User's Guide for the Georgia Tech C Compiler

Second Edition

Daniel H. Forsyth, Jr.
Edward J. Hunt
Jeanette T. Myers
Arnold D. Robbins

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

October, 1984

TABLE OF CONTENTS

Getting Started

Features of Georgia Tech C

Compile Time Facilities

Run Time Environment

Conversion

Bugs

Technical Information

Subsystem Managers Section

Foreword

The Georgia Tech C compiler and run time support library provide a C programming environment on Prime computer systems. The Georgia Tech C Compiler runs under and **requires** the Georgia Tech Software Tools Subsystem, Version 9 or later. Both run on PRIME 400, 500 and 50-series computers.

This guide documents the second version of the Georgia Tech C compiler and run time library which is released with Version 9 of the Software Tools Subsystem. The eight chapters of this guide

1) explain the use of the compiler,
2) describe the machine-dependent features of the implementation,
3) describe the compile time environment provided,
4) detail the behavior of the run time package,
5) enumerate problems of conversion from other systems,
6) document known compiler bugs and shortcomings,
7) provide some technical information on the implementation and performane of the C compiler, and
8) outline actions necessary to manage the C system.

A complete description of C can be found in The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). Further information on individual commands in the C system can be obtained from the Software Tools Subsystem Reference Manual, accessible both on paper and through the Subsystem 'help' command. The C run time library is only documented here. There are no 'help' entries for the individual subroutines.

Wherever a routine or facility has been changed from the first release of the C compiler, it will be explicitly noted as such.

It is to be noted that wherever it appears in this document, the term "Unix" is a trademark of AT&T Bell Laboratories, Inc.

Getting Started

Prerequisites

We assume that you are already familiar with the Subsystem; that you can create, delete, edit and list files; redirect input and output; obtain on-line documentation, etc. We also assume that you are familiar with the C programming language. If you are not, you should examine the Software Tools Subsystem User's Guide and The C Programming Language by Kernighan and Ritchie before continuing in this guide.

Throughout this guide, we boldface user input in our examples, as is the convention in the Software Tools Subsystem User's Guide.

Calling the C Compiler

There are several commands that call the C compiler:

```
cc      - compile a C program
ccl     - compile and load a C program
ucc     - "Unix-like" C compile and load
compile - general purpose compiler interlude
```

We follow with a brief description of each. For detailed information and examples refer to the Reference Manual entries for each command, e.g.:

```
] help cc
```

Cc --- compile a C program

'Cc' behaves much like the other Subsystem compiler interfaces. It is a program that takes a file whose name ends in ".c" and calls the programs necessary to convert it into a relocatable object program in a file whose name ends in ".b". 'Cc' calls two major programs: the compiler front end 'c1', and the code generator 'vcg'. If you have no compile-time errors in your program, you will not see any messages at all from either program. 'Cc' automatically "includes" the file "=cdefs=" (which is "=incl=/swt_def.c.i") containing macros and external data declarations for the C Standard I/O Library and for interfacing with the Subsystem.

Ccl --- compile and load a C program

'Ccl' compiles a ".c" file in the same way as 'cc'; it then calls 'ld', the Subsystem loader interface, to produce an executable program in a file with no suffix. Unfortunately, the Prime loader is somewhat noisy, so you receive a good bit of output during the execution of 'ld'.

Ucc --- compile and load a C program

'Ucc' is a "Unix-style" C compiler and loader. It is **not**, however, exactly like Unix's 'cc' or any other known Unix program! 'Ucc' recognizes file naming conventions for Subsystem supported languages and will use the appropriate preprocessor and/or compiler to process non-C files. Consequently, it can be used to compile and load several files of different languages into an executable program, as long as the main program is written in C. 'Ucc' now depends on the new 'compile' program to do most of its work. It is just smart enough to arrange to call 'compile' properly; it no longer knows about all the details of the C compiler, or how to go about compiling other languages.

Compile --- general purpose compile and load

'Compile' is a general purpose compiler interlude. It knows about the Subsystem naming convention for the more popular languages available under SWT and Primos. It will arrange to call the proper compiler for each source file, based on the suffix. You may tell it to pass options on to each different compiler or preprocessor, and also to tell it what is the "main" language, in order for it to load any necessary libraries and/or start-off routines. 'Ucc' now just rearranges its arguments, and calls 'compile'.

C Program Development --- An Example

For this example, the file "inout.c" contains the following C program:

```
main ()         /* copy input to output until EOF */
{
        int c;

        while ((c = getchar()) != EOF)
                putchar (c);
}
```

We can compile and load (i.e., link-edit) "inout" with the command

```
] ccl inout.c
```

Consistent with Subsystem convention, 'ccl' places the executable version of "inout.c" in a file named "inout". You can execute "inout" as follows:

```
] inout
a
a
echo me if you dare
echo me if you dare
<control-c>
]
```

Features of Georgia Tech C

Standard Implemented

The Georgia Tech C compiler is based on the specifications contained in The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.

