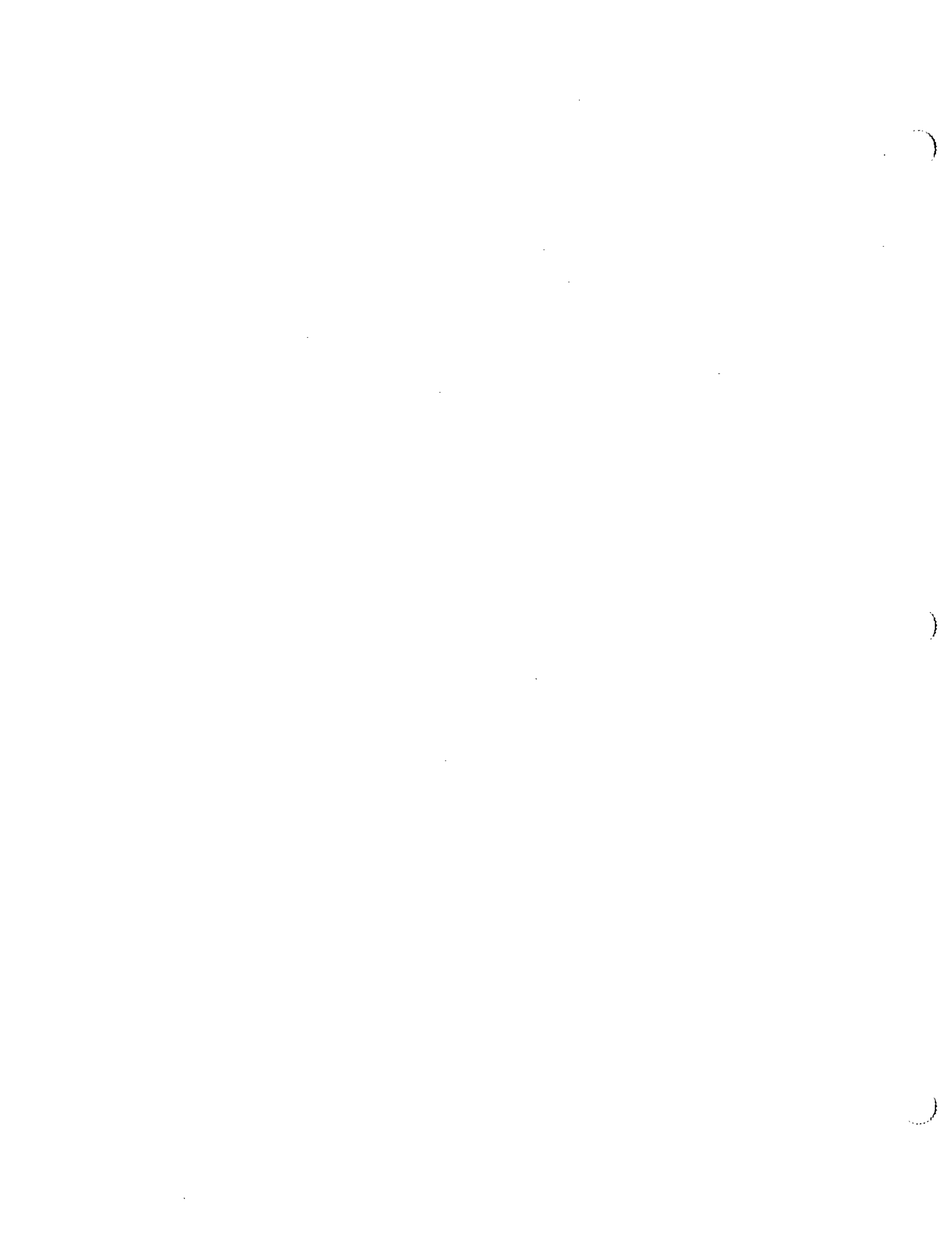
C. Formal Definition of Meta-Language

Chapter 3 provided an informal definition of the meta-language used to describe the syntax of 86/PL. This appendix provides the formal definition of the meta-language.

1. grammar = rule*.
2. rule = variable ‘=’ definition ‘.’.
3. definition = alternate [‘|’ alternate]*.
4. alternate = sequence*.
5. sequence = {unit | grouping | option} [‘*’].
6. grouping = ‘{’ definition [‘*’] ‘}’.
7. option = ‘[’ definition [‘*’] ‘]’.
8. unit = variable | literal.
9. variable = lc-letter [lc-letter | digit | ‘-’]*.
10. literal = ‘'’ {‘''’ | character] * ‘'’ | uc-letter*.
11. lc-letter = any-lower-case-letter.
12. uc-letter = any-upper-case-letter.
13. digit = any-decimal-digit.
14. character = any-character-except-quote.

The variables *lc-letter*, *uc-letter*, *digit*, and *character* have been loosely defined for the sake of simplicity. Formally, they can be defined by enumerating the characters which actually form their definition.

Within a *literal*, an upper-case letter and the corresponding lower-case letter are equivalent.



D. Multiply and Divide for Double Words

The only external calls generated by 86/PC are for multiply, divide, and MOD for the DWORD data type. The compiler uses LQ_DWORD_MUL for multiply and LQ_DWORD_DIV for both divide and MOD.

D.1 ROUTINES FOR INTEL 8086 ASSEMBLER

```
        name    xx_dword_mul_div
        assume  cs:xx_mul_div_code
xx_mul_div_code segment public 'CODE'
        public  lq_dword_mul
        public  lq_dword_div
;
;               M U L T I P L I C A T I O N
;
;       (dx:ax) <- (dx:ax) * (di:cx)
;
lq_dword_mul    proc    far
        xchg    ax,si
        xchg    ax,dx
        mul     cx
        xchg    ax,bx
        mov     ax,si
        mul     di
        add     bx,ax
        mov     ax,si
        mul     cx
        add     dx,bx
        ret
lq_dword_mul    endp


;
;               D I V I S I O N
;
;       (dx:ax) <- (dx:ax)  /  (di:cx)
;       (si:di) <- (dx:ax) mod (di:cx)
;
;       ******  No check is made for division by zero.  ******
;
;       Let N, D, Q, R be in the range [0FFFFFFFFh, 10000h], and let
;       n, d, q, r be in the range [0FFFFh, 0].
;
```

```
lq_dword_div    proc    far
        sub     si,si
        or      dx,dx
        jne     dword_num
        or      di,di
        jne     word_dword
;
;       Case I:  n/d = q rem r
;
word_word:
        div     cx
        xchg    di,dx
        ret
;
;       Case II:  n/D = 0 rem (r=n)
;
word_dword:
        mov     di,ax
        mov     ax,dx
        ret
;
;
dword_num:
        or      di,di
        jne     dword_dword
;
;       Case III:  N/d = Q rem r
;
dword_word:
        xchg    ax,dx
        mov     bx,dx
        mov     dx,di
        div     cx
        xchg    ax,bx
        div     cx
        mov     di,dx
        mov     dx,bx
        ret
;
;       Case IV:  N/D = q rem R
;
;      The largest quotient is OFFFFFFFFh / 000010000H = OFFFFh
; and the largest remainder is OFFFFFFFEh mod OFFFFFFFFh = OFFFFFFFEh
;
dword_dword:
        xchg    dx,di
        mov     bx,cx
        mov     cx,16
xloop:
        shl     ax,1
        rcl     di,1
        rcl     si,1
        sub     di,bx
        sbb     si,dx
        jnl     xhi
xlow:
        add     di,bx
        adc     si,dx
        jmp     short xact
xhi:
```

```
        inc     ax
xact:
        loop    xloop
;
        mov     dx,cx
        ret
lq_dword_div    endp

xx_mul_div_code ends

        end
```

