Part One

80/PL Language
80/PC Compiler
Reference Guide

Table of Contents

1. Introduction

This portion of the Experts-PL/M™ manual describes the 80/PL™ programming language and the operation of the 80/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.

1.1 FEATURES AND CAPABILITIES

The 80/PL language is a superset of the PL/M-86 language. However, the target machines are the Intel 8080 and 8085 and the Zilog Z80. The Intel MCS-80/85 object module format is used for the resulting object programs. Significant new features of 80/PL include:

- Support of the WORD, INTEGER, and POINTER data types;
- Support of the PL/M-86 string handling builtins;
- 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 and LOW builtins as assignment targets;
- 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.

See Appendix A for a complete description of the differences.

The compiler operates under the VAX/VMS™, UNIX™, and PC-DOS operating systems.

The output of the 80/PC compiler is normally in the Intel MCS-80/85 Relocatable Object Module Format, facilitating its use with other development tools and easing integration of new software with existing object modules. Optionally, some versions of the compiler can produce output in Tektronix LAS format for use with various Tektronix relocation and linking tools.

2. General Information

This chapter provides general information to users of the 80/PL language and the 80/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 80/PC UNDER UNIX AND PC-DOS

Under the UNIX and PC-DOS operating systems, the 80/PC compiler is invoked by:

```
80pc [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 the 80/PL library of support routines (Section 2.6.1) and any other 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 80/PC are:

-F Generate inline code in some cases, rather than calls to out-of-line support routines. This will produce a larger object module, but one which should execute faster.

-L Generate local symbol and line number records in the object file for possible use by a run-time debugging system. Use the source file line numbers. Do not generate line number records for lines in include files.

-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.

-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.

-O Turn off common subexpression optimization. This will generate less efficient code.

-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.

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

-Z Allow the compiler to emit code specific to the Z80. The Z80 short jumps and word arithmetic will be used in addition to the 8080 instructions. The extra Z80 registers are not used.

-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.

-i Generate Intel MCS-80/85 Relocatable Object Module Format. This is normally the default, but the compiler may be installed so that the "-t" switch is the default.

-t Generate Tektronix LAS Object Module Format. 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.

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

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

-X*laaa* 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*.

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

-X*Saaa* 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".

-X*taaa* 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 80PC 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 80/PC UNDER VMS

Under the VMS operating system, the 80/PC compiler is invoked by:

```
80PC [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
/[NO]OBJECT[=file-spec]       /OBJECT
/[NO]OPTIMIZE=(options)       /OPTIMIZE=SUBEXPRESSIONS
/[NO]PREPROCESS_ONLY          /NOPREPROCESS_ONLY
/SYNTAX
/[NO]Z80_CODE                 /NOZ80_CODE
```

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 "Q80". The default file type for the source file is "P80".

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
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.

/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 "Q80".

/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]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]INLINE_CODE	specifies that the compiler is to generate inline code rather than call to out-of-line support routines

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.

/Z80_CODE
/NOZ80_CODE

Specifies that the compiler is allowed to generate code specific to the Z80. The Z80 short jumps and word arithmetic instructions will be used in addition to the 8080 instructions.

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 80/PC

The 80/PC compiler consists of a driver, named 80pc, and a number of phases which perform the actual compilations. The phases used in a normal compilation, in the order executed, are:

80pp	The preprocessor which handles the compile-time control language described in Section 2.4.
80p1	The initial syntax analyzer and declarations processor.
80p2	The final syntax analyzer, semantics processor, and run-time storage allocator.
80pcg	The code generator.
80pjo	The optional jump optimizer.
80pfo	The final output generator.
80psym	The symbolic lister, used when the "-S" option ("/MACHINE_CODE" qualifier) is present.
80pxrf	The cross-reference lister, used when the "-x" option ("/CROSS_REFERENCE" qualifier) is present.

2.4 THE 80/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 80/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 80/PL identifiers (Rule 91). 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 96) 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. 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

The RESET control has the form

```
$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.

2.5 80/PL SOURCE FORMAT

An 80/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 80/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 80/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 80/PC compiler uses the format of the Intel MCS-80/85 Relocatable Object Module Formats. Some versions of the compiler are optionally able to produce Tektronix LAS format object modules.

2.6.1 The Run-Time Support Library

Object modules produced by 80/PC generally call out-of-line routines to perform word and string operations. Calls on short routines such as word subtract may be replaced by inline code sequences by specifying the "-F" ("/OPTIMIZE=INLINE") invocation option (Section 2.1.1, Section 2.2.1).

These routines reside in the *80pl.lib* library which is distributed with the 80/PC compiler. Object modules produced by 80/PC must be linked with this library to produce executable programs.

3. Introduction to the Meta-Language

This manual presents the complete syntax for the 80/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 80/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 80/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 66) 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 37) 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 60) may appear in a main program. A HALT statement (Rule 64) implicitly follows the last statement of a main program.

4.2.1 Main Program Statement Labels

Statement labels (Rule 67) 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
- they may be the target of a GOTO statement (Rule 58) 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 66) 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 32) and no other attributes.

4.5 PROCEDURE TYPES

Procedures are either untyped or have one of the basic types (Rule 32).

Untyped procedures are frequently referred to as *subroutines*. They are invoked by a CALL statement (Rule 57). 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 85) 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 37) 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.

The optional number is ignored and no 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 67) 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 94) 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 24) 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 never be optimized.

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* | *restricted-expression.*

Initialization attributes can only be applied to variables which are not based (Rule 30) 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 94), 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 is assigned to the code segment and cannot be changed at execution.