Additional Features

The Georgia Tech C compiler provides the following extensions to C:

1. Unions may be initialized. The first type entry in the union will be used to determine the format of the data. For example, "union {int a; double b;} x = 1;" would initialize "x" as an int, not a double.

2. Except for external names, all characters in all names are significant. External names are up to 8 characters in length, with no case significance. To allow access to Primos system calls, the dollar sign ("$") is also a legal character in identifiers. The external names in the object code produced by the compiler can be up to 32 characters long; it is the SEG loader that restricts their lengths to 8 characters. The 'bind' EPF loader does pay attention to the full 32 character names.

3. The late Unix Version 7 enhancements, structure assignment and "enum" types, are implemented (but not thoroughly tested).

4. C functions can call Fortran, PL/1, etc. routines, and vice versa. C uses the same calling sequence as all other Prime supported languages. SHORTCALL procedure calls (using the JSXB instruction) are not supported by Georgia Tech C.

5. The Ratfor/Algol68 radix notation may be used to specify integer constants. In addition to using a leading 0 for specifying octal and 0x for hexadecimal, Georgia Tech C recognizes the Ratfor radix syntax for integer constants up to base 36. (For instance, "7r123" is 123 base 7, i.e. 66.)

6. Single quotes may be used to specify packed character strings as in Fortran. The Georgia Tech C compiler treats a single character enclosed in apostrophes as a character constant, while more than one character enclosed in apostrophes is considered a pointer to an array of integers containing a packed "hollerith" character constant.

7. The data type "long unsigned" is supported, giving access to 32-bit unsigned numbers.

8. Initialization of automatic aggregates is supported. (The code generator is not particularly smart about it, though, so initializing huge automatic arrays is incredibly space-inefficient.)

9. Macro definitions ("#define"s) can be specified on the command line using the "-D" compiler option.

10. Directories to be searched for include files may be specified on the command line using the "-I" compiler option.

11. The special macros "__FILE__" and "__LINE__" are supported to provide access to the source file name and source line number (as a string constant and an integer constant), respectively.

Compile Time Facilities

Include File Organization

The C compiler package comes with several standard header files which perform a number of functions for the C programmer. All include files are kept in the directory =incl=.

To maintain compatibility with the previous release of the C compiler, the file "=cdefs=" which the C compiler automatically includes (unless you use the "-f" option) is still "=incl=/swt_def.c.i". This is also in accordance with the Subsystem naming convention for other "standard" include files. However, all other C include files end in ".h" (for "header"), which is the Unix convention (this should make porting code a little easier as well).

The "=cdefs=" file itself has been considerably reorganized for the second release. This organization is discussed below.

=incl=/swt_def.c.i

This file now contains very few actual definitions. Instead, it **#include**s separate files to provide the same functionality as it previously did.

We have reorganized the include files to both increase the available functionality, and to separate the features into appropriate, self-contained, "modules". So that previous programs which depend on "=cdefs=" to contain everything need from breaking, "=cdefs=" includes the files it needs. All the include files have been organized so that they may be included more than once. The definitions will only take effect the first time.

The following identifiers are defined in "=cdefs=" for use in determining what kind of hardware and software environment the program will run in. This is useful for writing code designed to be ported to more than one computing system. The identifiers are:

```
#define gtswt   1       /* using Software Tools */
#define primos  1       /* os is primos */
#define prime   1       /* hardware is prime */
#define pr1me   1       /* another name for prime */
```

We have **#define**d the identifier "gtswt" instead of just plain "swt", since "swt" is the name of the routine you have to call in order to exit to the Subsystem.

You would use these identifiers for tailoring your code to different environments. For instance

```
#ifdef gtswt
        /* code to do nifty stuff for Software Tools */
#else
#ifdef unix
        /* code to do nifty stuff for Unix */
#else
        /* code to do nifty stuff in a generic environment */
#endif
#endif
```

"=cdefs=" then includes the following four files (discussed below).

```
#include "=incl=/stdio.h"
#include "=incl=/ctype.h"
#include "=incl=/swt.h"
#include "=incl=/ascii.h"
```

Following the discussion of these four main files, we briefly discribe the other include files which are available.

=incl=/stdio.h

This file contains the declarations and definitions needed to use the C Standard I/O Library.

