SECTION I

Introduction

1. Introduction

The Compiler Users' Manual for DOS/8086 Cross to SSP/36 describes the Whitesmiths, Ltd. program development system as implemented under DOS and generating code for the IBM System/36 running SSP. It is divided into three sections. Section I discusses this manual and its use. Section II is a series of tutorials that describe the architecture of the program development system, how to invoke the compiler and begin using it, how to generate listings, and how to customize the compilation process to meet your specific needs. Section III is designed for use as a reference tool. It describes the details of the compiler passes and other utilities in "manual page" format.

1.1. Documentation Overview

Your Compiler Users' Manual discusses Whitesmiths' program development system relative to your host environment (the machine and operating system combination under which your code is to be compiled). The Interface Manual included with your package describes those aspects of the system that are specific to your target environment (the hardware on which your code is to be run, including operating system specific information as well as standalone operation).

1.1.1. How to Use This Manual

Users new to a Whitesmiths environment should first read the remainder of Section I, which discusses our documentation conventions. All users should read section 2.1, "Introduction to Whitesmiths' Program Development System", which explains the interrelationships among the compiler, tools, and host operating system. To get the most from the tutorials in Section II, read them all the way through, typing in and executing examples as you go. If you need information on a given topic area, use the Table of Contents or the Index to locate subsections of interest.

The manual pages in section III are written and organized to facilitate quick referencing of specifics. Each manual page includes a synopsis of how the program is invoked, a description of how it works, what flags it accepts, an explanation of what values it returns and what its output looks like, an example of its use, and pointers to related files or programs. These sections are organized alphabetically by program name. The more experienced you become with Whitesmiths' program development environment, the more useful you will find them.

Cross-referencing within and between manuals is provided wherever applicable. An index and a detailed table of contents is also provided with each manual to help you locate specifics within the text.

1.2. Notational Conventions

This section describes the notational conventions that appear throughout this manual.

1.2.1. Boldfaced Print

Boldface print is used to specify the following:

1) utility names

2) function names

3) flags to utilities

4) commands used with interactive utilities

5) C keywords or reserved words

6) formulas

7) variables and optional or user-defined command line elements described in "metanotion" form (i.e. enclosed in angle brackets < >, square brackets [ ], or both). The meanings of these notational conventions are discussed in subsection 1.2.4, "Notational Shorthand".

8) example filenames and values for variables, etc. that are referred to in explanations

9) specific values or characters used in commands or code, or referred to in explanations.

10) anything that the user types

1.2.2. Specifying Ranges

[ ] Square brackets are used to specify the range of a particular element (referred to as a "closed" interval). For example, the range specification for the closed interval [2 6] means that the element can be any number between 2 and 6, including both 2 and 6).

[ ) The parenthesis in this notation (referred to as a "half-open" interval) indicates that whatever value appears on the right-hand side of the set (or the left-hand side if that's where it appears) is not part of the valid set of values. For example, [3 7) means that the valid range of the element in question can be any number between 3 and 7, including 3 but not 7. Alternatively, valid numbers in the range (1 5] would be 2, 3, 4, and 5.

( ) Parentheses are used to specify the range of a particular element (referred to as an "open" interval). Given the open interval (0,5), therefore, the valid range of numbers is "1, 2, 3, 4".

1.2.3. Capital Letters

CAPITAL LETTERS are used for names which must be entered in all capital letters, or that appear as such in source code, header files, and so forth. Capitalization is not used simply for extra emphasis.

1.2.4. Notational Shorthand

Grammar, the rules by which syntactic elements are put together, is important to the remaining discussions. Some simple shorthand is used throughout this manual to describe grammatical constructs.

A name enclosed in angle brackets, such as <statement>, is a "metanotion", i.e., some grammatical element defined elsewhere. Any sequence of tokens that meets the grammatical rules for that metanotion can be used in its place, subject to any semantic limitations explicitly stated. Just about any other symbol stands for itself, i.e. it must appear literally, like the semicolon in

```
<statement> ; <statement>
```

Exceptions are the punctuation [,],]*, {, }, and |; these have special meanings unless made literal by being enclosed in single quotes.

Brackets surround an element that may occur zero or one time. The optional occurrence of a label, for instance, is specified by:

```
[ <label> : ] <statement>
```

This means that metanotion <label> may (but need not) appear before <statement>, and, if it does, it must be followed by a literal colon. To specify the optional, arbitrary repetition of an element, the notation []* is used. A comma-separated list of <ident> metanotions, for example (i.e., one instance of <ident> followed by zero or more repetitions), would be represented by:

```
<ident> [, <ident> ]*
```

Vertical bars are used to separate the elements in a list of alternatives, exactly one of which must be selected. The line:

```
char | int | long | short
```

requires the specification of any one of the four keywords listed.