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).

5.5.1 Restricted Expressions

A restricted expression has the syntax:

24. *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 80) may be the address of a long constant (Rule 81).

5.6 ELEMENT ATTRIBUTES

The syntax of an element attribute is:

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

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

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

5.6.1 The EXTERNAL Attribute

The EXTERNAL attribute is:

28. *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:

29. *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:

30. *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 88) gives an implied base which must be a previously defined word or pointer. The implied base will be used when an actual reference to the based item does not have an explicit base (Rule 90).

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:

31. *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:

32. *basic-type* = *BYTE* | *WORD* | *ADDRESS* | *INTEGER* | *POINTER* | *REAL*.

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

The POINTER type specifies an address of 16 bits. For compatibility with machines with addresses greater than 16 bits, the POINTER type is not the same as the WORD type.

The REAL type is recognized but not supported.

5.6.6 The STRUCTURE Attribute

The syntax of the STRUCTURE attribute is:

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

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

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

36. *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 27).

6. Executable Statements

The syntax of an executable statement is:

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

6.1 DO GROUPS

The syntax of a DO group is:

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

39. *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 67) 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:

40. *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:

41. *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 68) 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:

42. *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 88) is the loop index. The expressions (Rule 69) 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:

43. *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:

44. *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:

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

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

47. *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 68) 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:

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

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

50. *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:

51. *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 68) 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:

52. *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:

53. *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:

54. *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:

55. *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:

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

The right side expression (Rule 69) 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:

57. *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 88) 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:

58. *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.

6.4.4 The Null Statement

The syntax of the null statement is:

59. *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 43) and after the THEN or ELSE keywords in IF statements (Rule 45).

6.4.5 The RETURN Statement

The syntax of the RETURN statement is:

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

The form of the RETURN statement with the optional expression (Rule 69) 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:

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

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

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

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

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

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

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

The CAUSE$INTERRUPT statement generates the appropriate RESET instruction. The expression must be a constant operand (Section 7.3) from zero to seven (0-7).

6.5 ENDINGS

The syntax of an ending is:

66. *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:

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

Only executable statements may have statement labels.

6.7 COMPATIBLE TYPES

In assignment statements (Rule 56) and other similar situations, the right side must have a type that can be converted into the type of the left side. Bytes and words are compatible: bytes are converted to words by extending them with zeros; words are converted to bytes by truncation. Words and pointers are compatible and are the same size. Integers are only compatible with integers. When the right side is a constant, the context of the constant operand is integer.

6.8 CONDITIONAL EXPRESSION

The syntax of a conditional expression is:

68. *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:

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

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

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

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

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

74. *relop* = '<' | '<=' | '=' | '<>' | '>=' | '>'.

75. sum = term [{ '+' | '-' | *PLUS* | *MINUS*} term]*.

76. term = *secondary* [{ '*' | '/' | *MOD*} secondary]*.

77. secondary = ['+' | '-']* primary.

78. 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 i p" 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	i	p	name
+	b	w	i	–	unary plus
-	b	w	i	–	unary minus (negation)
*	w	w	i	–	multiplication
/	w	w	i	–	division
MOD	w	w	i	–	remainder
+	b	w	i	–	addition
-	b	w	i	–	subtraction
PLUS	b	w	–	–	add with carry
MINUS	b	w	–	–	subtract with carry
<	b	b	b	b	less than
<=	b	b	b	b	less than or equal to
=	b	b	b	b	equal to
<>	b	b	b	b	not equal to
>=	b	b	b	b	greater than or equal to
>	b	b	b	b	greater than
NOT	b	w	–	–	bitwise NOT (one's complement)
AND	b	w	–	–	bitwise AND
OR	b	w	–	–	bitwise OR
XOR	b	w	–	–	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	i	p
byte	b	w	–	–
word	w	w	–	–
integer	–	–	i	–
pointer	–	–	–	p

7.2 RELATIONAL OPERATORS

The relational operators

< <= = <> >= >

compare bytes, words, integers, and pointers. 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; 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:

79. *embedded-assignment* = *reference* ':=' *basic-expression*.

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

7.5 ADDRESSES

The syntax of an address is:

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

81. *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:

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

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

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

7.6.1 Function Reference

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

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

86. *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:

87. *target-reference* = *reference*.

If the target reference is a function reference (Rule 85), the function can only be one of the builtin pseudo-functions:

OUTPUT STACKPTR LOW HIGH

See Chapter 8 for further information.

7.6.3 Restricted Reference

The syntax of a restricted reference is:

88. *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:

89. *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 80) 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:

90. *explicit-based-reference* = [*basic-reference* |
'(' *expression* ')'] ['->' *basic-reference*] *.

Each basic reference following the arrow must be to a variable with the BASED attribute (Rule 30). If the based item was declared with an implicit base, that base is ignored.

The base expression can be an absolute number. 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:

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

92. *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:

93. *constant* = *string* | *number*.

94. *string* = ''' *string-character** '''.

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

96. *number* = *binary-number* | *octal-number* | *decimal-number* | *hex-number*.

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

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

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

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

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

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

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

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

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

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 89).

```
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, and UNSIGN functions change the type of their argument.

```
integer = INT (word)
integer = SIGNED (word)
word = DOUBLE (byte)
word = UNSIGN (integer)
```

8.3 DECIMAL ADJUSTMENT

The DEC function performs a decimal adjust on its argument.

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

8.4 ABSOLUTE VALUE

The IABS function returns the absolute value of its argument.