The following functions, macros, and symbolic constants are available for the programmer to use. (There are others, whose names begin with '_', which the programmer should not need to use, so they aren't discussed here).

```
typedef ... FILE;   The standard type FILE.

/* declaration of functions */
extern long ftell();
extern FILE *fopen(), *fdopen(),  *freopen(), *popen();
extern char *fgets(), *gets();
extern char *strcat(), *strcpy(), *strncpy(), *strpbrk(), *strtok();
extern char *strchr(), *strrchr(), *index(), *rindex();

stdin         Standard Input.
stdout        Standard Output.
stderr        Standard Error Output (Standard Output 3).
stdin1        Subsystem name for stdin.
stdout1       Subsystem name for stdout.
stdin2        Standard Input Port 2.
stdout2       Standard Output Port 2.
stdin3        Standard Input Port 3.
stdout3       Standard Output Port 3 (Error Output).
tty           File pointer which is always the terminal.
errin         Another name for stdin3.
```

errout	Subsystem name for stderr.
EOF	End of file indicator (not a valid character).
NULL	The empty pointer.
BUFSIZ	A convenient buffer size (used a lot on Unix).
TRUE	Represents logical true.
FALSE	Represents logical false.
L_cuserid	The length of a string to hold a user id.
L_ctermid	The length of a string to hold a terminal name.
L_tmpnam	The length of a string to hold a temporary file name.
P_tmpdir	The (string) name of a directory for temporary files.
getchar()	Get a character from stdin.
putchar(ch)	Put a character on stdout.
feof(fp)	Did EOF happen on this file?
ferror(fp)	Did an error occur on this file?
clearerr(fp)	Clear all error flags for this file.
fileno(fp)	Give the SWT file descriptor for the FILE pointer.

Any functions which don't return **int**, and which are not declared in "=incl=/stdio.h", should be declared before they're used. (The type of each function in the C library is given in the chapter on the run time environment, below.)

=incl=/ctype.h

This file defines the character testing macros discussed in the chapter on the run time environment. These macros are very useful, and are often faster than writing an explicit test (e.g. islower(c) should be faster than (c >= 'a' && c <= 'z')). The macros have been rewritten to only evaluate their argument once, so that they won't bite you if the argument has side effects (.e.g. islower(*c++)).

=incl=/swt.h

This file provides most of the functionality that the Ratfor programmer obtains from "=incl=/swt_def.r.i". Some of the Ratfor specific declarations have been deleted (for example, the "dynamic memory" routines). The programmer is referred to Appendix D of the User's Guide for the Ratfor Preprocessor and the "swt.h" file itself for details.

One thing that may need clarifying: The SET_OF_?* macros are used in the following way:

```
        .
        .
    switch (c = getchar()) {
    case SET_OF_UPPER_CASE:
            /* stuff for upper case */
            break;

    case SET_OF_LOWER_CASE:
            /* stuff for lower case */
            break;

    case SET_OF_DIGITS:
            /* stuff for digits */
            break;

    default:
            /* stuff for default */
            break;
    }
        .
        .
```

In other words, you supply the leading **case** and the trailing colon; the macro supplies everything else.

=incl=/ascii.h

This file contains definitions for the ASCII mnemonics, as well as for the control characters. E.g. Both BEL and CTRL_G are defined as octal 0207. The synonyms BACKSPACE, TAB, BELL, RHT, and RUBOUT for other characters are also defined.

=incl=/assert.h

This file defines the 'assert' macro. It must be specifically included in order to use it. See the chapter on the run time environment for what the 'assert' macro does.

=incl=/debug.h

This file declares a macro "debug" which is useful for putting debugging code into your programs. For instance:

```
    #include <debug.h>
    ...
    debug (fprintf (stderr, "i == %d\n", i));
    /* note the balanced parentheses */
    ...
```

If the symbol "DEBUG" has been defined before <debug.h> is included, then whatever occurs as an argument to the "debug" macro will be placed into the source code. Otherwise, "debug" becomes a null macro. The easiest way to turn debugging on is to

put "debug" statements in your code, and then do a "-DDEBUG" on the compiler command line. For larger blocks of code, you can do

```
#ifdef DEBUG
        /*
         * a lot of debugging code
         */
#endif
```

=incl=/keys.h

This file declares the symbolic keys for the Primos file system. It is the analogue of "=incl=/keys.r.i".

=incl=/lib_def.h

This file is analogous to the Ratfor include file "=incl=/lib_def.r.i". It contains symbolic constants and macros which are useful for dealing with the low level Software Tools Library routines.

=incl=/math.h

This file contains declarations for all the mathematical routines in the C library. These routines all return **double**.

=incl=/memory.h

This file contains declarations of mem?* functions. These functions are similar to the str?* functions, but work on arbitrary areas of memory, and do not care about the zero word, '\0'. This file should be included before using the functions, although you can always just declare each function before using it.

=incl=/setjmp.h

This file must be included if you intend to use the 'setjmp' and 'longjmp' non-local goto functions.

=incl=/swt_com.h

This file contains the necessary **#define**s and declarations for accessing the Software Tools common blocks from a C program. It has not been extensively tested. See the file for more details.

=incl=/varargs.h

This file contains definitions which allow you to portably write functions which expect a variable number of arguments. The macros are discussed below, in the chapter dealing with the run time environment. They have not been extensively tested, but do seem to work.

Loading C Programs For Bare Primos

Several of the routines in the C Library depend on the shared Subsystem libraries to do some of their work.

In order for you to write C programs to run under bare Primos, we have provided a second run time library with alternate versions of these few subroutines, as well as a second C start off routine. Most routines perform the same under both the Subsystem and bare Primos. Those few which behave differently under bare Primos are detailed in the chapter on run time facilities. In particular, they always return the value that indicates an error has occurred.

