SECTION II

Program Development Support

2.1. Introduction

Although commonly called a "compiler", the software described in this manual is more than that term alone implies. It is actually an integrated series of programs which form a "program development system". The architectural design and implementation of this system is such that it can be (and has been) hosted on many different CPUs and operating systems. With these ambitious requirements and resulting power come complexities. If you are a first-time user, you will want to develop a global understanding of the system first. With this in mind, the following is designed to provide an awareness of the architecture of the system, at least to the level of your program development needs.

2.1.1. A Basic View of the Compiler

The simplest way to view this program development system is as a "black box" compiler. Conceptually, this looks like:

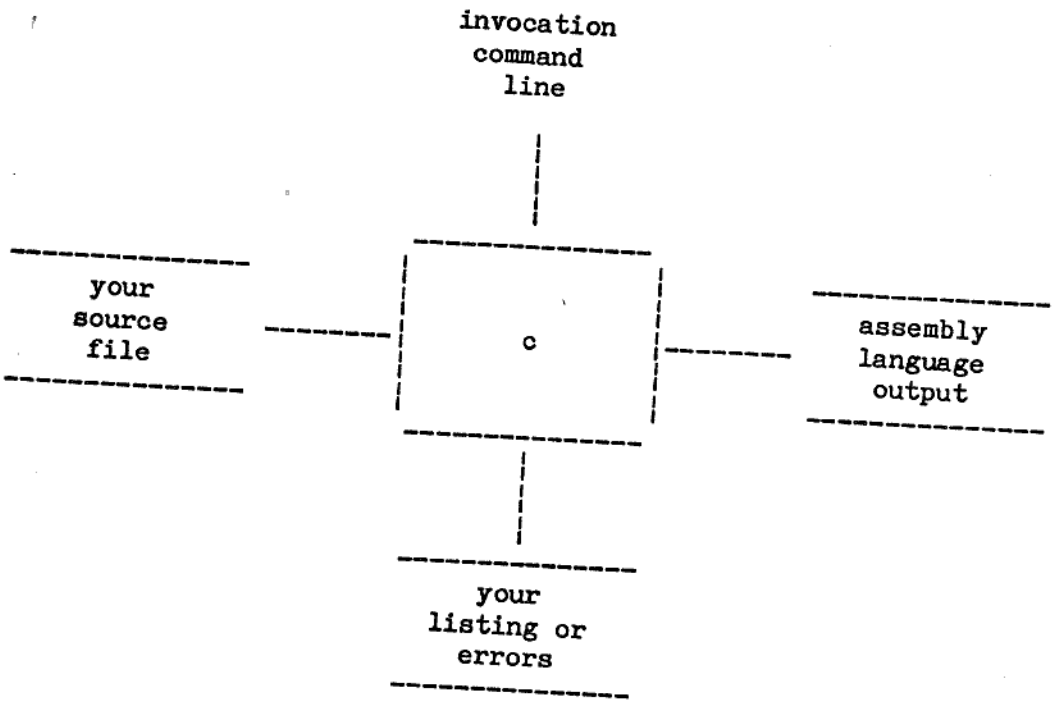

Figure II - 1

This shows the compiler as a single "program", called c. You invoke it with a command line on which you provide the name of your source file and some control information. It builds a sequential file of assembly language statements which you must assemble and link on your SSP/36 system. Your input source file could be in C, Pascal, or assembly language. The system is designed to give users great power and flexibility right up front, without any special modifications. If this view of the compilation process is sufficient for your needs, you should read the following tutorials in Section II of this manual to

help get you started: section 2.2, "Compiling Programs", section 2.5, "Generating Program Listings", and section 2.7, "Using the Debugger". You will not need to look any further into the "black box" until and unless you want to do something special or unusual.

This architecture and its component programs have been evolving since 1978, when Whitesmiths produced its first compiler. Without going into the design details behind it, you should be aware of the following:

1) All the individual parts of the system are designed and coded with strict adherence to the style and portability guidelines described in our C Language Manual.

2) All of the parts of the system are reusable. By replacing the code generator, assembler, and low-level libraries (CPU-specific and OS-specific) we can provide the identical program development system for many execution environments (Motorola 68000, 68010, 68020; Intel 8086, 80186, 80286; DEC VAX; PDP-11; IBM System/36; IBM 370, and many more).

3) This design approach allows us to easily move and package all of the parts on different host environments so that code can be generated for the same execution environment, or for different ones.

The following subsection, entitled "The Program Development System Architecture", describes the architecture of Whitesmiths' program development system for the DOS/8086 to SSP/36 Cross Compiler.

2.1.2. The Program Development System Architecture

If you look inside the "black box" described above, you will see several cooperating programs and some additional files which are controlled by the c driver program. Figure II - 2 shows the relationships among the most important of them.

