SECTION I

Introduction

1. Introduction

The Interface Manual for IBM System/36 describes the machine-dependent aspects of the Whitesmiths, Ltd. program development system for the IBM System/36. It is divided into four sections. Section I discusses this manual and its use. Section II consists of a series of detailed explanations describing: the System/36 assembler and the compiler/assembler interface, implementation-specific modifiers, the role of the linker in creating memory layout, and the error messages produced by the System/36 code generator. Section III.a describes the SSP system interface library functions in "manual page" format. Section IV describes the System/36 machine interface library functions that are called by code produced by the code generator as well as C-callable routines in "manual page" format.

1.1. How to Use This Manual

Users new to Whitesmiths environments should first read the remainder of Section I, which discusses our documentation conventions and the effective use of the "manual pages" used to document individual functions (and, in other manuals, utilities as well). To get the most from the detailed explanations of the topics in Section II, read each subsection all the way through first, and then go back and work through the specific topic areas of interest to you. Whitesmiths has endeavored to anticipate the information needs of our users, but a system on the level of complexity and flexibility of a compiler cannot be described so as to benefit all users equally.

The manual pages in sections III.a and IV are written and organized to facilitate quick referencing of specifics. Each manual page includes a synopsis of the return type and argument list of the function, a description of how it works, what arguments it takes, an example of its use, and pointers to related files or programs. The more experienced you become with the Whitesmiths program development environment, the more useful you will find the manual page format.

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, <u>including</u> 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 <u>not</u> 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 <u>between</u> 3 and 7, including 3 but <u>not</u> 7. Alternatively, valid numbers in the range (1, 5] would be 2, 3, 4, and 5.

( ) Parenthesis 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, and 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. Manual Page Conventions

This subsection deals with the format of the manual pages used to describe individual functions. 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 the documented function. The rest of this writeup is presented as a psuedo-manual page, with the remarks pertinent to each section of an actual page appearing under the heading for that section.

NAME

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

SYNOPSIS

The returned type and argument list of the function are given here, precisely as they would appear in a C program defining the function. Almost always, the types of arguments and function come from the set of psuedo-types defined by the standard headers. These psuedo-types are for the most part simple equivalents to types pre-defined by C, renamed to increase their mnemonic value, to isolate machine dependencies, to promote disciplined coding practices, or (usually) to achieve some combination of all three.

FUNCTION

Generally, this section contains three parts. The opening sentences state what the function does. Next comes a summary of each argument to the function, including its general effect on the function's operation. Each argument is called by the same name as was given in the SYNOPSIS. These names are usually mnemonic, to make citations distinct but still self-defining. Finally, an additional paragraph (or more) may provide details of how the routine works, or how specific argument values affect it.

Note that heavy use is made of the conventional symbol NULL to refer to a zero-valued pointer, and of NUL to refer to the ASCII code zero, or "\0". Other symbols defined in the standard header may also appear from time to time.

RETURNS

This section describes the range of possible return values for the function, and under what circumstances one value will be returned rather than another. Any location in the calling program altered via pointer access from the function is also documented here, although it may have already been mentioned under FUNCTION.

EXAMPLE

Here are shown one or more typical calls to the documented function, and (sometimes) their results. The examples are designed to be brief and evocative, or even useful as code fragments directly interpolated into user programs. Often, related functions have been given similar examples, either to emphasize the differences in usage between routines that perform similar tasks, or to show conventional patterns of use that apply across a family of routines.

A line consisting of just an ellipsis "..." is used to indicate the omission of some (incidental) code between the line preceding and the line following.

SEE ALSO

This section lists related functions that could be profitably examined in conjunction with the current one. Functions documented in the same manual section are simply listed by name; functions documented elsewhere in the same manual are listed followed by the number of the section containing them.

If the current function does not do quite what you want, or if its operation seems unclear to you, examine the related functions. Another function may come closer, and seeing the same issues addressed in different terms may aid your understanding of what's going on.

NOTES

Special behavior and/or known inconsistencies in the documented routine are mentioned here. Most often, these relate to insufficient checking of value sensitive user-supplied parameters, dangers inherent in the improper use of a function, or its behavior in combination with other factors. Also, notice is always given here if a "function" is in reality a macro.