The alternate C start off routine and run time library are in the files =lib=/nc$main and =lib=/nciolib, respectively. Since loading programs for bare Primos is not simple, 'ld' does not have an option for loading C programs for running without the Subsystem. You must do it yourself, by hand.

To load a C program for use with bare Primos, follow this procedure:

1) Load the file =lib=/nc$main. This is the alternate startoff routine, which does some extra initialization.

2) Load your C binaries.

3) Load =lib=/nciolib. This library contains versions of the few routines which act differently under bare Primos. This library also contains a special version of 'getarg', to allow 'argc' and 'argv' to work properly. The environment pointer, 'envp' (see below), will be set to NULL when a program is loaded for running under bare Primos.

4) Load =lib=/ciolib. This contains the rest of the C run time library.

5) Load =lib=/vswtmath, if your C program uses the C math routines.

6) Load =lib=/nvswtlib. This is the non-shared version of the Subystem library, which does most of the real work.

7) Load any system libraries, e.g. the Fortran library.

You should actually be able to use 'ld' to load your program, following this outline:

```
] ld -dnu -l nc$main <binaries> -l nciolib -l ciolib _
   [-l vswtmath] -l nvswtlb -t -m -o <executable_file>
```

You will probably not have too many programs to be run under bare Primos, but we have provided for this possibility.

Run Time Environment

Calling Primos and Subsystem Routines

C programs have access to all Primos system and library subroutines and Software Tools library routines. Georgia Tech C follows Prime's established conventions for parameter passing, thus allowing C routines to call or be called by programs written in other high-level languages or in assembly language. For example the following C program uses the Ratfor subroutine 'putch' for output:

```
main ()      /* copy input to output until EOF */
{
        int c;

        while ((c = getchar()) != EOF)
                putch (c, STDOUT);
}
```

'Ccl' and 'ucc' both use the Subsystem loader interface 'ld'. When loading C programs, 'ld' automatically includes the C Standard I/O Library, "ciolib", the SWT math library, "vswtmath", the shared shell library, "vshlib", and the shared Subsystem I/O and utility library "vswtlb". However, if another library is required, e.g. one of your own making, "mylib", then the following command must be used:

```
] ccl <program_name> -l mylib
```

The Main Program

All complete C programs must have a function named 'main', which is where execution will begin. The 'main' function in Georgia Tech C Programs may have zero, one, two, or three arguments. If there are arguments, the first is an integer, which indicates the number of command line arguments there were (including the command name). The second is a pointer to an array of character strings containing the text of the arguments. The final element in the array will be equal to NULL. The third argument is a similar pointer to an array of character strings containing a list of name=value pairs. These are your shell variables and their values. (This is just like the Unix environment pointer, although shell variables aren't as heavily used under Software Tools.) Try this sample program (call it junk.c):

```
main (argc, argv, envp)
int argc;               /* argument count */
char **argv;            /* argument values */
char **envp;            /* environment pointer */
{
        int i;

        for (i = 0; i < argc; i++)
                printf ("%s\n", argv[i]);

        for (i = 0; envp[i] != NULL; i++)
                printf ("%s\n", envp[i]);
}
```

Compile and run it with:

```
] ccl junk.c
] junk foo bar baz
```

You should see something like:

```
junk
foo
bar
baz
HOME=/uc/arnold
_prt_dest=LPB
_search_rule=^int,^var,&,=ubin=/&,=lbin=/&,=bin=/&
```

The program printed its arguments, and then the names and values of any shell variables you may have set.

C Run Time Library

The Georgia Tech C Run Time Library, "ciolib", is a version of the C Standard I/O library. It is automatically loaded with C programs by 'ccl' and 'ucc'. This section describes the routines available in "ciolib".

We have attempted to provide almost all the routines in Section 3 of the UNIX User's Manual, for Release 1 of UNIX System V. In particular, "ciolib" contains all of the routines marked "3S" (the Standard I/O Library), most of the routines marked "3M" (the Math library), as many as possible of the routines marked "3C" (routines automatically loaded with every C compilation), and even some of the routines marked "3X" (routines from specialized libraries). In addition, there are routines to emulate some of the more useful (and easy to implement) Unix system calls. These should help when porting programs originally written to run under Unix. Finally, there are a few routines which are not provided under Unix at all, but which allow access to certain features of Primos, or which are just generally useful.

NOTE: The calling sequences of two routines, 'c$ctov' and 'c$vtoc', have changed since the first release of the C compiler. The original motivation for these routines was that the C end-of-string character ('\0') was different from the Subsystem EOS. Since they are now the same, these routines have been brought closer in line with the behavior of the other C string routines. If you need them the old way, take a look at 'ctov' and 'vtoc' in section 2 of the Software Tools Subsystem Reference Manual. No other routines have been changed in how they are called, although the functionality and/or implementation of a routine may have changed.