```
integer = IABS (integer)
```

8.5 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 or words. Bits shifted out go into the carry. Zeroes are shifted in.

ROL and ROR rotate bytes or words. Bits rotated out go into both the other end of the byte or word and into the carry.

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

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.

8.6 REFERENCING SUBFIELDS

HIGH and LOW reference the two bytes of a word. HIGH and LOW 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 variable.

```
byte = HIGH (word)
byte = LOW (word)
HIGH (ref) = byte
LOW (ref) = byte
```

HIGH used as a function returns the high byte of its argument. LOW used as a function returns the low byte of its argument.

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

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

8.7 THE STACK POINTER

STACKPTR references the hardware stack pointer register. STACKPTR can be used as a normal function or as a pseudo-function.

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

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

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

8.8 TIME DELAYS

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

```
call TIME (word)
```

8.9 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 or word. 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 words.

8.9.1 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)
```

8.9.2 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)
```

8.9.3 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)
```

8.9.4 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)
```

8.9.5 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)
```

8.10 FLAG VALUES

The flag functions return the values of the machine flags.

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

8.11 INPUT AND OUTPUT

The argument to INPUT and OUTPUT is a byte constant specifying one of the hardware ports. INPUT is a byte function which reads a byte from one of the hardware ports. OUTPUT is a pseudo-function 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.

```
byte = INPUT (con)
OUTPUT (con) = byte
```

8.12 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.

8.13 OTHER BUILTIN IDENTIFIERS

For compatibility, the following identifiers are recognized but treated as errors.

```
LOCKSET INWORD OUTWORD FIX FLOAT ABS OFFSET$OF
SELECTOR$OF BUILD$PTR STACKBASE
```

A. 80/PL and PL/M-80 Differences

80/PL is approximately a superset of both PL/M-80 and PL/M-86. Since 80/PL is intended for the 8080 programmer, the extensions to PL/M-80 which are part of PL/M-86 will be described first. Then the extensions which go beyond both PL/M languages will be described.

A.1 EXTENSIONS COMPATIBLE WITH PL/M-86

Most of PL/M-86 is supported by 80/PL. The most notable exceptions are in the area of interrupts and 8086 hardware support such as STACKBASE. These are not supported because they do not fit with the 8080 architecture.

A.1.1 New Data Types

80/PL supports WORD, INTEGER, and POINTER data types of PL/M-86 in addition to the BYTE and ADDRESS data types of PL/M-80. The POINTER data type is treated as it is in the SMALL model of computation of PL/M-86. Pointers, therefore, are in practice just like addresses.

80/PL recognizes the REAL data type of PL/M-86 but does not implement it. Error messages are given for REAL declarations and for real constants.

A.1.2 New Builtin Procedures

The following builtin procedures support the INTEGER data type.

INT	UNSIGN	SAL
SIGNED	IABS	SAR

The following builtin procedures perform character string operations. The support routine for the XLAT builtin is not reentrant.

MOVB	SETB	FINDB	SKIPB
MOVW	SETW	FINDRB	SKIPRB
MOVRB	CMPB	FINDW	SKIPW
MOVRW	CMPW	FINDRW	SKIPRW
XLAT			

A.1.3 The '@' Operator

The '@' operator, like the '.' operator, yields a location. For the '@' operator this location has the POINTER data type.

A.1.4 The CAUSE$INTERRUPT Statement

The CAUSE$INTERRUPT statement has the form:

```
CAUSE$INTERRUPT ( number ) ;
```

It emits the Restart instruction for the 8080. Number must be in the range 0 to 7 for the 8080.

A.2 EXTENSIONS BEYOND BOTH PL/M-80 AND PL/M-86

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

A.2.1 Reserved Words

80/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.2.2 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 80/PL but will in PL/M-80. It is therefore possible in 80/PL but not in PL/M-80 to include, in every module of a program, a single file which contains all the external declarations for the program.

A.2.3 The Interrupt Attribute

The interrupt number in the interrupt attribute is optional. If it is given, the number is ignored and no entry is made in the interrupt vector.

A.2.4 Restricted Expressions

A restricted expression (Rule 24) 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.

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

A.2.5 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.2.6 Builtin Functions as Assignment Targets

The builtin functions HIGH and LOW 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.2.7 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.2.8 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.3 UNSUPPORTED PL/M-80 AND PL/M-86 FEATURES

PL/M-80 can generate an absolute object file with no external references. This feature is not supported. Neither is the calculation of the stack size nor the attachment of interrupt procedures to the interrupt vector.

The features of PL/M-86 not supported are those that need the capabilities of the 8086. They are:

- The STACKBASE function.
- Word or variable unit numbers for the INPUT and OUTPUT functions.
- The INWORD and OUTWORD functions.

- The LOCKSET function.
- The REAL data type and the FIX, FLOAT and ABS functions.
- The SET$INTERRUPT and INTERRUPT$PTR functions.
- The SELECTOR$OF, OFFSET$OF, and BUILD$PTR functions.

B. Error Messages

The 80/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 80/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 80/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 80/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 80/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.