D.2 ROUTINES FOR TEKTRONIX 8086 ASSEMBLER

```
        name    dword_mul_div
        section i.dword_mul_div,class=INSTRQQ

        global  lq_dword_mul
        global  lq_dword_div
;
;               M U L T I P L I C A T I O N
;
;       (dx:ax) <- (dx:ax) * (di:cx)
;
lq_dword_mul
        xchg    ax,si
        xchg    ax,dx
        mul     cx
        xchg    ax,bx
        mov     ax,si
        mul     di
        add     bx,ax
        mov     ax,si
        mul     cx
        add     dx,bx
        rets


;
;               D I V I S I O N
;
;       (dx:ax) <- (dx:ax)  /  (di:cx)
;       (si:di) <- (dx:ax) mod (di:cx)
;
;       ****** No check is made for division by zero.  ******
;
;       Let N, D, Q, R be in the range [0FFFFFFFFh, 10000h], and let
;       n, d, q, r be in the range [0FFFFh, 0].
;
lq_dword_div
        sub     si,si
        or      dx,dx
        jne     dword_num
        or      di,di
        jne     word_dword
;
;       Case I:  n/d = q rem r
```

```
;
word_word div   cx
        xchg    di,dx
        rets
;
;       Case II:  n/D = 0 rem (r=n)
;
word_dword mov  di,ax
        mov     ax,dx
        rets
;
;
dword_num or    di,di
        jne     dword_dword
;
;       Case III:  N/d = Q rem r
;
dword_word xchg ax,dx
        mov     bx,dx
        mov     dx,di
        div     cx
        xchg    ax,bx
        div     cx
        mov     di,dx
        mov     dx,bx
        rets
;
;       Case IV:  N/D = q rem R
;
;      The largest quotient is 0FFFFFFFFh / 000010000H = 0FFFFh
; and the largest remainder is 0FFFFFFFEh mod 0FFFFFFFFh = 0FFFFFFFEh
;
dword_dword xchg dx,di
        mov     bx,cx
        mov     cx,#16
xloop   shl     ax,#1
        rcl     di,#1
        rcl     si,#1
        sub     di,bx
        sbb     si,dx
        jnl     xhi
xlow    add     di,bx
        adc     si,dx
        jmpsh   xact
xhi     inc     ax
xact    loop    xloop
;
        mov     dx,cx
        rets

        end
```

E. Installing on VAX/VMS

This chapter discusses the method of installing an 86/PC delivery tape on a VAX under the VMS operating system. The discussion assumes that the delivery is made on a 9-track magnetic tape. If some other medium is used, the same general method should apply. Any special instructions will be found on an installation memorandum packed with the delivery.

This discussion assumes that the installation is being performed by an experienced VMS systems programmer.

E.1 SUPPORTED OPERATING ENVIRONMENT

This version of 86/PC requires a Digital Equipment Corporation VAX or Micro-VAX and the VMS operating system, release 4.3 or later.

E.2 RESTORING THE TAPE

The delivery tape is in standard VMS *backup* format. It may be restored by

```
$ alloc mta tape
$ mount/foreign/den=1600 tape
$ backup/rew/log tape:86pc.bkp [.86pc...]
```

This creates a subdirectory of the current directory called *86ds*.

E.3 DEFINING LOGICAL NAMES

A logical name must be defined to specify the installed location of the 86/PC components. Suppose that restoring the tape (Section E.2) created the *86ds* directory as a subdirectory of *sys$disk:[tools]*. The following definition should be made:

```
$ define sys$86as sys$disk:[tools.86pc]
```

If only a few people will be using 86/PC, this definition may be placed in those person's LOGIN.COM files. If many people will be using 86/PC, the definition should be made system-wide.

E.4 INSTALLING THE NATIVE COMMANDS

86/PC is intended to be installed as a VMS native command and executed using standard DCL syntax. A CLD file is provided which defines the invocation syntax to the DCL processor and a HELP file is provided which can provide on-line help.

The command is installed by the *SET COMMAND* DCL command. To install 86/PC, use

```
$ set command sys$86pc:86pc
```

Help on the use of 86/PC may be obtained by

```
$ help/lib=sys$86pc:86pc 86pc
```

If only a few people will be using 86/PC, these definitions may be placed in those person's LOGIN.COM files. If many people will be using 86/PC, the definitions should be made system-wide by modifying the system DCL command tables.

F. Installing on UNIX Systems

This chapter discusses installing the 86/PC compiler on various UNIX systems. Only binary installations are discussed in this chapter. If you are installing a UNIX source version of 86/PC, see Appendix I.

This discussion assumes that the installation is being performed by an experienced UNIX user or systems programmer.

F.1 SUPPORTED OPERATING ENVIRONMENT

Generally, the UNIX version of 86/PC requires a 16-bit or 32-bit, byte-addressing machine which runs a true Version 7, System III, System V, or Berkeley 4.2/4.3 bsd UNIX or XENIX operating system. Only the most common of these are supported in binary form.

F.2 BINARY INSTALLATION

The delivery medium for a binary installation will normally consist of a 9-track magnetic tape. In some cases, however, it may be one or more diskettes or other special types of recording media.