As the diagram shows, the c driver is but one control program which invokes several others in order to compile your source. Each of these programs (along with some others not shown here) is individually described on a manual page in Section III or Section IV of this manual. The following is an overview of how they interact:

1) c uses its command line invocation in conjunction with a "prototype" file to control the whole process.

2) ptc reads Pascal source and translates it into C source. Note that if you have a C compiler only, ptc is not included.

3) pp, cp1, and p236 are the core of the compiler, turning C source into assembly language source.

4) The IBM ASM macro assembler translates assembler source into object modules.

5) The IBM OLINK overlay linker combines object modules given as direct input to it, scans libraries for other necessary objects, and incorporates all of them into one load module. It is used mostly from the procedure members CALINK, CEALINK or CWLINK.

6) lm and pr produce listings using the original source files in conjunction with control information saved by various other programs.

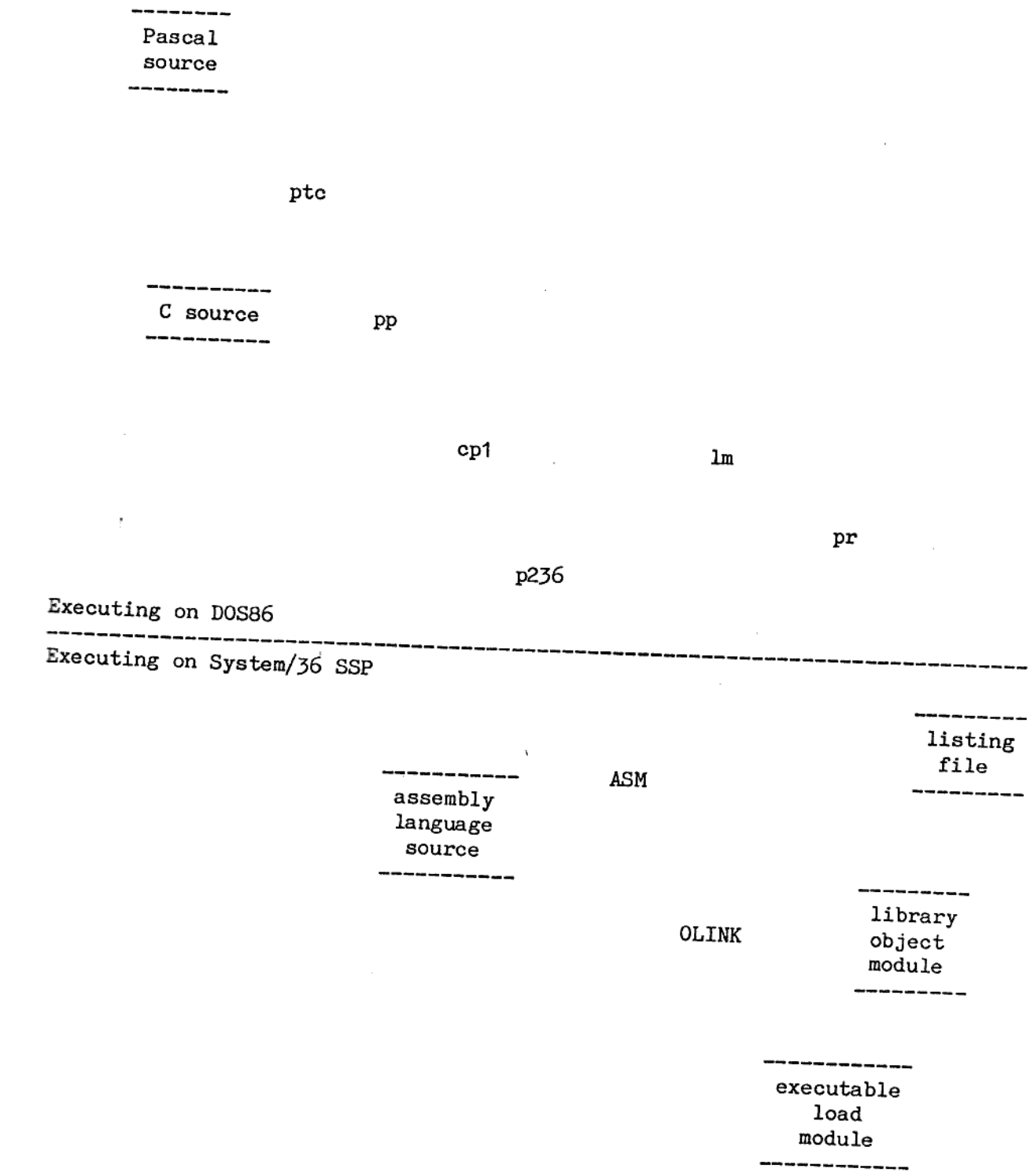
Pascal source
ptc
C source
pp
cp1
lm
pr
p236
Executing on DOS86
Executing on System/36 SSP
listing file
assembly language source
ASM
library object module
OLINK
executable load module

Figure II - 2

2.2. Compiling Programs

This section demonstrates in simple terms how to invoke the compiler driver, and describes the behavior of some of its more commonly used flags. It covers only a few of the many possible combinations of output files, listings, and executable image configurations available using the standard program development system. For a complete description of the driver and associated flags, see the c manual page in Section III of this manual. For an introduction to the "prototype file" that directs the compilation process, see section 2.3, "Introduction to Prototype Files". Advanced support for creating your own "programmable flags" and customizing the compilation process itself is available in section 2.4, "Modifying Prototype Files".