F ARG COUNT REQUEST

E ARGUMENTS NEEDED

E ARRAY ATTRIBUTE IS INCOMPATIBLE
E AT ATTRIBUTE IS INCOMPATIBLE
E AT LEAST ONE CASE REQUIRED
W AT LEAST ONE GLOBAL NAME WAS TRUNCATED
E ATTRIBUTE CANNOT BE USED WITHIN A PROCEDURE
F BAD BUS FILE (FIX TO ABS)
F BAD MODE IN WIDTH
E BASE VARIABLE IS UNDECLARED
E BASED ATTRIBUTE IS INCOMPATIBLE
E BUILTIN IS NOT ADDRESSABLE
E BYTE CONSTANT OVERFLOW
E BYTE OR WORD REQUIRED
E BYTE WORD OR INTEGER REQUIRED
S CALL STACK OVERFLOW; EXPRESSION TOO COMPLEX
F CALLED VFREE WITH ALL PAGES LOCKED
E CANNOT BE NESTED WITHIN EXTERNAL
E CANNOT BE NESTED WITHIN EXTERNAL PROCEDURE
E CANNOT BE NESTED WITHIN REENTRANT OR INTERRUPT
F CAN'T FIND /USR/BIN/SORT OR /BIN/SORT
F CAN'T FORK SORT
E COMPILER CONTROL SYNTAX
E CONSTANT OVERFLOW
E CONSTANT REQUIRED
E CONTROL IS OUT OF PLACE
F CONTROL STACK UNDERFLOW
F COULDN'T CREATE DICT FILE
F COULDN'T OPEN DICT FILE
F CURRNT HAS UNKNOWN LOCATION
F DANGLING NODE
E DATA ATTRIBUTE IS INCOMPATIBLE
F DICT OPEN ERROR
S DICT OVERFLOW
F DICT WRITE ERROR

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
F EOF BEFORE END OF MODULE
E EOF IN QUOTED STRING
E EQUAL EXPECTED
F ERROR WRITING 0 PAGE
W EXCESSIVE INTEGER VALUE
E EXPLICIT ARRAY DIMENSION REQUIRED
E EXPRESSION SYNTAX
E EXTERNAL ATTRIBUTE IS INCOMPATIBLE
E GOTO TARGET NOT DEFINED
E GOTO TARGET NOT REACHABLE
S I/O ERROR ON CLOSE
S I/O ERROR ON READ
S I/O ERROR ON SEEK
S I/O ERROR ON WRITE
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 MODE
S INIT STACK OVERFLOW
E INITIAL ATTRIBUTE IS INCOMPATIBLE
E INITIAL VALUE DOES NOT MATCH DATA TYPE
E INTEGER CONSTANT OVERFLOW
E INTEGER REQUIRED
E INTERRUPT ATTRIBUTE IS INCOMPATIBLE
E INTERRUPT PROCEDURE CANNOT BE TYPED
E INTERRUPT PROCEDURE CANNOT HAVE PARAMETERS
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
F LABEL ADDRESS ERROR
F LABEL DOES NOT MATCH PC
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)
F LITERAL STACK UNDERFLOW
E LITERALLY TYPE IS INCOMPATIBLE
F LOOPING IN EMIT OPERATION
F LOST SYNCHRONIZATION 1
F LOST SYNCHRONIZATION 2
F LOST SYNCHRONIZATION 3
E MAXIMUM LITERALLY NESTING EXCEEDED
E MISPLACED STATEMENT
E MISSING RIGHT PAREN
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
F NO PATH FOUND
W NO VALUE RETURNED FROM FUNCTION
F NODE SIZE TOO LARGE
F NODE STACK OVERFLOW
E NOT WITHIN A BLOCK
E NUMBER EXPECTED
E OPERAND MODES INCOMPATIBLE WITH OPERATOR
F ORIGIN SYNCHRONIZATION
F OUT OF RANGE GET_ADDR
S OUTPUT BUFFER OVERFLOW
E PREMATURE END OF FILE
F PREMATURE END-OF-FILE
F PREMATURE EOF
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 CONSTANTS NOT SUPPORTED
E REAL REQUIRED
E REAL TYPE IS NOT SUPPORTED
E RECURSIVE LITERALLY
E REENTRANT ATTRIBUTE IS INCOMPATIBLE
E REFERENCE REQUIRED
E RESTRICTED ADDRESS CANNOT BE BASED
E RESTRICTED ADDRESS REQUIRED
E RESTRICTED CONSTANT EXPRESSION REQUIRED
E RESTRICTED EXPRESSION REQUIRED
E RIGHT PARENTHESIS EXPECTED

F SEGMENT WITH NO NAME
E SEGMENT WRAPAROUND
E SIMPLE VARIABLE REQUIRED
E SIZE OF DATA EXCEEDS ALLOCATED SPACE
F SORT FAILED
E SOURCE LINE IS TOO LONG TO PROCESS
S STACK IMAGE OVERFLOW; BASIC BLOCK TOO COMPLEX
F STACK UNDERFLOW
W STACK, MEMORY OR COMMON ENCOUNTERED
E STACKBASE IS NOT SUPPORTED
F STATE STACK UNDERFLOW
E STATEMENT CANNOT 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
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
S TREE BUFFER OVERFLOW
E TYPE DEFINITION IS REQUIRED
E TYPED PROCEDURE REQUIRED
E UNCLOSED CONDITIONAL ASSEMBLY CONSTRUCTS
E UNDECLARED PARAMETER
W UNINDEXED ARRAY REFERENCE
F UNKNOWN BLOCK TYPE
E UNKNOWN COMPILER CONTROL
E UNKNOWN COMPILER CONTROL TYPE
F UNKNOWN INCOMING MACRO TYPE

F UNKNOWN MFD TYPE
F UNRECOGNIZED JUMP TYPE
E UNSUPPORTED BUILTIN FUNCTION
E UNSUPPORTED VARIABLE UNIT NUMBER
E UNSUPPORTED WORD INPUT OR OUTPUT
E UNSUPPORTED WORD UNIT NUMBER
W UNTYPED PROCEDURE REQUIRED
E VALUE RETURNED FROM SUBROUTINE
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 80/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. Linking With Tektronix Tools

Some versions of the 80/PC compiler can generate Tektronix-compatible object files (LAS format). However, an application program frequently consists of several object files which must be linked together. In any case, the linked application program must be assigned physical addresses in the target system. This chapter addresses linking an application program and assigning it physical addresses; both operations are performed with the Tektronix *link* program.