In general, installation is performed by logging in as *root*, mounting the delivery tape, and entering

```
cd /
tar xv
```

If other methods are required, a memorandum describing them will accompany the delivery.

The result of the installation will be to place the driver for 86/PC into the */usr/bin* directory. The individual phases of 86/PC will be placed in */usr/lib/86pc*.

F.3 SELECTING 86/PC FINAL OUTPUT DEFAULT

As delivered, the 86/PC compiler will produce object modules in Intel standard object module format by default. In some versions, this may be changed to produce Tektronix LAS instead by

```
cp /usr/lib/86pc/86ptfo /usr/lib/86pc/86pfo
```

To restore Intel object as the default, do

```
cp /usr/lib/86pc/86pifo /usr/lib/86pc/86pfo
```

F.4 TAILORING WITH ENVIRONMENT VARIABLES

The 86/PC compiler can be extensively tailored by using environment variables. So as not to clutter the environment, this method is best when only a few simple changes are desired. The methods described in Section F.5 are preferred when extensive changes are to be made.

F.4.1 Global Tailoring Changes

Many parts of 86/PC use temporary files (by default located in */usr/tmp*) and many need to find the location of the directory (by default */usr/lib/86pc*) where various phases reside. The following environment variables may be used to change these locations:

A86PCTMP — If this variable is defined, its contents will be used as a prefix for all temporary file names. For example, if it contains "/tmp/", all temporary files will be created in the */tmp* directory.

A86PCLIB — If this variable is defined, its contents will be used as a prefix for all phase names. For example, if it contains "/u3/tools/dev/", 86/PC will look in the */u3/tools/dev* directory for each of the compiler phases.

F.4.2 Local Tailoring Changes

Environment variables may be used to provide arguments to 86/PC. The contents of the variable "*A86PCHEAD*" will be processed as arguments to 86/PC before the arguments on the invocation line are processed. After processing the invocation line arguments, the contents of the variable "*A86PCTAIL*" are processed as arguments.

For example, the definition

```
A86PCHEAD="-p58 -Xs.plm -Xo.obj -Xl.lis"
export A86PCHEAD
```

will cause all 86/PC compilations to use a listing page depth of 58 lines, a source suffix of ".plm", an object suffix of ".obj", and a listing suffix of ".lis".

F.4.3 Specifying Maximum Number of Arguments

86/PC component has been configured to process a reasonable number of arguments. Sometimes, however, this predetermined maximum may not be enough. If the variable

"*A86PCMAXA*" is defined, its contents are taken as a decimal integer giving the maximum number of arguments to allow for 86/PC.

Note that these variables cannot be used to increase the maximum beyond any limits which may be imposed by the operating system.

F.5 TAILORING WITH AN INITIALIZATION FILE

Very detailed tailoring for 86/PC may be performed by use of a file named *86pc.ini*. When any 86/PC is executed, this file is searched for in the following places (in order): the current directory and in */usr/lib/86pc*. If there is an *A86PCINI* environment variable, its contents are used as the complete path to the file instead.

If it is found, it is opened and read, searching for lines which begin with the name of the command (i.e. "86pc") followed by a single colon. The remainder of each such line is processed just as if it were a set of invocation options appearing before the options on the invocation line. When the last such line is processed, the actual invocation options are processed. Finally, the file is searched for lines beginning with the command name followed by *two* colons and these lines are processed as invocation options.

Lines are processed in order of appearance, and any *double-colon* lines must follow all *single-colon* lines.

F.5.1 Examples

The *86pc.ini* line

```
86pc: -p58 -Xs.plm -Xo.obj -Xl.lis
```

will establish a page depth of 58 lines, source suffix of ".plm", object suffix of ".obj", and listing suffix of ".lis" for all 86/PC invocations.

G. Installing on PC-DOS

This chapter discusses the method of installing the 86/PC compiler on an IBM PC computer under the PC-DOS operating system.

This discussion assumes that the installation is being performed by someone who is familiar with installing system software under DOS.

G.1 SUPPORTED OPERATING ENVIRONMENT

This version of 86/PC requires an IBM PC-XT or PC-AT or equivalent computer and the PC-DOS or MS-DOS operating system, Version 3.1 or later.

G.2 RESTORING THE DISKETTE

The 86/PC delivery consists of one or more diskettes in standard DOS copy format. By default, the various components reside in the "\86pc.lib" directory which must be created by

```
mkdir \86pc.lib
```

To restore the diskettes, first set the current directory to be 86pc.lib by

```
cd \86pc.lib
```

and then restore the diskettes by performing

```
copy a:*.* .
```

for each.

G.3 MAKING THE TEMPORARY DIRECTORY

86/PC needs to create temporary files while operating. These are created in the directory "\tmp". If this directory does not exist, it should be created by

```
mkdir \tmp
```

G.4 INSTALLING 86/PC IN THE SEARCH PATH

The 86/PC command should be in a directory which is in your search path. The simplest way to do this is just to place the "86pc.lib" directory in the search path. If your AUTOEXEC.BAT file does not have a *path* statement, add

```
path c:\86pc.lib
```

to the file. If there already is a *path* statement, add ";c:\86pc.lib" to the end of the statement. It might, for example, then look like

```
path c:\bin;c:\86pc.lib
```

A slightly more complex, but more efficient, method is to place only the command name in a directory that is in the search path. Many systems have, for example, a "\bin" directory to contain commands and this is a good place to move the 86/PC command. It is only necessary to move the *86pc.exe* file to the "\bin" directory, leaving the subphases in the "\86pc.lib" directory.

G.5 SELECTING 86/PC FINAL OUTPUT DEFAULT

As delivered, the 86/PC compiler will produce object modules in Intel standard object module format by default. With some versions, Tektronix LAS may be produced instead, by

```
copy \86pc.lib\86ptfo.exe \86pc.lib\86pfo.exe
```

To restore Intel object as the default, do

```
copy \86pc.lib\86pifo.exe \86pc.lib\86pfo.exe
```

G.6 TAILORING WITH ENVIRONMENT VARIABLES

The 86/PC compiler can be extensively tailored by using environment variables. So as not to clutter the environment, this method is best when only a few simple changes are desired. The methods described in Section G.7 are preferred when extensive changes are to be made.

G.6.1 Global Tailoring Changes

86/PC uses temporary files (by default located in *tmp*) and needs to find the location of the directory (by default *86pc.lib*) where various phases reside. The following environment variables may be used to change these locations:

A86PCTMP If this variable is defined, its contents will be used as a prefix for all temporary file names. For example, if it contains "d:\tmp\", all temporary files will be created in the \tmp directory on device "d:".