To invoke the compiler driver without giving it any special instructions, type c on the command line, followed by the name of your source file, as shown below. (For the purposes of this tutorial, the source file used is echo.c.)

```
C> c echo.c
```

The compiler will respond as follows:

```
C> echo off
echo.c:

C>
```

This simple command line is sufficient for c to compile your program using the default information obtained by following a series of commands found in the "prototype file" that the driver opens as soon as it is invoked. The prototype file, generally speaking, directs c to invoke the compiler passes -- pp (preprocessing), p1 (parsing), and p236 (code generation). The assembly language source file must then be moved to System/36 to assemble and link the resulting code to create an executable file. As the driver calls the utilities to perform these steps, it passes the program name (echo) along, changing the suffix from step to step using extensions specified in the prototype file, or that you can specify on the command line. When processing is completed, your operating system prompt will return and you will be ready to move the resulting assembly language file to System/36 (provided, of course, that no fatal errors were detected). The default name for the assembly language file created by the driver (which is target operating system-dependent) can be changed to be anything you like, as explained below. Note also that your C source files must end with a .c suffix or the compiler will not process them.

Echoing of Process Names During Compilation

It may often be useful to observe the progress of compilation by having the compiler echo the name of each step in the process as it is executed. To turn on this feature, invoke the compiler driver with the following command line:

```
C> c -v echo.c
```

Compilation will be performed in "verbose" mode, as specified by the -v flag. Prior to executing each command, the compiler will print out the command being executed, along with its arguments, to the "standard output".

Invoking the Debugger

Another very useful feature of the program development system is the cdb source-level debugger. cdb works by binding special libraries to your program during linking, which are activated when the program is run. The debugger is most easily enabled right from the driver command line. To do so, type:

```
C> c -ddebug echo.c
```

Saving the Output of Intermediate Passes

The usual result of a successful compilation is a single executable file. However, the code generated at one or more of the intermediate steps along the way may also be of interest. These intermediate files are easy to save. For example, the following request to the driver:

```
C> c +1 echo.c
```

will save the object module output of cp1. Instead of overwriting the temporary file that the compiler creates (which has the suffix .1), it will be left in place as compilation proceeds. The flag +* tells the compiler to save each temporary file with a suffix specified by *, and to halt processing after the step that generated the last file so requested. Each file is designated by its suffix. In this instance, the object file (with a .1 suffix), will be saved, and compilation will stop after the object file is generated.

Compiling Pascal Programs

To compile a Pascal program, invoke the c driver as you ordinarily would. The prototype file will invoke ptc, (the Pascal-to-C translator), and associated libraries, when it encounters the .p suffix to an input file on the command line. If you do not have a Pascal compiler, c will be totally unable to process files with .p suffixes. If you have a Pascal compiler, the following command line will invoke it:

```
C> c fahr.p
```

Moving the Assembly Language Source to System/36

If you are using PC-Support to transfer files between the PC and System/36 you may use the batch file TOWRK36.BAT. It will transfer a file MAIN.ASM to a source member MAIN in the library WRKLIB on System/36. Change the batch file if you want to transfer the file to another library.

Assembling and Linking Programs on System/36

To assemble and link programs compiled on your PC to run on System/36, you must have to switch to System/36 mode (through the use of a "hot key sequence" if you are using IBM PC-Support). If your session library is not WRKLIB, change it by typing:

```
SLIB WRKLIB
```

Then you assemble the file with the command:

```
ASM MAIN
```

You may then link the object member with the supplied procedure file **CEALINK:**

```
CEALINK MAIN
```

The OLINK procedure file CEALINK links your program using the Extended ANSI library. The OLINK procedure file CALINK links your program using the ANSI library.

You may load and run the linked load member by the following OCL-statements:

```
// LOAD MAIN
// RUN
```

2.3. Introduction to Prototype Files

When the compiler driver is invoked, it looks for the appropriate "prototype file" to give it specific directions on what steps to perform to complete the compilation process. The purpose of the prototype files is to make it easier to customize the steps performed during compilation. Instead of working with the compiler driver itself, only the simpler, human-readable prototype file need be changed.