D.1 WRITING A LINKER COMMAND FILE

It is normally simplest to write an 8080 linker command file directly as is discussed in this section.

D.1.1 Important Symbols

The 80/PC compiler uses several symbols which are defined at link-time to resolve certain references.

HEAPBASEQQ	This symbol is always required. It is the address of the stack segment in Intel-format object files.
STKBASEQQ	This symbol is always required. It is the largest address corresponding to a byte in the stack (not the address of the first byte past the stack).
ENDREL	This is the Tektronix equivalent of the MEMORY array. The linker has virtually complete control over this symbol.
INSTRQQ	This is the class name of all code segments produced by 80/PC.
DATAQQ	This is the class name of all data segments produced by 80/PC.

D.1.2 Example 1

An application consists of an 80/PL main program object module residing in root.q, and two 80/PL support object modules residing in *first.q* and second.q. The application has <2700H bytes of code, <300H bytes of constants, <1800H bytes of data, <200H bytes of stack, and no references to the MEMORY array. The target machine will have ROM from locations 0-03FFFH, and RAM from locations 4800H-87FFH.

First, we define the link-time constants:

```
-D HEAPBASEQQ=8600H
-D STKBASEQQ=87FFH
```

Next, we define address ranges for the various classes:

```
-L class=INSTRQQ range 0040H-3FFFH
-L class=DATAQQ range 4800H-85FFH
```

The class names must be capitalized as shown. Since no memory range was split, it is not necessary to name the memory ranges. Finally, we define the object files to be linked:

```
-O root.q
-O first.q
-O second.q
```

This gives us the linker command file

```
-D HEAPBASEQQ=8600H
-D STKBASEQQ=87FFH
-L class=INSTRQQ range 0040H-3FFFH
-L class=DATAQQ range 4800H-85FFH
-O root.q
-O first.q
-O second.q
```

D.1.3 Example 2

An application consists of an 80/PL main program object module residing in *main.q*, three 80/PL support object modules residing in *sub1.q*, *sub2.q* and sub3.q, and a library of support routines residing in *support.lib*. The application requires a large amount of space for code, <0C00H bytes for data, and <4000h bytes for the stack (the application has several reentrant procedures, hence the large stack). The target machine will have the following memory layout:

```
ROM     locations 0-7FFFH, 0F000H-0FFFFH
RAM     locations 8000H-0EFFFH
```

(the ROM chip for locations 0-07FFH contains a complete interrupt-processing system, written previously; the ROM chip for locations 0F800H-0FFFFH will contain a monitor program; neither is actually available for the application itself).

First, we define the link-time constants:

```
-D HEAPBASEQQ=0B000H
-D STKBASEQQ=0EFFFH
```

Next, since the memory range for code is split, we name the part of memory available to code:

```
-m instructions=0800H-07FFFH 0F000H-0F7FFH
```

Next, we define address ranges for the various classes:

```
-L class=INSTRQQ range instructions
-L class=DATAQQ range 8000H-0AFFFH
```

Finally, we define the object files to be linked:

```
-O main.q
-O sub1.q
-O sub2.q
-O sub3.q
-O support.lib
```

This gives us the linker command file

```
-D HEAPBASEQQ=0B000H
-D STKBASEQQ=0EFFFH
-m instructions=0800H-07FFFH 0F000H-0F7FFH
-L class=INSTRQQ range instructions
-L class=DATAQQ range 8000H-0AFFFH
-O main.q
-O sub1.q
-O sub2.q
-O sub3.q
-O support.lib
```

D.2 INTERFACING TO THE 8540/8560

The 8080 ultimately communicates with the outside world through IN and OUT instructions. During program development, it is normally inconvenient to have specific i/o ports assigned and connected to the development system. Normally, i/o is performed through interface routines ("read", "write") which exist in two forms, one for the final stage (in which the appropriate IN and OUT instructions appear), and one for the development state (in which "magic" linkages to the debugging/operating system appear). This chapter addresses development-stage i/o on the Tektronix 8540.

D.2.1 The SVC Solution

A general solution to the i/o problem requires the use of the 8540 SVC (service call) facilities, described in Section 6 of the Tektronix *8540 System Users Manual*. Section D.4 and Section D.5 contain listings of some assembly-language and 80/PL procedures which may simplify the 80/PC user's task of interfacing with the SVC facilities. This section describes their function and use. It should be noted that object file format and SVC format are not necessarily the same: although 80/PC produces Tektronix LAS-format object files, the 8080 uses SAS-format SVC's and SRB's.

The procedures may be classified as

- an initialization routine
- SVC function routines
- SVC interface routines
- a four-byte utility routine
- SVC executors (for emulation modes 0, 1, and 2)

The SVC function and interface routines are written to use a single SVC (namely, SVC1); equally well, the SVC function and interface routines could be passed an SVC number (1-8 or 0-7) as an extra argument. The SVC executors will function properly in either case. The variable *mode* should be initialized to the desired emulation mode (0, 1, or 2).