In the following, NULL denotes the null pointer (defined in "=incl=/stdio.h" as "(char *) 0"). Note that, on the Primes, ASCII NUL is represented as octal 0200, while '\0', the zero character, has the octal value 0.

Finally, remember that "=cdefs=" includes the files "=incl=/stdio.h", "=incl=/ctype.h", "=incl=/swt.h", and "=incl=/ascii.h", so their contents are automatically available, unless you specify the "-f" option.

UNIX System Calls

This section describes the routines in "ciolib" which are not part of the Standard I/O Library per se, but which emulate Unix system calls.

The Unix i/o system calls operate on integers, called file descriptors. Due to the similarity with Software Tools file descriptors, these routines usually act as interludes to their SWT counterparts, but return the values described in the Unix User's Manual.

. chdir --- change directory

Calling Information:

```
int chdir (path)
char *path
```

'Chdir' is used to change the current working directory. It uses the SWT routine 'getto' to actually change directory. 'Chdir' returns 0 if it succeeded, -1 if it failed.

Note that under Primos, if a program does a 'chdir', you will be in the new directory when the program exits, not where you were when the program started.

. **close --- close an open fd**

Calling Information:

```
int close (fd)
int fd;
```

'Close' closes a file associated with the file descriptor 'fd' returned by 'creat' or 'open'. 'Close' flushes any data buffers associated with the file and returns 0 if it was successful. If an error occurs, 'close' returns -1. This is not the same as SWT's 'close' (it's a macro), so "=incl=/stdio.h" must be included for 'close' to work as described.

. **creat --- create a file**

Calling Information:

```
int creat (name, mode)
char *name;
int mode;   /* protection mode; not used on Prime */
```

'Creat' creates and opens the file 'name' with WRITE access and returns a file descriptor. The new file has protection keys of "a/" (owner has all permissions). If the file 'name' already exists, 'creat' opens it for writing and truncates it to length 0. An existing file must have either "wt/" or "a/" protection keys (owner has **both** write and truncate permission). A return value of -1 indicates that the file cannot be created or that an attempt was made to 'creat' an existing file with the wrong protection keys. 'Mode' is ignored in this implementation.

. **open --- open a file, return a SWT fd**

Calling Information:

```
int open (name, mode)
char *name;
int mode;
```

'Open' provides a "Unix-style" call to open a file for reading and/or writing and returns a file descriptor. 'Mode' = 0 specifies read access, 1 specifies write access, and 2 specifies read/write access. A return value of -1 indicates that the file does not exist (as determined by filtst(2)), or cannot be opened (access mode does not match protection keys, or no free file descriptors are available) or that 'mode' was invalid. The C 'open' is not the same as SWT's 'open' (it's a macro), and requires that "=incl=/stdio.h" be included to function correctly.

. **exit --- exit from this program**

Calling Information:

```
int exit (exit_val)
int exit_val;   /* not used on the Primes */
```

'Exit' closes all open files and returns to the Subsystem (or to bare Primos). Temporary files that may have been created during program execution remain in directory "=temp=". 'Exit_val' is unused.

. **getpid --- return the current process number**

Calling Information:

```
int getpid()
```

'Getpid' returns the current process number. It uses the Subsystem routine 'date' to retrieve this information from Primos.

. **lseek --- position to a designated word in file**

Calling Information:

```
long lseek (fd, offset, origin)
int fd, origin;
long offset;
```

'Lseek' positions the read/write pointer for the file associated with file descriptor 'fd' (returned by 'creat' or 'open') to the word designated by 'offset' and 'origin'. If 'origin' = 0, 'offset' is the number of words from the beginning of the file. If 'origin' = 1, 'offset' is the number of words forward(backward) from the current position. If 'origin' = 2, 'offset' is the number of words past(before) the end of the file. See 'fseek' for further discussion.

If 'lseek' succeeds, it returns the current file position; otherwise it returns -1. 'lseek' calls 'markf', which will flush the buffers associated with 'fd'.

. **read --- read raw words from a file**

Calling Information:

```
int read (fd, buf, nw)
int fd, nw;
char *buf;
```

'Read' reads words from the file associated with the file descriptor 'fd' (returned by 'creat' or 'open') until 'nw' words have been read or until it encounters the end of file. If 'fd' is attached to a terminal device, 'read' will collect characters until it encounters a NEWLINE.

If an error occurred, 'read' returns -1; if 'read' encounters the end of the file or if a disk error occurred, it returns 0, so both -1 and 0 should be taken as error returns. Otherwise, 'read' returns the number of words transferred to 'buf'.

. **unlink --- delete a file**

Calling Information:

```
int unlink (path)
char *path;
```

Since Primos does not support links to files, 'unlink' always removes the file. If the file is open by any other user, or if the file does not have protection keys "t/" or "a/" (owner has truncate permission) 'unlink' will fail. 'Unlink' returns -1 on failure and 0 otherwise.