A86PCLIB If this variable is defined, its contents will be used as a prefix for all phase names. For example, if it contains "d:\tools\", 86/PC will look in the *\tools* directory on device "d:" for each of the compiler phases.

G.6.2 Local Tailoring Changes

Environment variables may be used to provide arguments to 86/PC. The contents of the variable "*A86PCHEAD*" will be processed as arguments to the 86/PC before the arguments on the invocation line are processed. After processing the invocation line arguments, the contents of the variable "*A86PCTAIL*" are processed as arguments.

For example, the definition

```
set A86PCHEAD=-p58 -Xs.plm -Xo.obj -Xl.lis
```

will cause all 86/PC compilations to use a listing page depth of 58 lines, a source suffix of ".plm", an object suffix of ".obj", and a listing suffix of ".lis".

G.6.3 Specifying Maximum Number of Arguments

86/PC has been configured to process a reasonable number of arguments. Sometimes, however, this predetermined maximum may not be enough. If the variable "*A86PCMAXA*" is defined, its contents are taken as a decimal integer giving the maximum number of arguments to allow for 86/PC.

Note that these variables cannot be used to increase the maximum beyond any limits which may be imposed by the operating system.

G.7 TAILORING WITH AN INITIALIZATION FILE

Very detailed tailoring for 86/PC may be performed by use of a file named *86pc.ini*. When 86/PC is executed, this file is searched for in the following places (in order): the current directory, *\86pc.lib*, *c:\86pc.lib*, **, and *c:*. If there is an *A86PCINI* environment variable, its contents are used as the complete path to the file instead.

If it is found, it is opened and read, searching for lines which begin with the name of the command (i.e., "86pc") followed by a single colon. The remainder of each such line is processed just as if it were a set of invocation options appearing before the options on the invocation line. When the last such line is processed, the actual invocation options are processed. Finally, the file is searched for lines beginning with the command name followed by *two* colons and these lines are processed as invocation options.

Lines are processed in order of appearance, and any *double-colon* lines must follow all *single-colon* lines.

G.7.1 Examples

The *86pc.ini* line

```
86pc: -p58 -Xs.plm -Xo.obj -Xl.lis
```

will establish a page depth of 58 lines, source suffix of ".plm", object suffix of ".obj", and listing suffix of ".lis" for all 86/PC invocations.

H. Installing on Tektronix 856x

This chapter discusses the method of installing an 86/PC delivery diskette on a Tektronix 856x under the TNIX operating system.

This discussion assumes that the installation is being performed by an experienced TNIX user or systems programmer.

H.1 SUPPORTED OPERATING ENVIRONMENT

This version of 86/PC requires a Tektronix 8560, 8561, or 8562 computer and the TNIX operating system, Version 2 or later.

H.2 RESTORING THE DISKETTE

The delivery diskette is in standard TNIX *fbr* format. To install it, log in as *root* and enter

```
install
```

If questions appear on the screen during installation, answer them appropriately.

The result of the installation will be to place the driver for 86/PC into the */usr/bin* directory. The individual phases of 86/PC will be placed in */usr/lib/86pc*.

H.3 SELECTING 86/PC FINAL OUTPUT DEFAULT

As delivered, the 86/PC compiler will produce object modules in Intel standard object module format by default. To change this default so that Tektronix LAS is produced instead, do

```
cp /usr/lib/86pc/86ptfo /usr/lib/86pc/86pfo
```

To restore Intel object as the default, do

```
cp /usr/lib/86pc/86pifo /usr/lib/86pc/86pfo
```

H.4 TAILORING WITH ENVIRONMENT VARIABLES

The 86/PC compiler can be extensively tailored by using environment variables. So as not to clutter the environment, this method is best when only a few simple changes are desired. The methods described in Section H.5 are preferred when extensive changes are to be made.

H.4.1 Global Tailoring Changes

86/PC uses temporary files (by default located in */usr/tmp*) needs to find the location of the directory (by default */usr/lib/86pc*) where various phases reside. The following environment variables may be used to change these locations:

A86PCTMP — If this variable is defined, its contents will be used as a prefix for all temporary file names. For example, if it contains "/tmp/", all temporary files will be created in the /tmp directory.

A86PCLIB — If this variable is defined, its contents will be used as a prefix for all phase names. For example, if it contains "/u3/tools/dev/", 86/PC will look in the */u3/tools/dev* directory for each of the compiler phases.

H.4.2 Local Tailoring Changes

Environment variables may be used to provide arguments to 86/PC. The contents of the variable "*A86PCHEAD*" will be processed as arguments to 86/PC before the arguments on the invocation line are processed. After processing the invocation line arguments, the contents of the variable "*A86PCTAIL*" are processed as arguments.

For example, the definition

```
A86PCHEAD="-p58 -Xs.plm -Xo.obj -Xl.lis"
export A86PCHEAD
```

will cause all 86/PC compilations to use a listing page depth of 58 lines, a source suffix of ".plm", an object suffix of ".obj", and a listing suffix of ".lis".

H.4.3 Specifying Maximum Number of Arguments

86/PC has been configured to process a reasonable number of arguments. Sometimes, however, this predetermined maximum may not be enough. If the variable "*A86PCMAXA*" is defined, its contents are taken as a decimal integer giving the maximum number of arguments to allow for 86/PC.

Note that these variables cannot be used to increase the maximum beyond any limits which may be imposed by the operating system.

H.5 TAILORING WITH AN INITIALIZATION FILE

Very detailed tailoring for 86/PC may be performed by use of a file named *86pc.ini*. When any 86/PC command is executed, this file is searched for in the following places (in order): the current directory and in */usr/lib/86pc*. If there is an *A86PCINI* environment variable, its contents are used as the complete path to the file instead.

If it is found, it is opened and read, searching for lines which begin with the name of the command (i.e., "86pc") followed by a single colon. The remainder of each such line is processed just as if it were a set of invocation options appearing before the options on the invocation line. When the last such line is processed, the actual invocation options are processed. Finally, the file is searched for lines beginning with the command name followed by *two* colons and these lines are processed as invocation options.

Lines are processed in order of appearance, and any *double-colon* lines must follow all *single-colon* lines.

H.5.1 Examples

The *86pc.ini* line

```
86pc: -p58 -Xs.plm -Xo.obj -Xl.lis
```

will establish a page depth of 58 lines, source suffix of ".plm", object suffix of ".obj", and listing suffix of ".lis" for all 86/PC invocations.