D.2.1.1 Initialization Routine

The routine *initialize$srbs* initializes the SRB pointer vector in low memory. It should be called before any SVC function routine or SVC interface routine is called.

D.2.1.2 Four-Byte Utility Routine

The four-byte utility routine *swap4* interconverts between LAS-format and Intel-format 4-byte quantities.

An *LAS-long* is a four-byte integer; its most significant byte is stored at its lowest-addressed location, and its least significant byte is stored at its highest-addressed location. An *Intel-dword* is also a four-byte integer; its least significant byte is stored at its lowest-addressed location, and its most significant byte is stored at its highest-addressed location. Since 80/PL has no dword data type, an Intel-dword is most conveniently entered as a two-element array of words.

Swap4 takes two pointers as arguments (to the source value and the target value), and places the source bytes into the target, in reversed order.

D.2.1.3 SVC Function Routines

The SVC function routines take arguments from the referencing 80/PL program. The routines could be rewritten to also accept an SVC number (a byte or word argument). The SVC function routines pass a function number and their arguments to an SVC interface routine, and optionally extract information from the appropriate SRB and return it. The SVC function routines are not absolutely necessary; an industrious programmer could define literals for the various function numbers, and just call the SVC interface routines.

Starred routines do not actually appear in the listing; their definition should be obvious from the other routines.

D.2.1.4 Zero-Argument Routines

The following routines take no arguments.

```
svc$abort           1fh - abort program
svc$exit            1ah - exit program
svc$last$coni       11h - get last CONI character
svc$log$error       09h - log error message
svc$read$clock      16h - read program clock
```

Note that `svc$log$error` logs the *previous* error status.

D.2.1.5 One-Argument Routines (Pointer)

The following routines take a single argument, a pointer, which points at a RETURN-terminated filespec, specifying a load file to load. A pointer value is returned: the transfer address.

```
svc$load$ovl            17h - load overlay
```

D.2.1.6 Two-Argument Routines (Byte, Pointer)

The following routines take two arguments. The first, a byte, is the channel number. The second, a pointer, points at a RETURN-terminated name or filespec.

```
    svc$assign$channel      10h - assign channel
    svc$create$file         90h - create file
    svc$open$for$read       30h - open for read
*   svc$open$for$update     70h - open for read or write
*   svc$open$for$write      50h - open for write
```

D.2.1.7 Three-Argument Routines (Byte, Pointer, Pointer)

The following routines take three arguments. The first, a byte, is the channel number. The second, a pointer, points at a dword file offset. The third, a pointer, points at a dword which will hold the returned value, the new file offset (after the seek).

```
    svc$seek$rel$to$0       44h
*   svc$seek$rel$to$eof     64h
*   svc$seek$rel$to$here    24h
```

D.2.1.8 Three-Argument Routines (Byte, Byte, Pointer)

The following routines take three arguments. The first, a byte, is the channel number. The second, a byte, is a number of bytes or characters to read or write. The third, a pointer, points at a line or buffer. All the routines return a word value: the number of bytes or characters read or written.

```
    svc$read$asc$go         81h - read ascii and proceed
    svc$read$asc$wait       01h - read ascii and wait
*   svc$read$bin$go         41h - read binary and proceed
*   svc$read$bin$wait       c1h - read binary and wait
*   svc$rewrite$asc$go      a2h - rewrite ascii and proceed
*   svc$rewrite$asc$wait    22h - rewrite ascii and wait
*   svc$rewrite$bin$go      e2h - rewrite binary and proceed
*   svc$rewrite$bin$wait    62h - rewrite binary and wait
    svc$write$asc$go        82h - write ascii and proceed
    svc$write$asc$wait      02h - write ascii and wait
*   svc$write$bin$go        42h - write binary and proceed
*   svc$write$bin$wait      c2h - write binary and wait
```

D.2.1.9 Three-Argument Routines (Byte, Byte, Pointer)

The following routines take three arguments. The first, a byte, is the number (ordinal) of the desired parameter. The second, a byte, is the maximum size of the parameter, in bytes. The third, a pointer, points at a line or buffer which will hold the argument.

```
*   svc$get$cmd$parm    13h - get command line parameter
*   svc$get$exec$parm   16h - get execution line parameter
```

D.2.1.10 SVC Interface Routines

The SVC interface routines accept an SVC function number and a number of other arguments. These routines set up an SRB and then call *svcgo*. *Svcgo* calls the assembly-language SVC executor appropriate to the selected emulation mode.

The following table lists the arguments for the various SVC interface routines.

svcgo	(no arguments)
svcx	function (byte)
svcxb	function (byte), buffer (pointer)
svcxcb	function (byte), channel (byte), buffer (pointer)
svcxclb	function (byte), channel (byte), length (byte), buffer (pointer)
svcxd	function (byte), offset (pointer)
svcxplb	function (byte), position (byte), length (byte), buffer (pointer)

D.2.1.11 SVC Executors

The SVC sequence for emulation modes 0 and 1 requires an OUT instruction followed by one NOP instruction; the sequence for mode 2 requires an OUT instruction followed by two NOP instructions. Since the compiler cannot generate arbitrary NOP's, these procedures must be written in assembly language. The assembly-language routine *svcall1* performs the SVC sequence for emulation modes 0 and 1; *svcall2* performs the sequence for mode 2.

D.3 POSSIBLE TEKTRONIX LINKER ERROR MESSAGES

This section describes some of the common linker errors and the situations which provoke them. It assumes that fairly standard linker commands are used; users sufficiently sophisticated to use linker features not described in Appendix D are assumed to need no further instruction here.