This tutorial is a very general overview of what a typical prototype file looks like and how it influences the compiler driver. It should enable most users to read and make simple modifications to a prototype file. However, much of the detail important to those making major changes to prototype files, establishing a new default prototype file on your system, or writing one from scratch, is of necessity omitted. Users who need more information about these kinds of issues should read this essay and then refer to section 2.4, entitled "Modifying Prototype Files".

Prototype files are written in a "non-procedural" language -- one with no real control structures. It is the lines in the prototype file, in combination with the flags and the suffixes of the files on the command line, that determine the results of compilation.

The example on the next page, which is referred to in the discussion to follow, illustrates the relationship between a typical prototype file and the actual processing performed on source files by the compiler driver.

Given the following prototype file:

```
sp:pp -o (o) (i)
asm:
c:pp -o (o) {lincl?+lincl} -x (i)
1:cp1 -o (o) -cmu -b0 -n6 -r0 {debug?+debug} (i)
2:p236 -o (o) {listcs?+list} (i)
asm:
```

entering the command

```
c hello.c
```

would cause the driver to create and run the following processes:

```
pp -o ctempc.c0 -x hello.c
cp1 -o ctempc.c1 -cmu -b0 -n6 -r0 ctempc.c0
p236 -o hello.asm ctempc.c1
```

hello.c is compiled into hello.asm.

Note: DOS/86 is the host operating system assumed in this example.

2.3.1. Syntax

Any given line of a prototype file falls into one of three categories. It is either:

1) a step in the compilation process,

2) a partial step (one line of a multi-line step), or

3) a comment or an empty line.

Steps

A "step" is a line that begins with any number of alphanumeric characters followed by a colon. This alphanumeric sequence is referred to as a label. The compiler driver interprets labels and uses them to create files in different categories. A line beginning with c: means "files with a .c suffix enter at this step to begin processing".

From the introductory example, the line to run all files with a .c suffix (C source files) through the preprocessor pass, with the -x flag to pp turned on, looks like:

```
c:pp -o (o) -x (i)
```

Note: the syntax behind (o) and (i) is explained below, under the subheading "Variables".

Multi-line Steps

A multi-line step follows the same syntax rules as a one-line step. It begins with a label and consists of invocations of several different programs or utilities that together complete some process. The second (and any subsequent) lines in a multi-line step is distinguished from an ordinary one-line step by its label, which consists of a tab character or space without a colon, followed by something other than a # (to distinguish it from a comment, as explained below).

Here is an example of a multi-line step:

```
c:echo rootname.suffix
     echo "compilation begun"
     echo "compile to assembler only"
```

The following is an example of a multi-line step from a simplified prototype file, expanded to include some general comments (the lines starting with #) and whitespace:

```
2:p236 -o (o) (i)
#    process output of the parsing pass with
#    the -o flag to the code generator turned on

     copy (o) objects
#    copy the contents of the intermediate files created
#    by the substep above to the directory "objects",
#    thus saving backup copies of the intermediate output
```

Comments and Whitespace

A comment is any character sequence in which the first non-whitespace character is a "pound sign" #. A comment can appear on a line by itself, or set off on the right side of a "step". Everything following the # on a line is interpreted as a comment, and is ignored by the compiler driver. An empty line, like a comment, is also ignored.

Continuation

To indicate the continuation of a step or partial step on the following line, a backslash \ must be the last character on the line. Total logical line length may not exceed 512 characters.

2.3.2. Variables

Since prototype files are input-driven, there must be some means for accepting arguments from the compiler driver command line and passing them between steps within the file. This is accomplished by the character sequences (i), (o), and (r), which perform manipulation on file names to guide compilation from one step to the next.

The sequence (i), when it appears in a step within a prototype file, expands to the name of the input file (i.e. the file currently being acted upon by the driver, or a temporary filename if the previous step under consideration did not save an intermediate). If, for example, the compiler is invoked by the command

```
C> c file.c
```

and the current step in the prototype file appears verbatim as:

```
c:pp (i)
```

the request for action passed to the driver program would be "run the preprocessor pass of the compiler on the input file". The input file name is then expanded within the driver to file.c.

The sequence (o) expands to a temporary filename, or a name derived from the current file under consideration, with its suffix replaced by the label of the next step in the prototype file. It means "insert the output file name here", as in the following example:

```
1:cp1 -o (o) (i)
o:
```

The sequence (r) expands to the root (the filename without any existing suffix) of the file currently under consideration.