I. Source Installation on UNIX Systems

This appendix discusses installation of a source version of 86/PC on a UNIX system. This is a complicated process and should only be performed by an experienced UNIX systems programmer.

This chapter only discusses the actual building and installing of the 86/PC compiler. Appendix F should be read for a discussion of various post-installation tailoring options.

I.1 SUPPORTED OPERATING ENVIRONMENT

Generally, the UNIX version of 80/DS requires a 16-bit or 32-bit, byte-addressing machine which runs a true Version 7, System III, System V, or Berkeley 4.2/4.3 bsd UNIX or XENIX operating system. Because of the lack of compatibility among the various systems, installing of a source version of 86/PC may require changes to the programs in order to make them operate correctly. Such changes are the responsibility of the installer.

I.2 RESTORING THE DELIVERY TAPE

The 86/PC compiler is distributed in a manner intended to be very portable. It does not depend upon the availability of any particular tape archiving program. This chapter describes the format of the delivery tape and discusses the steps required for restoring such a tape.

I.2.1 Tape Format

The 86/PC software is distributed on one reel of 9-track, standard magnetic tape containing several files:

1. A small C source program, named "sarin", which may be used to extract the source files from the archives contained in the other tape files;

2. One or more archive files containing the source.

I.2.2 Restoring the Tape

A distribution in the form of an industry-standard magnetic tape will have the format:

```
source for sarin utility
<EOF>
source archive #1
<EOF>
source archive #2
<EOF>
.
.
.
<EOF>
<EOF>
```

All data blocks on the tape are 512 bytes long. A tape will have one or more source archives.

The files may be restored by a shell sequence such as:

```
(dd of=sarin.c;
dd of=sarc1;
dd of=sarc2;
.
.
.
) </dev/rmt0
```

where */dev/rmt0* is the name of the raw interface for the tape drive on which the delivery tape is mounted.

I.2.3 Compiling the Sarin Utility

The sarin utility may be used to extract the source files from the archive. It should be compiled by

```
cc sarin.c -o sarin
```

which will leave an executable version of the utility in the file "sarin".

Because all tape blocks are 512 bytes long, it is possible that the last block of the sarin.c file will contain one or more trailing ASCII nulls. This should not cause any problems. However, if the C compiler complains about illegal characters which appear to be at the end of the file, the shell sequence

```
ed sarin.c
w
q
```

should remove the trailing nulls.

I.2.4 Extracting Source Files From the Archives

Once the sarin utility has been compiled, it may be used to process the archives.

I.2.4.1 Structure of the Source Archives

The archives consist of a sequence of lines, each terminated by a newline. Lines whose first three characters are "$" are *control lines*. The possible control lines are:

```
{$}C commentary
```

which causes the commentary to be displayed when the archive is processed;

```
{$}D directory-name
```

which causes the named directory to be created by "mkdir";

```
{$}F file-name
```

which causes the named file to be created and opened;

```
{$}E
```

which causes the file named in the last "$F" control line to be closed; and

```
{$}Z
```

which terminates the archive. Any line which is not a control line is source; it is written to the file named in the last "$F" control line.

I.2.4.2 Processing the Source Archives

The commands

```
sarin -t <sarc1
sarin -t <sarc2
.
.
.
```

will display a listing of the names of all archived files and directories. The listing will be written on the standard error file.

Actual extraction of the source files may be performed by the commands

```
sarin <sarc1
sarin <sarc2
.
.
.
```

These will create the necessary directories and subdirectories and restore all source files. A progress log will also be displayed on the standard error file.

During extraction, warning messages will be issued if any of the directories or subdirectories already exist, but extraction will proceed using the existing directories.

I.3 INSTALLING THE 86/PC COMPILER

This chapter discusses the procedures to produce a working 86/PC from the delivery tape.

I.3.1 Restoring the 86/PC Delivery Tape

The 86/PC delivery tape has two archive files. The restoration process is described in Section I.2.

Restoring the archives will create the following directory structure:

```
86pc.d              (root directory for 86pc)
     common.d:      (directory for common C source)
     p1.d:          (directory for p1 source)
     p2.d:          (directory for p2 source)
     pc.d:          (directory for driver source)
     pcg.d:         (directory for pcg source)
     pfo.d:         (directory for pfo source)
     pjo.d:         (directory for pjo source)
     pp.d:          (directory for pp source)
     psym.d         (directory for psym source)
     ptfo.d         (directory for ptfo source)
     pxrf.d         (directory for pxrf source)
```

I.3.2 Modifying the 86/PC Shell Scripts

All examples in this chapter assume that you have already restored the delivery tape and extracted the source files. It is also assumed that your working directory is "86pc.d". The following shell scripts are provided:

pcdefs.sh	which defines things for the other shell scripts
pccompile.sh	which compiles everything
pclink.sh	which links everything
pcinstall.sh	which installs everything
pcprint.sh	which prints everything

Before proceeding with the generation and installation, examine these shell scripts and make any required changes as indicated by their comments and the following descriptions.

I.3.2.1 Modifying pcdefs.sh

This shell script sets up definitions of shell variables used in other shell scripts.

I.3.2.1.1 Installation Directory for the Driver

The shell variable I is set to the directory where the 86/PC driver is installed. As distributed, the shell variable is defined as

```
I=/usr/bin
```

I.3.2.1.2 Installation Directory for the Compiler Phases

The shell variable P is set to the directory where the 86/PC driver expects to find its phases. The library will also be installed there. As distributed, the shell variable is defined as

```
P=/usr/lib/86pc
```

This corresponds to the default definition of PDIR in some of the C source code. If the definition of P is changed, then PDIR must also change. This is done through the C compilation flags.

I.3.2.1.3 C Compilation Flags

The shell variable C is set to the flags to be used for C compilations. As distributed, the shell variable is defined as

```
C='-I../common.d'
```

This sets the search path for include files.

I.3.2.1.3.1 Temporary File Directory

TDIR is defined in some source modules to be "/usr/tmp/". This is the name of the directory in which 86/PC should create temporary files. Some UNIX systems, notably PWB, do not normally have the "/usr/tmp" directory. For such systems, TDIR should be redefined – usually to "/tmp/". This is done by adding

```
-DTDIR="/tmp/"
```

to the definition of C.

I.3.2.1.3.2 Phase Installation Directory

PDIR is defined in some source modules to be "/usr/lib/86pc/". This is the name of the directory in which the 86/PC driver expects to find its phases. If you wish to change this for some reason, PDIR should be redefined. This is done by adding

```
-DPDIR="/new/dir"
```

to the definition of *C*. Note that the shell variable *P* must also be redefined.