1.2.5. Introduction to Using the Utilities

Each of the utilities described in this manual is a separate "program" that may be run, or "invoked", by typing a "command line", usually in response to a "prompt" such as the DOS "C> ". This section provides a systematic guide to the conventions that govern how command lines are specified. Section 1.2.6,

entitled "Manual Page Conventions", summarizes the layout of the individual utility descriptions that follow, so that you know what information appears where, and in what format.

Command Lines

In general, a command line has three major parts: a program name, an optional series of flags, and a series of non-flag arguments, usually given in that order. Each element of a command line is most often a string, separated by whitespace from all the others. Most of the time, the program name is just the (file) name of the utility you want to run. Flags, if given, change some aspect of what a utility does, and generally precede all other arguments, because a utility must interpret them before it can safely interpret the remainder of the command line.

The meaning of the non-flag arguments strongly depends on the utility being run, but there are three general classes of command lines, presented here (where possible) with examples taken from the portable software tools, since they run much the same way on many different systems:

1) program name and flags followed by a series of filenames. These programs (called filters) process each file named in the order specified. pr is an example.

2) program name and flags followed by a series of arguments that are not filenames, but strings to which the program gives some other interpretation. The echo program included with the compiler package is one such utility.

3) program name and flags, followed by a mandatory argument, followed by other arguments and the rest of the command line. lib belongs to this class. Note that lib, a Whitesmiths utility used to build and maintain subroutine libraries, is not shipped with the DOS/8086 to SSP/36 cross compiler.

A summary of the command line classes looks like this:

Class	Example	Syntax
filter	pr	<progname> <flags> <files>
string arguments	echo	<progname> <flags> <args>
mandatory argument	lib	<progname> <flags> <arg> <files>

Note that, in general, <flags> are optional in any command line.

Flags

Flags are used to select options or specify parameters when running a program. They are command line arguments recognized by their first character, which is always a - or a +. The name of the flag (usually a single letter) immediately follows the leading - or +. Some flags are simply YES/NO indicators (either they're named or they're not), but others must be followed by some additional

information, or "value". This value, if required, may be an integer, a string of characters, or just one character. String or integer values are specified as the remainder of the argument that names the flag; they may also be preceded by whitespace, and therefore be given as the next argument.

The flags given to a utility may appear in any order, and two or more may be combined in the same argument, so long as the second flag can't be mistaken for a value that goes with the first one. Some flags have only a value, and no name. These are given as a - or + immediately followed by the value.

Thus, all of the following command lines are equivalent, and would pass to c the flags -n and -s:

```
C> c -n -s
C> c -s -n
C> c -sn
```

And each of the following command lines would pass the three flags -c3, -n, and -4 to pr:

```
C> pr -c3 -4 -n file1
C> pr -4 -nc 3 file1
C> pr -n4 -c 3 file1
```

In short, if you specify flags so that you can understand them, a utility should have no trouble, either.

Usually, if you give the same flag more than once, only the last occurrence of the flag is remembered, and the repetition is accepted without comment. Sometimes, however, a flag is explicitly permitted to occur multiple times, and every occurrence is remembered. Such flags are said to be "stacked," and are used to specify a set of program parameters, instead of just one.

Another special flag is the string "--", which is taken as a flag terminator. Once it is encountered, no further arguments are interpreted as flags. Thus, a string that would normally be read as a flag, because it begins with a - or a +, may be passed to a utility as an ordinary argument by preceding it with the argument "--". The string "-" also causes flag processing to terminate wherever it is encountered, but, unlike "--", is passed to the utility instead of being "used up", for reasons explained below.

If you give an unknown flag to a utility, it will usually display a hint to remind you of what the proper flags are. This message summarizes the format of the command line the utility expects, and is explained below in the SYNOPSIS section of the pseudo-manual page. Should you forget what flags a utility accepts, you can force it to output this "usage summary" by giving it the flag -help, which is never a valid flag argument.

Finally, be warned that some combinations of flags to a given utility may be invalid. If a utility doesn't like the set you've given, it will output a message to remind you of what the legal combinations are.

Files

Any utility that accepts a series of input filenames on its command line will also read its standard input, STDIN, when no input filenames are given, or when the special filename "-" is encountered. pr can therefore be made to read STDIN just by typing:

```
C> pr
```

while the following would process file1, then STDIN, then file2, and write them to STDOUT:

```
C> pr file1 - file2
```

Whenever STDIN is read, it is read until end-of-file, and so should not be given twice to the same program.