D.3.1 Section Names

The compiler names sections in the following manner:

I.*module*	the code section for *module*
C.*module*	the constant section for *module*
D.*module*	the data section for *module*
A.*module*	the absolute section for *module* (not always present)

If the section name would be longer than 16 characters, the first 6 and the last 10 characters of the name are used.

D.3.2 Typical Errors

Errors which are commonly encountered include:

- **link:100 (S) Name** *symbol* **in section** *section* **previously defined** – *Symbol* is declared "public" in *section*. It was either declared "public" in some other module, or it is one of the class names (INSTRQQ, DATAQQ) or address symbols (HEAPBASEQQ, STKBASEQQ).
- **link:110 (E) No memory allocated to** *section* – The memory range assigned to the *section*'s class of segments (code, constant, or data) is not large enough to hold all of them. *Section* is effectively still relocatable. The memory range for *section*'s class should be increased, or *section*'s size should be decreased.
- **link:114 (E) Absolute section** *section* **conflicts with -L switch** – *Section* is absolute, it appeared in a -L command, and the address in the -L command doesn't match the address of *section*. Remove the offending -L command.
- **link:115 (E) Truncation error at** *address* – A 16-bit address-like quantity does not fit in 16 bits. *Address* is the physical address of the 16-bit quantity. It is *not* the address of the referenced item (the target). Typically, it is the address of the offset in an assembly-language instruction, which is 1-2 greater than the address of the first byte in the instruction.
- **link:118 (W) Transfer address undefined** – The object modules did not include a main program, and no -x command appeared. This may be the desired situation.
- **link:119 (W) Processor changed from** *family-1* **to** *family-2* – Different Tektronix processors include different microprocessors in the same family. If both families contain the desired target microprocessor, this warning is innocuous.
- **link:125 (W) Reserved name** *symbol* **used incorrectly** – This appears to occur when a user program defines ENDREL to be a public datum or procedure. The linker has reserved the definition of ENDREL to itself.
- **link:128 (E) Absolute or symbol file section** *section* **cannot be relocated** – *Section* is an absolute section, and it appeared in a -L command. This message appears even if the address in the -L command is appropriate to *section*. Remove the offending -L command.

D.4 ASSEMBLY-LANGUAGE ROUTINES FOR SVC'S

This section gives the Tektronix-assembler source for SVC routines for the 8080.

D.4.1 Utility Routines

These are utility routines used by other routines.

```
        list    dbg,xref,sym
        name    svcsupport1
        section i.svcsupport1,class=INSTRQQ
        global  svcall1,svcall2

;*  swap (v):  return a word value with the bytes swapped
;*

swap    mov     l,b     ; (bc) = value
        mov     h,c
        ret


;*  swap4 (source-pointer, target-pointer):  place the
```

```
;*              4-byte source value in the target, with
;*              with the bytes reversed (convert between
;*              Intel and SAS dwords)
;*

swap4   inx     d
        inx     d
        inx     d
        ldax    b
        inx     b
        stax    d
        dcx     d
        ldax    b
        inx     b
        stax    d
        dcx     d
        ldax    b
        inx     b
        stax    d
        dcx     d
        ldax    b
        stax    d
        ret

        end
```

D.4.2 SVC Executors

These are the procedures to perform SVC's in emulation modes 0, 1, and 2 on the Tektronix 8540.

```
        list    dbg,xref,sym
        name    svcsupport2
        section i.svcsupport2,class=INSTRQQ
        global  svcall1,svcall2

;*  svcall1 (svc-port):  do a mode-1 svc
;*
;*  svcall2 (svc-port):  do a mode-2 svc
;*

svcall1 mvi     l,0c9h          ; 'cet' instruction in (l)
        jmp     merge
svcall2 lxi     h,0c900h        ; 'nop' in (l), 'ret' in (h)
merge   push    h
        mvi     b,0             ; port in (c), 'nop' in (b)
        push    b
        mvi     b,0d3h          ; port in (c) (no longer needed),
                                ; 'out' in (b)
        push    b
        lxi     b,retadd        ; return address from built-up
                                ; procedure
        push    b
        lxi     h,3
        dad     sp              ; starting address of built-up
                                ; procedure
        pchl
retadd  lxi     h,6             ; prune built-up procedure
                                ; from stack
```

```
        dad     sp
        sphl
        ret

        end
```