I.3.2.1.3.3 Treatment of Warning Messages

When compiling with some C compilers, a number of warning messages will be issued. These messages relate to different interpretations of the proper way to use the C language in certain circumstances; they can be ignored. These warnings can be suppressed by adding *-w* to the definition of *C*.

I.3.2.1.3.4 C Compiler Optimization

Most C compilers can provide processing to attempt to produce more optimal code; this may be specified by adding *-O* to the definition of *C*. It is possible that this will produce a smaller, faster 86/PC. Considering the problems frequently encountered with the use of such optimizers, we do not recommend this unless you have first successfully installed and tested 86/PC without optimization.

I.3.2.1.3.5 Other C Compiler Options

Any other desired C compiler options of local interest (usually none) may be specified by adding to the definition of *C*.

I.3.2.1.4 Specifying Linker Options

The shell variable *L* is set to the flags to be used by the linker. As distributed, the shell variable is defined as

```
L='-i'
```

I.3.2.1.4.1 Split I/D Linking

When the *-i* option is used, the linker produces an executable image (text file) using separate code and data spaces. This is advisable for best performance of 86/PC except on the VAX under Berkeley UNIX. Berkeley UNIX uses a demand paging mechanism and the split i/d concept is not relevant to that environment. When generating 86/PC under Berkeley UNIX, remove the "-i" from the definition of *L*.

I.3.2.1.4.2 Other Linker Options

Any other desired linker options of local interest (usually none) may be specified by adding to the definition of *L*.

I.3.2.2 Modifying pccompile.sh

The local name of the C compiler should replace "cc" if necessary.

I.3.2.3 Modifying pclink.sh

The local name of the C compiler should replace "cc" if necessary.

I.3.2.4 Modifying pcinstall.sh

This shell script, among other things, installs the 86/PC manual pages (provided as nroff source). This should be checked for compatibility with your installation's standards.

I.3.2.5 Modifying pcprint.sh

This shell script uses the "pr" program. If there is a more appropriate local routine, that may be used instead.

I.3.3 Using the 86/PC Shell Scripts

The following sections describe how to generate 86/PC by running the shell scripts that have been examined and, if necessary, modified.

I.3.3.1 Compiling the Source

The command

```
sh -v pccompile.sh
```

compiles all of 86/PC.

I.3.3.2 Linking the Object

Next, the object is linked to make the executable modules by running pclink.sh.

I.3.3.3 Installing 86/PC

86/PC is next installed by running pcinstall.sh. This installs the driver, all the phases and the manual pages according to the directory assignments made in pcdefs.sh. It should be run while logged on as the owner of these directories.

I.3.3.4 Listing 86/PC

A listing of the source of 86/PC may be produced by:

```
sh -v pcprint.sh >pcprint
```

Index

-a invocation option 3
-B invocation option 5
-d invocation option 4, 5
-E invocation option 5
-i invocation option 4, 5
-J invocation option 4
-K invocation option 6
-l invocation option 3, 4
-M invocation option 3
-O invocation option 4
-p invocation option 4, 5
-s invocation option 3, 4
-t invocation option 4, 5
-TT invocation option 5
-V invocation option 6
-x invocation option 3
-Xi invocation option 5
-Xl invocation option 5
-Xp invocation option 4
-Xs invocation option 4, 5

.i file suffix 5
.q file suffix 3
.S file suffix 4

/CROSS_REFERENCE qualifier 7
/DEBUG qualifier 7
/DEFINE qualifier 9
/INCLUDES qualifier 9
/LIST qualifier 7, 8
/MACHINE_CODE qualifier 8
/MODEL qualifier 8
/NOCROSS_REFERENCE qualifier 7
/NODEBUG qualifier 7
/NOLIST qualifier 8
/NOMACHINE_CODE qualifier 8
/NOOBJECT qualifier 8
/NOOPTIMIZE qualifier 8

/OBJECT qualifier 8
/OPTIMIZE qualifier 8
/SYNTAX qualifier 9
/tmp directory 83
/usr/bin directory (TNIX) 75
/usr/bin directory (UNIX) 67
/usr/lib/86pc directory 83
/usr/lib/86pc directory (TNIX) 75
/usr/lib/86pc directory (UNIX) 67
/usr/tmp directory 83

186 model option 8

286 model option 8

8087 emulator not supported 48
8087 mode setting 44
86/PC delivery tape, restoring 82
86/PC installation 85
86/PC invocation 3, 6
86/PC manual pages 85
86/PC overall operation 10
86/PC phases 83
86/PC shell scripts 82
86/PC, listing 85
86p1 compiler phase 10
86p2 compiler phase 10
86pc.d directory 82
86pc.ini initialization file (DOS) 73
86pc.ini initialization file (TNIX) 77
86pc.ini initialization file (UNIX) 69
86pc.lib directory (DOS) 71
86pcg compiler phase 10
86pfo compiler phase 10
86pifo compiler phase 10
86pjo compiler phase 10
86pp compiler phase 10
86psym compiler phase 10
86ptfo compiler phase 10
86pxrf compiler phase 10

A86PCINI environment variable (DOS) 73
A86PCINI environment variable (TNIX) 77
A86PCINI environment variable (UNIX) 69
ABS builtin 41
Absolute base 36
ADDRESS data type 24
Addresses 35
Archive structure 81
Argument files 6
Arguments 30
Arguments, maximum number (DOS) 73

Arguments, maximum number (TNIX) 76
Arguments, maximum number (UNIX) 68
Arrays 24
Assembly listing option (-a) 3
Assembly listing option (-S) 4
Assignment statement 30, 32, 39, 41, 45
Assignments, embedded 35
AT attribute 22
AT_VARIABLES option 8

BASED attribute 24, 36
Based references 36
Based variable 22, 24
Based, explicitly 46
BASED_VARIABLES option 9
Basic type attributes 24
Binary installation (UNIX) 67
Blanks 13
Block IF statement 29
BNF 15
BUILD$PTR builtin 41
Builtin for absolute value 41
Builtin for decimal adjustment 41
Builtin for stack pointer manipulation 41
Builtin for time delay 42
Builtin identifiers and functions 39
Builtin to reference memory 44
Builtins for input and output 44
Builtins for shifts and rotates 40
Builtins for size of variables 39
Builtins for string comparison 43
Builtins for string moving 42
Builtins for string operations 42
Builtins for string scanning 43
Builtins for string setting 42
Builtins for string translation 42
Builtins for subfield referencing 40
Builtins for type conversions 39
Builtins to test flag values 43
BYTE data type 24
Byte variable 40