. **write --- write raw words to a file**

Calling Information:

```
int write (fd, buf, nw)
int fd, nw;
char *buf;
```

'Write' writes 'nw' words from 'buf' to the file associated with a file descriptor 'fd' (returned by 'creat' or 'open'). If an error occurs or if 'fd' is attached to "/dev/null", 'write' returns -1. Otherwise, 'write' returns the number of words written.

The C Standard I/O Library

The following routines are those listed as "3S", i.e. the actual Standard I/O Library. Input/Output operations in the Standard I/O Library occur on objects of type "FILE *". These are known variously as file pointers, or I/O streams.

The routines are listed below in roughly alphabetical order. However, logically associated routines (and/or macros) are grouped together.

. **ctermid --- return a filename for a terminal**

Calling Information:

```
char *ctermid (s)
char *s;
```

'Ctermid' returns the standard Georgia Tech SWT terminal name "/dev/tty". If 's' is not NULL, then 'ctermid' copies "/dev/tty" into it, and returns 's'. 'S' should be at least 'L_ctermid' characters long. 'L_ctermid' is defined in "=incl=/stdio.h".

. **cuserid --- return the user's login name**

Calling Information:

```
char *cuserid (s)
char *s;
```

'Cuserid' returns the user's login name consisting of 'L_cuserid' - 1 or fewer lower case non-blank ASCII characters followed by '\0'. ('L_cuserid' is defined in "=incl=/stdio.h".)

. **fclose --- close a stream**

Calling Information:

```
int fclose (stream)
FILE *stream;
```

'Fclose' closes the file and flushes the buffer associated with 'stream'. 'Fclose' returns 0 if the close was successful, EOF (-1) otherwise.

. **ferror --- indicate if an error has occurred on a given stream**

Calling Information:

```
int ferror (stream)
FILE *stream;
```

'Ferror' returns TRUE if an error has occurred while doing i/o on 'stream.' It returns FALSE otherwise. This is actually a macro in "=incl=/stdio.h".

. **feof --- indicate if EOF has occurred on a given stream**

Calling Information:

```
int feof (stream)
FILE *stream;
```

'Feof' returns TRUE if EOF has occurred on 'stream', and FALSE otherwise. 'Feof' should be used to find out if EOF has actually occurred, particularly when using 'fread' and 'fwrite'. This is actually a macro in "=incl=/stdio.h".

. **clearerr --- clear any errors associated with a given stream.**

Calling Information:

```
int clearerr (stream)
FILE *stream;
```

'Clearerr' will clear all of the error and EOF flags associated with 'stream'. This is actually a macro in "=incl=/stdio.h".

. **fileno --- return a Subsystem file descriptor**

Calling Information:

```
int fileno (stream)
FILE *stream;
```

'Fileno' is a macro in "=incl=/stdio.h" which returns the Software Tools Subsystem file descriptor associated with 'stream'. (Each FILE structure contains a Subsystem file descriptor, along with other information that the programmer should not need to access.) This permits you to use Subsystem routines that require a file descriptor rather than a file pointer, for instance:

```
FILE *fp;
...
fp = fopen ("file", "w");
/* do formatted i/o with a Subsystem routine */
print (fileno (fp), "i = *d*n", i);
```

. **fflush --- flush all buffers for a stream**

Calling Information:

```
int fflush (stream)
FILE *stream;
```

'Fflush' ensures that the contents and position of the open file reflect all output and positioning operations performed by the program. In other words, any in-memory C library and operating system buffers are flushed to disk, so that the permanent file matches the "logical" file (the file that the program has been working with). This is analogous to making changes to a file with the screen editor, and then issuing a "w" command to force the changes back out to the permanent file.

Please note that 'fflush' called on a disk file anywhere other than after a NEWLINE has been read or written may cause undesirable results, since Primos measures file positions in words and the i/o library writes in units of bytes. Flushing in the middle of a line can cause the compressed-blank count to be lost on an input file or can cause an additional '\0' (padding the last word) to be written to an output file. 'Fflush' also dumps the stream's 'ungetc' buffer.

'Fflush' returns EOF (-1) if the flush failed, 0 if it was successful.

. **fopen --- open an i/o stream**

Calling Information:

```
FILE *fopen (name, mode)
char *name, *mode;
```

'Fopen' opens a file 'name' and returns a pointer to an i/o stream. If the file does not exist, 'fopen' will create it. The stream has the mode specified in 'mode':

Mode	equivalent SWT mode
"r"	read, file not truncated
"r+"	read/write, file not truncated
"w"	write, file truncated
"w+"	read/write, file truncated
"a"	append for writing, file not truncated
"a+"	read/append for writing, file not truncated

Opening a file for write ("w", "w+") access truncates the file to length 0, while opening it for read ("r", "r+") or append ("a", "a+") access does not. The file pointer is positioned to the beginning of the file for both read and write access, and to the end of the file for append access. Append mode forces all writes to occur on the end of the

file, even if the file was opened for reading as well, and it is not currently at the end of the file.