1.2.6. Manual Page Conventions

This subsection deals with the format of the manual pages describing each of the utilities. Manual pages are terse, but complete and tightly organized. They are designed for quick reference, and usually do not offer much tutorial help. Manual pages are divided into several standard sections, each of which covers one aspect of using the documented utility. The rest of this essay is presented as a pseudo-manual page, with the remarks on each section of a real page appearing under the heading for that section.

program name

NAME

program name - the name and a concise description of the utility

SYNOPSIS

This section gives a one-line synopsis of the command line that the utility expects. The synopsis indicates the main components of the command line: the utility name itself, the flags the utility accepts, and any other arguments that may (or must) appear. A utility will output a synopsis of its usage online if it is passed an invalid flag or argument, or if an expected parameter is left off. You can also generate this online message intentionally by typing the utility name followed by the flag -help.

Flags are listed by name inside the delimiters -[and]. They generally appear in alphabetical order. Flags consisting only of a value are listed after all the others. If a flag includes a value, the kind of value it includes is also indicated by one of the following codes, which are given immediately after the flag name:

Code	Kind of Value
*	string of characters
#	integer (word-sized)
##	integer (long)
?	single character

A # designates an integer representable in the word size of the host computer, which may limit it to a maximum as small as 32,767 on some machines. A ## always designates a long (four-byte) integer, which can represent numbers over two billion. An integer is interpreted as hexadecimal if it begins with 0x or 0X, as octal if it begins with 0, and as decimal otherwise. A leading + or - sign is always permitted.

If a flag may meaningfully be given more than once (and stacks its values), then the value code is followed by a caret ^.

For example, the synopsis of the **pr** utility:

pr -[attn* c# err f# h it# l# m n ot# p s? t* w# +## ##] <files>

indicates that pr accepts sixteen distinct flags, of which -c, -f, -it, -l, -ot, and -w include word-sized integer values, -err, -h, -m, -n, and -p include no values at all, -attn and -t include a string of characters, -s includes a single character, and the two flags +## and -## are nameless, consisting only of a long integer.

Note that flags introduced by a hyphen - (which is assumed unless otherwise specified) are shown without it. In the pr manual page, for example, -c# would refer to the flag listed above as c#, while -[c# f# w#] would refer to the flags -c#, -f#, and -w#.

The position and meaning of non-flag arguments are indicated by "metanotions", that is, words enclosed by < and > (like <files> in the example above). Each metanotion represents zero or more arguments to be given on

the command line. When entering a command line, you type whatever the metanotion represents, and type it at that point in the line. In the case of pr, you would enter one or more filenames at the point indicated by **<files>**. A metanotion enclosed in square brackets is optional, and may appear zero or more times.

FUNCTION

The **FUNCTION** section generally contains three parts: an overview of the operation of the utility, a description of each of its flags, and (if necessary) additional information on how various flags or other arguments interact, or on details of the utility's operation that affect how it is used.

Usually, the opening overview is brief, summarizing what the utility does and how it uses its non-flag arguments. Flag descriptions follow, consisting of a separate sentence or more of information on each flag, introduced by the same flag name and value code given under the SYNOPSIS heading. Each description states the effect the flag has if given, and the use of its value, if it includes one. The parameters specified by flag values generally have default settings that are used when the flag is not given; such default values appear at the end of the description. Flags are listed in the same order as in the synopsis line.

Finally, one or more paragraphs may follow the flag descriptions, which explain what combinations of flags are not permitted, or how certain flags influence others. Any further information on the utility of interest to the general user is also given here.

RETURNS

When it finishes execution, every utility returns one of two values, called "success" or "failure". This section states the conditions under which a utility will return one rather than the other. A successful return generally indicates that the utility was able to perform all necessary file processing, as well as all other utility-specific operations. In any case, the returned value is guaranteed to be predictable.

Note that the returned value is often of no interest -- it can't even be tested on some systems. But when it can be tested, it is instrumental in controlling command scripts.

EXAMPLE

Given here are one or more examples of how the utility can be used. The examples seek to show the use of the utility in realistic applications, in conjunction with related programs.

Generally, each example is surrounded by explanatory text, and is set off from the text by an empty line and indentation. In each example, boldfaced lines preceded by a prompt (such as C>) represent user input; other lines may be the utility's response to the input shown.

FILES

This section lists any external files that the utility requires for correct operation. Most often, these files are system-wide tables of information or common device names.

SEE ALSO

Listed here are the names of related utilities or tutorials, which should be examined if the current utility doesn't do quite what you want, or if its operation remains unclear to you. Another utility may come closer, and seeing the same issues addressed in different terms may aid your understanding of what's going on. Other documents in the same manual section as the current page are referred to simply by title; documents in a different section of the same manual are referred to by both title and section number.

NOTES

This section documents inconsistencies or shortcomings in the behavior of the utility. Most often, these consist of deviations from expected conventions. Known dangers inherent in the improper use of a utility, or its behavior in combination with other factors, will also be pointed out in this section.