The following example shows how arguments to the driver are passed via the prototype file to the indicated programs under DOS for execution under SSP/36. The labels shown determine what the suffixes of any newly-created files will be. The

```
C>  c myfile.c
```

causes the prototype file commands:

```
   sp:pp -x -o (o) (i)
    1:cp1 -o (o) (i)
    2:p236 -o (o) (i)
  asm:
```

to expand to:

```
pp -x -o tmp1 myfile.c
cp1 -o tmp2 tmp1
p236 -o myfile.s tmp2
```

2.3.3. Using Programmable Flags

The programmable flag feature allows users to specify their own programmable flags to the compiler passes. The sequence:

```
{<flagname>?[<string1>][:<string2>]}
```

tests for the presence of a programmable flag on the command line.

The <flagname> string specifies the actual command line option to test for. If the option is present, <string1> replaces the entire brace-enclosed sequence. If the option does not appear on the command line, <string2> replaces the entire sequence.

2.3.4. Saving Intermediate Files

Intermediate files are saved by specifying +* flags to the compiler driver. If, on the command line, you specify:

```
C>   c +1 +2 +asm file.p
```

the compiler driver will save the intermediate files file.c, file.1, and file.2, and will stop after creating file.2 (provided there are no fatal errors or other barriers to successful compilation). Note that the suffixes actually created on your system may be different depending on your target operating system.

2.3.5. The Grouping Line

Note: Because the DOS/8086 Cross to System/36 Cross Compiler does not support the Whitesmiths linker or object tools, the prototype file as shipped contains no grouping line or follow-up lines. You may, however, add such lines yourself, as explained in section 2.4, "Modifying Prototype Files".

There is a special type of step that you can add to a prototype file called the "grouping line", which differs visually from other steps in that its label is followed by two colons instead of one. The grouping line is run after all processing of filename arguments to the compiler driver has been successfully completed. It is commonly used to bind the components of a program together from multiple object or source files and is customarily one of the last lines in the file. The grouping line takes the form:

```
o::lnk -o (o) <startup files> (i) <libraries>
```

Another special characteristic of the grouping line is that each file output by the program immediately preceding it is made permanent, rather than being temporary like the files output by prior programs. The permanent file output is given the same name as the input file originally specified by the user, but with its suffix changed to match the label of the grouping line. The driver will save all files with a suffix that matches the label on the grouping line, since these files may be needed later for relinking or other processing. It will then take the names of all the files saved (in this case all files with a .o suffix) and insert them (separated by a host-specific character, typically a space) where the (i) appears in the grouping line of the prototype file. The output file string (o) for the grouping line is derived from the -o option to the compiler driver on the command line.

2.3.6. Follow-up Lines

Follow-up lines are steps which follow the grouping line, and indicate optional transformations that may be performed subsequent to the "binding" performed on the grouping line. Utilities which modify the header of the executable object module to execute under operating systems other than the host, or utilities which produce symbol table maps, build hex files, copy programs to diskette, and so forth are often invoked via follow-up lines in the prototype file. The suffix used is derived from the label on that particular line. Programmable flags are a good way to turn these procedures on and off.

2.4. Modifying Prototype Files

The purpose of this section is to describe the compiler driver and its associated prototype file for users who need to write customized prototype files or install a different standard prototype file. Familiarity with section 2.3, "Introduction to Prototype Files", is assumed.

Compiler Driver/Prototype File Interaction

All Whitesmiths compilers consist of three or more separate programs, which must be run in sequence, and which communicate via intermediate files. The multi-pass compiler driver, c, through an associated prototype file, standardizes the way in which the components of the compiler are run. Its function is to automate the invocation of the multi-pass compiler program by performing the following tasks:

1) naming the programs (such as compiler passes) to be run, and running them in the correct sequence.

2) specifying invariant parameters, such as flags, that are to be passed to a given program every time it is run.

3) associating user-specified parameters, such as optional flags or source filenames, with the programs to be run.

4) passing the output of one program to the next program in the sequence, which must accept it as input.

5) end processing, including naming the final output file.

When c is typed on the command line, the compiler driver is invoked. The driver looks for the appropriate prototype file along whatever search path your system's "path" variable is set to. (The path is a specific list of places in the system's directory structure that are searched through to find executable commands. The path variable is initialized by the system administrator, but can be reset by the user to allow individual customization of the command set.) The driver looks for a prototype file which has a name matching its own followed by a suffix. Normally the c driver looks for a file called c.pro. However, the name of the prototype file to be searched for can be set directly via the -proto* flag to the driver.

Once the prototype file has been opened, the driver creates a specialized internal memory structure to store the information it gets from the prototype file. This structure is in the form of a graph of the processes to be run. It also includes the name of the process, its arguments, the names of associated input and output files, and whether any intermediate files are to be permanent or temporary. It then applies this structure to filenames on the command line, using the filenames as arguments to the processes in the list, and then runs the processes accordingly.