'Fopen' returns NULL if no streams are available, if an invalid mode is supplied, if any of the arguments are bad, or if the access mode does not match the Primos protection keys.

. **freopen --- associate a new file with an opened stream**

Calling Information:

```
FILE *freopen (name, mode, stream)
char *name, *mode;
FILE *stream;
```

'Freopen' closes the file currently associated with 'stream', opens the file 'name' with mode 'mode' (same as in 'fopen'), and associates it with 'stream'. This function finds use in associating named files with the standard stream identifiers 'stdin', 'stdout', and 'stderr'.

'Freopen' returns NULL if the mode specified is invalid or if the file 'name' cannot be opened. If the file does not exist, 'freopen' will create it. In any case, 'stream' will be closed first.

Use of this routine is normally not possible in a non-Unix environment; Software Tools is an exception, since its file descriptors are very similar to those used by Unix i/o system calls.

. **fdopen --- associate a stream with an opened file**

Calling Information:

```
FILE *fdopen (fd, mode)
int fd;
char *mode;
```

'Fdopen' gets an i/o stream and associates it with the file descriptor 'fd' returned by 'creat' or 'open'. The stream has the mode specified in 'mode'; 'mode' may take the same values as the 'mode' argument to 'fopen'. This implementation of 'fdopen' **does not** check to make sure that modes of the file descriptor and the stream are the same.

The function returns NULL if an invalid mode is specified or if there are no free i/o streams. Successful execution returns a pointer to the newly assigned stream.

. **fread --- read raw words from a stream**

Calling Information:

```
int fread (ptr, itemsize, nitems, stream)
char *ptr;
int itemsize, nitems;
FILE *stream;
```

'Fread' reads 'nitems' * 'itemsize' words from 'stream' into the buffer addressed by 'ptr'. ('Itemsize' may be determined using the **sizeof** operator.) The function returns the number of items of size 'itemsize' read without error. If an error occurs or if it encounters end-of-file, 'fread' returns 0. Results are unpredictable if more words are requested than there is space in the buffer.

. **fwrite --- write raw words to a stream**

Calling Information:

```
int fwrite (ptr, itemsize, nitems, stream)
char *ptr;
int itemsize, nitems;
FILE *stream;
```

'Fwrite' writes 'itemsize' * 'nitems' words onto 'stream' from the buffer pointed to by 'ptr'. If an error occurs, 'fwrite' returns 0; otherwise it returns the number of successfully written items of size 'itemsize'.

. **fseek --- position to a designated word in a stream**

Calling Information:

```
int fseek (stream, offset, origin)
FILE *stream;
long offset;
int origin;
```

'Fseek' first flushes the stream buffers (including the 'ungetc' buffer) to clear up any pending i/o on 'stream' and then positions the read/write pointer to the word specified by 'offset' and 'origin'. If 'origin' = 0, 'offset' is the number of words from the beginning of the file. If 'origin' = 1, 'offset' is the number of words forward (backward) from the current position. If 'origin' = 2, 'offset' is the number of words past (before) the end of the file. 'Fseek' returns -1 if an error occurs or 0 if it succeeds.

There are several things to note about 'fseek'. First, it is not possible to seek past the end of a Primos file; zero words ('\0's) must be written to extend the file. Second, positioning to the end of a Primos sequential-format file requires Primos to read all of the blocks in the file (i.e. 'origin' = 2 can be quite slow). Third, since Primos text

files contain blank compression and '\0' padding, an 'fputs' of a 30-character string will probably not change the file pointer's position by 30; 'ftell' at the beginning of a line is the only reliable way to obtain a file position of a line in a text file. Finally, flushing the stream buffers may have undesirable results if it occurs during the formation of a line (i.e. before 'fputs', 'fprintf', etc. have put out a complete line).

. **rewind --- rewind to beginning of stream**

Calling Information:

```
int rewind (stream)
FILE *stream;
```

'Rewind' positions the read/write pointer associated with 'stream' to the beginning of the file. It is equivalent to "fseek (stream, 0L, 0)". Returns 0 if it was successful or -1 if an error occurred. (Under Unix, 'rewind' returns no value).

. **ftell --- return absolute position in a stream**

Calling Information:

```
long ftell (stream)
FILE *stream;
```

'Ftell' returns the current word position in 'stream' after flushing the stream buffers (and the 'ungetc' buffer) to clear any pending i/o on the stream.

Under Primos, 'ftell' may actually corrupt (slightly) an output text file when it flushes the stream buffers. On text files, 'markf' is the only reasonable way to determine the position of the beginning of a line. See the comments under 'fseek'. 'Ftell' returns -1 if an error occurs.

. **getc --- get a character from a stream**

. **getchar --- get a character from 'stdin'**

Calling Information:

```
int getc (stream)
FILE *stream;

int getchar()
```

'Getc' obtains the next character from 'stream'. If there are no more characters, or if an error occurs, 'getc' returns EOF (-1).

'Getchar' is a macro; it is defined as 'getc(stdin)' in "=incl=/stdio.h".