C compiler 84
C compiler optimization 84
C compiler warnings 84
C compiler, other options 84
CALL statement 18, 30, 36
Capabilities and features 1
CARRY builtin 43
Carry machine flag 40
CASE statement 28

CAUSE$INTERRUPT statement 31
Class of procedures 19
CLD files (VMS) 66
CMPB builtin 43
CMPW builtin 43
CODE model option 8
Code segment 22
Codes, return 6
Comments 13
Compatible types 28, 30, 32
Compilation, conditional 12
Compile-time constants 11
Compile-time control language 10
Compile-time expressions 11
Compile-time variables 5, 9, 11
Compiler controls 10
Compiler debugging options 5
Compiler invocation 3, 6
Compiler version option (-V) 6
Completion status 10
Conditional compilation 12
Conditional expression 27, 29, 32
Constant expression 46
Constant operand 23, 31, 34, 35
Constants 37
Constants, compile-time 11
Constants, signed 23
Contiguous allocation 21
Control line 10
Control, EJECT 12
Control, ELSE 12
Control, ELSEIF 12
Control, ENDIF 12
Control, IF 12
Control, INCLUDE 11
Control, LIST 12
Control, NOLIST 12
Control, RESET 12
Control, SET 11
Control, SUBTITLE 12
Control, TITLE 12
Controls 10
Controls, unimplemented 13
Cross reference listing option (-x) 3

DATA attribute 22, 24
DATA model option 8
Data segment 23
DCL (VMS) 66
DCL command tables (VMS) 66
Dd utility (UNIX) 80

Debug output 7
DEC builtin 41
Declarations 21
Declarations, factored 21
DECLARE statement 21, 45
Default file suffixes 4
Defining logical names (VMS) 65
Definition of a module 17
Delivery tape, restoring (UNIX) 79
Differences between 86/PL and PL/M-86 45
Dimension attribute 24
Directory list option (-I) 5
Directory, /tmp 83
Directory, /usr/lib/86pc 83
Directory, /usr/tmp 83
DISABLE statement 31
Diskette (DOS) 71
Diskette (TNIX) 75
Divide and multiply 61, 63
DO groups 27
DO statement 27
Dollar sign 37
DOS 71
DOUBLE builtin 39, 46
DWORD data type 14, 24, 61

Editing DCL command tables (VMS) 66
EJECT control 12
Element attributes 23
ELSE control 12
ELSE statement 30, 47
ELSEIF control 12
ELSEIF statement 29, 47
Embedded assignments 35
ENABLE statement 31
END statement 27, 32
ENDIF control 12
ENDIF statement 30, 47
Endings 32
Entry point of main program 17
Environment variable tailoring (DOS) 72
Environment variable tailoring (TNIX) 76
Environment variable tailoring (UNIX) 68
Error message list 49
Error messages 49
Errors 49
Executable statements 27
Explicit base 24
Explicit base, absolute 36
Explicitly based references 36
Expression operators 33

Expressions 33
Expressions, compile-time 11
Expressions, conditional 32
Expressions, restricted 23
Extensions 45
EXTERNAL attribute 21, 23, 24, 46
External procedures 19
External variable 22
Extracting source files (UNIX) 80

Factored declarations 21, 22
Fatal errors 49
Fbr format (TNIX) 75
Features and capabilities 1
Files, temporary 83
Final output (DOS) 72
Final output (TNIX) 75
Final output (UNIX) 67
FINDB builtin 43
FINDRB builtin 43
FINDRW builtin 43
FINDW builtin 43
FIX builtin 39
FLAGS builtin 43
FLOAT builtin 39
Formal definition of meta-language 59
Format of source 13
Format, object module 14
Function reference 18
Functions 18, 36

GET$REAL$ERROR builtin 48
Global tailoring changes (DOS) 72
Global tailoring changes (TNIX) 76
Global tailoring changes (UNIX) 68
GOTO statement 18, 31
Grammar 15
Group label 28
Group names 27

HALT statement 31
HEAD environment variable (DOS) 73
HEAD environment variable (TNIX) 76
HEAD environment variable (UNIX) 68
HIGH builtin 40, 46, 47

IABS builtin 41
Identifiers 36
IF block 29, 47
IF control 12
IF statement 28, 30
Implied base 24

INCLUDE control 11
Inexact reference 36, 39
INITIAL attribute 22, 24
Initialization file tailoring (DOS) 73
Initialization file tailoring (TNIX) 77
Initialization file tailoring (UNIX) 69
INIT$REAL$MATH$UNIT builtin 48
INPUT builtin 44
Install command (TNIX) 75
Installation, VAX/VMS 65
Installing 86/PC 85
Installing CLD files (VMS) 66
INT builtin 39, 46
INTEGER data type 24, 35
Intel OMF final output (DOS) 72
Intel OMF final output (TNIX) 75
Intel OMF final output (UNIX) 67
Internal procedures 19
INTERRUPT attribute 44, 46
Interrupt procedures 19, 44
Interrupt vector 19, 48
INTERRUPT$PTR builtin 44, 46
Introduction 1
Invocation options 3, 7
Invoking 86/PC 3, 6
INWORD builtin 44
Iterative DO statement 28, 36

Jump optimizer option (-J) 4
JUMPS option 9

LABEL attribute 21
Label definitions 32
Label reference 31
Label, group 27
LAST builtin 39, 46
Ld options 84
LENGTH builtin 39, 46
Lines per page 4
LINE_NUMBERS option 7
Linker (ld) options 84
Linker options, other 84
Linking option, split i/d 84
LIS file type 8
LIST control 12
List of error messages 49
Listing 86/PC 85
Listing controls 12
Listings 8
LITERALLY attribute 22
Literals in the meta-language 15

Local symbol record option (-d) 4
Local symbol record option (-L) 4
Local tailoring changes (DOS) 73
Local tailoring changes (TNIX) 76
Local tailoring changes (UNIX) 68
LOCKSET builtin 44
Logical names (VMS) 65
Long constant 23
LOW builtin 40, 46, 47
LQ_DWORD_DIV 14, 61
LQ_DWORD_MUL 14, 61