The flow of control is "event-oriented" (implicit within the prototype file) and data-dependent (driven by the input the driver takes from the prototype

file and associated command line arguments). Labels in the prototype file are interpreted by the driver as "entry", "exit", or "drop-off" points for a particular category of files. During normal processing of command line arguments (after the flags) an attempt is made to match each incoming file name with an entry step in the prototype file, and then to put that file through the various steps and jumps designated in the prototype file until the grouping line (or the last line if there is no grouping line) is reached. When all files have reached the grouping line, or when the last line is reached, it is executed, and so are any follow-up lines. c expects to submit each of its input files to at least one prototype command. An input file not matching the label of any of the commands in the prototype file will cause an error.

If there are any fatal errors in any of the programs involved, the driver closes all open temporary files and stops compilation. If one or more command line arguments or filenames are illegal, or if the compiler returns a failure status, the driver moves on to the next filename on the command line and begins processing again from there.

The only error that the driver checks for on its own is illegal command line length. In general, the driver passes along whatever it receives via the prototype file and the command line.

Jumps

The syntax for a "jump" within a prototype file consists of a legal label corresponding to a default file suffix, followed by a colon, alone on a line by itself. It causes the driver to jump down to the next occurrence of that label and resume processing there. If there are no more occurrences, compilation stops and the output file created by the last step performed has that suffix. The following example of a jump statement:

```
asm:
```

means "jump down to the next occurrence of an asm: label".

Multi-line Steps

Multi-line steps are used when more than one round of processing is to take place on a class of files (those with the same suffix). Transformation of the suffix or creation of temporary files can take place only as a result of the action of the step as a whole, and not through any one of its lines taken alone. A given line in a multi-line step does not modify filenames (by appending or changing suffixes) or create new files in and of itself. It exists as part of a series of commands to the driver that perform multiple processes on a given class of files. A one-line step sets in motion only one process. Note: if any single line in a single- or multi-line step is too long, continue it with a backslash \ or the results will be unpredictable. Be careful not to put a backslash at the end of a step, as the driver will attempt to include the next step as part of it.

Compression of Whitespace

Whitespace within lines of a prototype file undergoes "compression" when interpreted by the compiler driver. Tab characters are converted to spaces, which are in turn compressed to a single space. Any and all expanses of whitespace in the prototype file, unless enclosed in double quotes " " or "escaped" on a character-by-character basis with the backslash \ character, are compressed to a single space. The use of quotes and backslash \ characters to preserve the appearance of strings works in much the same way as it does in the C language itself. For example, the string

```
a   b   c
```

would ordinarily be compressed to the form

```
a b c
```

By surrounding the string with double quotes, or "escaping" each character by preceding it with a \, a string can pass through the driver uncompressed. To maintain the appearance of the original string above, it could be inserted in a prototype file as either:

```
"a   b   c"
```

or

```
\a\ \ \ \b\ \ \ \c
```

The \ can also be used to allow insertion of a physical \ into the file. To include an actual backslash character, enter it as \\.

2.4.1. Using Programmable Flags

The programmable flag feature gives users the flexibility to specify their own programmable flags to the compiler passes, to customize flags, perform arbitrary mapping of one flag to another, or adapt control flow within the compilation process based on the presence or absense of flags on the command line.

The sequence:

```
{<flagname>?[<string1>][:<string2>]}
```

comprises a test for the presence of an option on the command line, and the steps to perform based on the results of that test.

The <flagname> string specifies the actual command line option to test for. If the option is present, <string1> replaces the entire brace-enclosed sequence. If the option does not appear on the command line, <string2> replaces the entire sequence. Although the special characters ({ } ?) are mandatory, it is not necessary to specify both character strings. Thus, the legal configurations for programmable flag commands are:

```
{flagname?yes-string:no-string}

{flagname?yes-string}
```

{flagname?:no-string}

Note: while the number of files listed on the command line is restricted only by the operating system's limit on command line length, no more than 20 programmable flag options may be specified at one time.

Creating New Programmable Flags

Programmable flags govern the expansion of brace-enclosed sequences within the prototype file as it is read in. The structure of this sequence as it pertains to any given programmable flag determines what will happen if the flag appears on the command line.

Creating a new programmable flag and making it work is not difficult. The main problems are defining the behavior of the flag and writing a brace-enclosed sequence to carry it out. For example, assume that **xxx** represents a new programmable flag. All that is needed to invoke its behavior (from the user's point of view) is to include **-dxxx** on the command line to the compiler driver.

From the point of view of the prototype file, the addition of a syntactically correct brace-enclosed sequence in the right place is all that is needed. This sequence can take one of three basic forms:

{xxx?abc} - becomes the string abc when the prototype file is read into memory by the driver (where abc is any string that does not contain an unquoted colon).

{xxx?:def} - makes def disappear if -dxxx appears on the command line.

{xxx?abc:def} - expands abc if the flag appears on the command line, and def if it does not.