D.5 80/PL ROUTINES FOR SVC'S

This section gives the 80/PL source for SVC routines for the 8080.

```
svc$test:
        do;

        /*******************************
         *                             *
         * Suitable for 8080/8085 Only *
         *                             *
         *******************************/

        declare dot literally '.',
                ptr literally 'word';

/*      "Index" is the svc we'll be using.  It's declared
        literally, although it could be passed through as
        an argument to the procedures.
*/
        declare index literally '0';

        declare svc1 literally '0f7h',
                svc2 literally '0f6h',
                svc3 literally '0f5h',
                svc4 literally '0f4h',
                svc5 literally '0f3h',
                svc6 literally '0f2h',
                svc7 literally '0f1h',
                svc8 literally '0f0h';

        declare svc (8) byte public data (
                svc1, svc2, svc3, svc4, svc5, svc6, svc7, svc8);

        declare srb (8) structure (
                fn byte,
                chan byte,
                status byte,
                four byte,
                count byte,
                lth byte,
                bufp ptr) public;

/*      "Mode" is the mode (0, 1, 2) of the SVC's.  It
        should correspond to the value selected with the
        debugger EM command.  It may be declared public
        to make it easier to find should it be necessary
        to modify its value at debug-time.
*/
        declare mode word initial (0);
svcall1:
        procedure (n) external;
        declare n byte;
```

```
end svcall1;

svcall2:
        procedure (n) external;
        declare n byte;
end svcall2;

swap:
        procedure (v) word external;
        declare v word;
end swap;

swap4:
        procedure (sp, tp) external;
        declare (sp, tp) ptr;
end swap4;


/*
 *  initialize$srbs
 *
 *      This routine must be called before any svc is used!
 */

initialize$srbs:
        procedure public;

        declare v (8) word at (40h);

        v(0) = swap (dot srb(0));
        v(1) = swap (dot srb(1));
        v(2) = swap (dot srb(2));
        v(3) = swap (dot srb(3));
        v(4) = swap (dot srb(4));
        v(5) = swap (dot srb(5));
        v(6) = swap (dot srb(6));
        v(7) = swap (dot srb(7));
end initialize$srbs;


svc$abort:
        procedure public;

        call svcx (1fh);
end svc$abort;


svc$exit$program:
        procedure public;

        call svcx (1ah);
end svc$exit$program;


svc$last$coni:
        procedure byte public;
```

```
        call svcx (1fh);
        return srb(index).four;
end svc$last$coni;


svc$log$error:
        procedure public;

        call svcx (09h);
end svc$log$error;


svc$read$clock:
        procedure word public;

        declare w based * word;

        call svcx (1fh);
        return (swap ((dot srb(index).four)->w));
end svc$read$clock;


svc$assign$channel:
        procedure (c, b) public;

        declare c byte, b ptr;

        call svcxcb (10h, c, b);
end svc$assign$channel;


svc$create$file:
        procedure (c, b) public;

        declare c byte, b ptr;

        call svcxcb (90h, c, b);
end svc$create$file;


svc$open$for$read:
        procedure (c, b) public;

        declare (c, b) word;

        call svcxcb (30h, c, b);
end svc$open$for$read;


svc$seek$rel$to$0:
        procedure (c, op, r) public;

        declare c byte, (op, r) ptr;
```

```
        declare w based * word;

        call svcxcd (44h, c, op);
        call swap4 (dot srb(index).count, r);
end svc$seek$rel$to$0;


svc$load$ovl:
        procedure (b) ptr public;

        declare b ptr;

        declare w based * word;

        call svcxb (17h, b);
        return swap ((dot srb(index).four)->w);
end svc$load$ovl;


svc$read$asc$go:
        procedure (c, m, b) byte public;

        declare (c, m) byte, b ptr;

        call svcxclb (81h, c, m, b);
        return srb(index).count;
end svc$read$asc$go;


svc$read$asc$wait:
        procedure (c, m, b) word public;

        declare (c, m) byte, b ptr;

        call svcxclb (01h, c, m, b);
        return srb(index).count;
end svc$read$asc$wait;


svc$write$asc$go:
        procedure (c, m, b) word public;

        declare (c, m) byte, b ptr;

        call svcxclb (82h, c, m, b);
        return srb(index).count;
end svc$write$asc$go;


svc$write$asc$wait:
        procedure (c, m, b) word public;

        declare (c, m) byte, b ptr;

        call svcxclb (02h, c, m, b);
```

```
        return srb(index).count;

end svc$write$asc$wait;



svcgo:
        procedure public;

        declare a byte;

/*      The next line is not necessary; it simply transfers
        the function value through a register (A).  It is
        useful during debugging on the 8540:  if the user
        sets a breakpoint on the "if (mode < 2);" statement,
        the function value is immediately visible in the
        register display (one need not find the proper
        address in the srb vector and execute a debugger
        D command).  It is most useful when index is a
        variable rather than a literal.
*/
        a = srb(index).fn;

        if (mode < 2);
                call svcall1 (svc(index));
        else;
                call svcall2 (svc(index));
        endif;
end svcgo;

svcx:
        procedure (fn);

        declare fn word;

        srb(index).fn = fn;
        call svcgo;
end svcx;

svcxb:
        procedure (fn, b);

        declare fn byte, b ptr;

        srb(index).fn = fn;
        srb(index).bufp = swap (b);
        call svcgo;
end svcxb;

svcxc:
        procedure (fn, c);

        declare (fn, c) word;

        srb(index).fn = fn;
        srb(index).chan = c;
        call svcgo;
end svcxc;

svcxcb:
```

```
        procedure (fn, c, b);

        declare (fn, c) byte, b ptr;

        srb(index).fn = fn;
        srb(index).chan = c;
        srb(index).bufp = swap (b);
        call svcgo;
end svcxcb;

svcxcd:
        procedure (fn, c, op);

        declare fn byte, c byte, op ptr;

        srb(index).fn = fn;
        srb(index).chan = c;
        call swap4 (op, dot srb(index).count);
        call svcgo;
end svcxcd;


svcxclb:
        procedure (fn, c, l, b);

        declare (fn, c, l) byte, b ptr;

        srb(index).fn = fn;
        srb(index).chan = c;
        srb(index).lth  = l;
        srb(index).bufp = swap (b);
        call svcgo;
end svcxclb;

svcxplb:
        procedure (fn, x, l, b) public;

        declare (fn, x, l) byte, b ptr;

        srb(index).fn = fn;
        srb(index).four = x;
        srb(index).lth  = l;
        srb(index).bufp = swap (b);
        call svcgo;
end svcxplb;
end;
```

Index