Machine flag, carry 40
Machine flags 43
Main programs 17
Manual organization 2
Manual pages, 86/PC 85
MAXA environment variable (DOS) 73
MAXA environment variable (TNIX) 76
MAXA environment variable (UNIX) 69
Maximum number of arguments (DOS) 73
Maximum number of arguments (TNIX) 76
Maximum number of arguments (UNIX) 68
MEMORY builtin 21, 44
MEMORY model option 8
Meta-language, formal definition 59
Meta-language, introduction 15
Models of computation 8
Module definition 17
Module level 17, 19, 23, 24
Module name 17
Modules 17
MOVB builtin 42
MOVE builtin 42
MOVRB builtin 42
MOVRW builtin 42
MOVW builtin 42
Multiply and divide 61, 63

Naming scope 17, 18, 27, 29, 31
Native commands (VMS) 66
Newline character 13
NOAT_VARIABLES option 8
NOBASED_VARIABLES option 9
NOJUMPS option 9
NOLIST control 12
Non-terminal symbols 15
NOPOINTERS option 9
NOPREPROCESS_ONLY qualifier 9
NOSUBEXPRESSIONS option 8
Null statement 31

Numeric constants 37

Object module 17
Object module format 14
Object module name format 3, 7
OFFSET$OF builtin 40, 46, 47
Operands, constant 34
Operators 33
Optimization 22
Optimization control option (-O) 4
Optimization, C compiler 84
Options to control the preprocessor 5, 9
Options, invocation 3, 7
Options, linker (ld) 84
Organization of manual 2
Other C compiler options 84
Other linker options 84
OUTPUT builtin 44
OUTWORD builtin 44
Overall operation 10
Overflow 28

P4 model option 8
P86 file type 7
Parameters of procedures 18
PARITY builtin 43
Path command (DOS) 72
Pccompile.sh shell script 82, 84, 85
Pcdefs.sh shell script 82, 83, 85
Pcinstall.sh shell script 82, 85
Pclink.sh shell script 82, 84, 85
Pcprint.sh shell script 82, 85
PI builtin 42
POINTER data type 25
POINTERS option 9
Preprocessor control options 5, 9
PREPROCESS_ONLY qualifier 9
Procedure class 19
Procedure declarations 18
Procedure parameters 18
Procedure scope 18
Procedure, typed 18
Procedure, untyped 18
Procedures 17
Procedures, external 19
Procedures, interrupt 19
Procedures, reentrant 19
Pseudo-function 39, 40, 41
PUBLIC attribute 21, 23
Public procedures 19

Q86 file type 7, 8

REAL data type 24
Recognition of statements 13
Redirecting standard error file 6
Reentrant procedures 19
References 35
Relational operators 34
Reserved words 13, 45
RESET control 12
RESTORE$REAL$STATUS builtin 48
Restoring delivery tape (UNIX) 79
Restoring the 86/PC delivery tape 82
Restoring the diskette (DOS) 71
Restoring the diskette (TNIX) 75
Restoring the tape (VMS) 65
Restricted expression 22, 23, 46
Restricted reference 24, 28, 30, 36
Return codes 6
RETURN statement 18, 31
ROL builtin 40
ROM model option 8
ROR builtin 40
Rules 15

S 25
SAL builtin 40
SAR builtin 40
Sarin utility 80
Sarin utility (UNIX) 79
SAVE$REAL$STATUS builtin 48
SCL builtin 40
Scope of names 17, 27, 29, 31
Scope of procedures 18
SCR builtin 40
Search path (DOS) 72
SELECTOR$OF builtin 40, 46, 47
Set command DCL command (VMS) 66
SET control 11
SETB builtin 42
SETW builtin 42
SET$INTERRUPT builtin 44
SET$REAL$MODE builtin 44
Severe errors 49
SHL builtin 40
SHR builtin 40
SIGN builtin 43
SIGNED builtin 39
Signed constants 23
Simple statements 30
SIZE builtin 39, 46
SKIPB builtin 43
SKIPRB builtin 43

SKIPRW builtin 43
SKIPW builtin 43
Source archive structure 81
Source format 13
Source listing option (-l) 3
Special statements 31
Split i/d linking option 84
SQRT builtin 42
Stack base 31, 41
STACK model option 8
Stack pointer 31, 41
Stack size 48
STACKBASE builtin 41
STACKPTR builtin 41
Standard error file, redirecting 6
Statement labels 21, 32
Statement labels in main programs 17
Statement recognition 13
STATEMENT_NUMBERS option 7
Status, completion 10
String constants 37
STRUCTURE attribute 25
Structure data type 46
SUBEXPRESSIONS option 8
Subroutines 18
SUBTITLE control 12
Supported operating environment (DOS) 71
Supported operating environment (TNIX) 75
Supported operating environment (UNIX) 67, 79
Supported operating environment (VMS) 65
Symbolic listing 8
Symbolic listing option (-a) 3
Symbolic listing option (-S) 4
Syntax checking option (-s) 3
Sys$86pc logical name (VMS) 65

TAIL environment variable (DOS) 73
TAIL environment variable (TNIX) 76
TAIL environment variable (UNIX) 68
Tailoring changes, global (DOS) 72
Tailoring changes, global (TNIX) 76
Tailoring changes, global (UNIX) 68
Tailoring changes, local (DOS) 73
Tailoring changes, local (TNIX) 76
Tailoring changes, local (UNIX) 68
Tailoring with environment variables (DOS) 72
Tailoring with environment variables (TNIX) 76
Tailoring with environment variables (UNIX) 68
Tailoring with initialization files (DOS) 73
Tailoring with initialization files (TNIX) 77
Tailoring with initialization files (UNIX) 69

Tape (VMS) 65
Tape format (UNIX) 79
Tar command (UNIX) 67
Target reference 36
Tektronix 8086 Assembler 63
Tektronix 856x 75
Tektronix LAS final output (DOS) 72
Tektronix LAS final output (TNIX) 75
Tektronix LAS final output (UNIX) 67
Temporary directory (DOS) 71
Temporary files 83
TIME builtin 42
TITLE control 12
TNIX 75
Type attributes 24
Typed procedure 18, 31, 39

UNDO statement 27, 28, 47
Unit of compilation 17
UNIX systems 67
UNSIGN builtin 39, 46
Untyped procedure 18, 30, 31, 39

Variables, compile-time 5, 9, 11
VAX 65
Version number of compiler (-V) 6
VMS 65

Warnings 49
Warnings, C compiler 84
WHILE statement 27
WORD data type 24
Word variable 40

XLAT builtin 42

ZERO builtin 43

\tmp directory (DOS) 71