It is also possible to nest programmable flag expansion sequences. Such a nested sequence can be used to obtain "or" and "and" constructs when testing for combinations of programmable flags. {a?{b?AB:A-}:{b?-B:--}} expands to:

AB if both -da and -db appear on the command line
A- if -da appears but not -db
-B if -db appears but not -da
-- if neither programmable flag appears

2.5. Generating Program Listings

Whitesmiths' c compiler driver is designed to help the user compile source code so as to allow intermediate files, such as assembly or object code, to be saved during compilation instead of being overwritten by the next compiler pass. This capability forms the basis for giving users great flexibility in specifying the contents of their code and error listings.

Listings of the output of any or all of the compiler passes are obtained by specifying one (or several) of a series of built-in programmable flags to the c compiler driver. These flags are:

-dlistcs - list C source and assembly language source (the output of the code generator) together (if the input is not a .c file, you must also give the +c flag to the compiler driver)

-dlistps - list Pascal source and assembly language source together

-dlistpc - list Pascal source and C source (the output of ptc) together

-dlistpcs - list Pascal source, C source, and assembly language source together (if the input is not a .c file, you must also give the +c flag to the driver)

Note: If you do not have a Pascal compiler, only the -dlistcs flag applies.

This group of flags works by passing the **+list** flag to the relevant compiler passes. Any -dlist* flag on the driver command line forces the invocation of lm, the listing merger utility, which merges the listings output by the compiler, and pr, which paginates the listing and provides a header on top of each page.

There are two other standard programmable flags that can be used in conjunction with the above flags to turn on special listing features. These are:

-dlincl - include a copy of the contents of relevant header files with the listing output.

-dlo - redirect the listing output from the default "standard output" to the file root.suffix, where root is the name of the original source file, with its original suffix replaced by the host system's default suffix for listings. (The default on DOS systems is .lst.) This flag is useful for saving listings without explicit redirection of output on the command line.

All passes of the compiler driver accept the same line references and error message flags, which are generated by each pass along with the source code translation. When the driver exits, all files created will have embedded in them relevant error listings from the pass that generated them, as well as listings from all previous passes where diagnostics were output (i.e. files of assembly language code will contain diagnostics generated by the compiler and/or the assembler, depending on if either or both passes found errors).

However, any error message generated by any intermediate step points to the line in the original source code where the error occurred. In other words, the output listing will hold line number references from the originating language. Assembler error messages, for example, will refer to lines in the source code, not the assembly code.

It is also possible, by stopping the compiler at the desired intermediate step, removing the original source line referrents, and compiling again with the intermediate code as the source, to generate error listings that point to lines in the intermediate-language file. For example, to generate a listing of assembly language code with error messages that refer to the assembly language code itself instead of the original C or Pascal source, stop compiling after the code generator pass, thus automatically saving the resulting machine-dependent assembly code in a permanent file. Strip out the original #line sequences, and begin compilation again from there. Assembler will then be the originating language to which error messages point.

The example on the following pages shows the listing produced by compiling the C source file **echo.c** (included with your compiler) with the **-dlistcs** programmable flag option:

```
C> c -dlistcs echo.c
```

echo.asm Page 1

```
 OPTIONS NOXREF
* C36 ASSEMBLER CODE, 86-07-30
MAIN START 0
 BD $TEXT
* TEXT
*    1  /*       ECHO ARGUMENTS TO STDOUT
*    2   *       copyright (c) 1980, 1986 by Whitesmiths, Ltd.
*    3   */
*    4  #include <wslxa.h>
*    5
*    6  /*       flags:
*    7            -m              output newline between arguments
*    8            -n              do not put newline at end of arguments
*    9   */
*   10  BOOL mflag = {NO};
* BSS
*MFLAG EQU *
$DATA EQU *
 DS 2CL1
*   11  BOOL nflag = {NO};
NFLAG EQU *
 DS 2CL1
*   12
*   13  TEXT *_pname = {"echo"};
* DATA
_PNAME EQU *
 DC AL2($5)
$5 EQU *
 DC XL5'8583889600'
$4 EQU *
 DC AL2(NFLAG)
$6 EQU *
 DC AL2(MFLAG)
$01 EQU *
 DC AL2($51)
$51 EQU *
 DC XL13'946B957AC6404C819987A26E00'
$41 EQU *
 DC AL2($52)
$02 EQU *
 DC AL2($53)
$53 EQU *
 DC XL2'4000'
$52 EQU *
 DC XL2'2500'
$22 EQU *
 DC AL2($54)
$54 EQU *
```

echo.asm Page 2

```
 DC XL2'2500'
*   14
*   15  /*      output args separated by space or newline
*   16   */
*   17  BOOL main(ac, av)
*   18          BYTES ac;
*   19          TEXT **av;
*   20          {
* TEXT
*MAIN EQU *
$TEXT EQU *
 LA $03,#XR2
 BD @CENT
$03 EQU *
*   21          IMPORT BOOL mflag, nflag;
*   22          FAST COUNT n, ns, nw;
*   23          TEXT *between;
*   24
*   25          getflags(&ac, &av, "m,n:F <args>", &mflag, &nflag);
 MVC 237(2,#XR1),$4+1
 MVC 235(2,#XR1),$6+1
 MVC 233(2,#XR1),$01+1
 LA 252(,#XR1),#WR4
 ST @AC0+7,#WR4
 MVC 231(2,#XR1),@AC0+7
 LA 250(,#XR1),#WR4
 ST @AC0+7,#WR4
 MVC 229(2,#XR1),@AC0+7
 LA -22,#XR1
 B GETFLA
*   26          if (!ac)
 CLC 251(2,#XR1),@CONST+3
 JNE $1
*   27                  return (YES);
 MVI @AC0+6,0
 MVI @AC0+7,1
 B @CRET
$1 EQU *
*line 28, bytes 99
*   28          between = mflag ? "\n" : " ";
 CLC MFLAG+1(2),@CONST+3
  JE $21
 MVC @AC0+7(2),$41+1
 B $61
$21 EQU *
 MVC @AC0+7(2),$02+1
$61 EQU *
 MVC 239(2,#XR1),@AC0+7
```

echo.asm Page 3

```
*   29          for (nw = 0, ns = 0; ac; --ac, ++av)
 MVI 240(,#XR1),0
 MVI 241(,#XR1),0
 MVI 242(,#XR1),0
 MVI 243(,#XR1),0
$11 EQU *
*line 29, bytes 172
 CLC 251(2,#XR1),@CONST+3
 JE $12
*   30                      {
*   31                      if (nw)
 CLC 241(2,#XR1),@CONST+3
 JE $15
*   32
*   33                              {
 MVI 236(,#XR1),0                   nw += write(STDOUT, between, 1);
 MVI 237(,#XR1),1
 MVC 235(2,#XR1),239(,#XR1)
 MVI 232(,#XR1),0
 MVI 233(,#XR1),1
 LA -18,#XR1
 B WRITE
 ALC 241(2,#XR1),@ACO+7
*   34                              ++ns;
 ALC 243(2,#XR1),@CONST+7
$15 EQU *
*line 36, bytes 254
*   35
*   36                          }
 L 253(,#XR1),#XR2          nw += write(STDOUT, *av, n = lenstr(*av));
 MVC 235(2,#XR1),1(,#XR2)
 LA -16,#XR1
 B LENSTR
 MVC 245(2,#XR1),@ACO+7
 MVC 237(2,#XR1),245(,#XR1)
 L 253(,#XR1),#XR2
 MVC 235(2,#XR1),1(,#XR2)
 MVI 232(,#XR1),0
 MVI 233(,#XR1),1
 LA -18,#XR1
 B WRITE
 ALC 241(2,#XR1),@ACO+7
*   37                      ns += n;
 ALC 243(2,#XR1),245(,#XR1)
*   38                      }
 ALC 251(2,#XR1),@CONST-1
 ALC 253(2,#XR1),@CONST+11
 J $11
```

echo.asm Page 4

```
$12 EQU *
*line 38, bytes 353
*   39          if (!nflag)
 CLC NFLAG+1(2),@CONST+3
 JNE $16
*   40                  {
*   41                  nw += write(STDOUT, "\n", 1);
 MVI 236(,#XR1),0
 MVI 237(,#XR1),1
 MVC 235(2,#XR1),$22+1
 MVI 232(,#XR1),0
 MVI 233(,#XR1),1
 LA -18,#XR1
 B WRITE
 ALC 241(2,#XR1),@ACO+7
*   42                  ++ns;
 ALC 243(2,#XR1),@CONST+7
$16 EQU *
*line 44, bytes 428
*   43                  }
*   44          return (nw == ns);
 CLC 241(2,#XR1),243(,#XR1)
  JNE $42
 MVI @ACO+6,0
 MVI @ACO+7,1
 B $62
$42 EQU *
 MVI @ACO+6,0
 MVI @ACO+7,0
$62 EQU *
 B @CRET
*   45          }
 ENTRY MAIN
 EXTRN LENSTR
 EXTRN WRITE
 ENTRY _PNAME
 ENTRY NFLAG
 ENTRY MFLAG
 EXTRN GETFLA
 EXTRN @CRET
 EXTRN @CENT
MFLAG EQU $DATA
#XR1 EQU X'01'
#XR2 EQU X'02'
#ARR EQU X'08'
#IAR EQU X'10'
#WR4 EQU X'44'
#WR5 EQU X'45'
```

echo.asm Page 5

```
#WR6 EQU X'46'
#WR7 EQU X'47'
 EXTRN @ACO
 EXTRN @FP
 EXTRN @XR2
 EXTRN @CONST
 END
```