. **fgetc --- get next character from stream**

Calling Information:

```
int fgetc (stream)
FILE *stream;
```

'Fgetc' is another function which does what 'getc' does. It was initially created to serve as a real function that one could take the address of etc., since under Unix, 'getc' is a macro. On the prime, both 'getc' and 'fgetc' are functions.

. **getw --- get a machine word from a stream**

Calling Information:

```
int getw (stream)
FILE *stream;
```

'Getw' returns a single 16-bit word from a stream or EOF (-1) if an error occurred or no more characters were available. If 'stream' is attached to a terminal, 'getw' returns EOF when a NEWLINE is encountered. Since EOF could be an actual word value, 'feof' should be used to see if end of file has actually occurred.

. **gets --- get a string (up to a newline) from 'stdin'**

Calling Information:

```
char *gets (s)
char *s;
```

'Gets' copies the next line from 'stdin' into 's', discarding the NEWLINE and terminating 's' with '\0'. If 'gets' returns a line, the function return value is 's'; otherwise it is NULL.

. **fgets --- get a string from a stream**

Calling Information:

```
char *fgets (line, size, stream)
char *line;
int size;
FILE *stream;
```

'Fgets' fetches the next line from 'stream' by copying characters into 'line' from the stream buffer until 'size'-1 characters have been copied or until it encounters the next NEWLINE character. The NEWLINE character is kept. The function appends '\0' as the last character in 'line'. 'Fgets' returns 'line' if it obtained a line, and NULL otherwise.

. **popen --- initiate a "pipe" to/from a process**

. **pclose --- close a stream obtained from popen**

Calling Information:

```
FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

'Popen' takes two string arguments. The first, 'command', is a command to be executed by the Software Tools shell. The second, 'type', is either "r" or "w", to read from the standard output of the command, or to write to the standard input of the command, respectively. The function return is a stream which can be treated like any other object of type "FILE *".

'Popen' will return NULL if 1) another "pipe" is still open, 2) the 'type' argument is invalid, 3) the shell could not execute the command, or 4) needed temporary files could not be created.

'Pclose' closes the stream obtained from 'popen'. It returns 0 if it could successfully close the "pipe", otherwise it returns -1.

For programs that use "=lib=/nciolib", 'popen' always returns NULL, and 'pclose' always returns -1.

See the help on shell(2) in the Software Tools Subsystem Reference Manual for some caveats when dealing with the shell.

. **printf --- formatted output to 'stdout'**

. **fprintf --- formatted output to a stream**

. **sprintf --- formatted memory to memory conversion**

Calling Information:

```
int printf (control [, arg1, arg2, ..., arg10])
char *control;
untyped arg1, arg2, ..., arg10;

int fprintf (stream, control [, arg1, arg2, ..., arg10])
FILE *stream;
char *control;
untyped arg1, arg2, ..., arg10;

int sprintf (string, control [, arg1, arg2, ..., arg10])
char *string;
char *control;
```

untyped arg1, arg2, ..., arg10;

'Printf' formats its arguments ('arg1', ..., 'arg10') according to conversion specifications in 'control' and outputs the resulting character string on 'stdout'. The arguments may be pointers (i.e., to strings) or names of variables (e.g., ints, floats, ...). 'Fprintf' does the same thing, but to the named 'stream'. 'Sprintf' places the formatted output into 'string'. These routines take care of accessing the arguments according to the specifications of the 'control' string. In the following discussion 'printf' should be taken as a generic name for all three functions.

The 'control' string contains literal characters, which are copied to the "output" directly, and up to 10 conversion specifiers, each of which must have a corresponding argument. These routines conform as closely as possible to the specifications given for 'printf' in the UNIX System V User's Manual (Release 1).

A conversion specifier may consist of the following:

required	%	Begins conversion specification.
optional	flag	Modifies the meaning of the conversion specifier.
optional	<number>	Minimum field width. More space will be used if needed. If <number> begins with "0", then "0" will be used as a padding character; otherwise, 'printf' pads the output field with blanks.
optional	.	Separates field width and precision.
optional	<number>	Precision: maximum number of characters to print from a string or maximum number of digits to the right of the decimal point in a real number.
optional	[lh]	Size of argument indicator. 'l' indicates a long integer, 'h' a short. The 'h' modifier is recognized, but has no effect on the conversion.
required		Conversion specifier.

A field width or precision may be a '*' instead of a decimal integer. In this case, the next argument in the argument list will be treated as an integer and used for the field width or precision.

The flag may be one of the following:

- Left justify the conversion within the field.

+ The result of a signed conversion is always signed. I.e. a '+' will be prepended if the result is positive.

<blank> If the first character of a signed conversion is not a minus sign, the result will be prepended with a blank. The '+' flag overrides the <blank> flag.

Convert the result to an alternate form. This flag has no effect on the **c, d, s,** and **u** conversion specifiers. For **o** conversion, the precision is increased so that the result has a leading 0. For **x** (**X**) conversion, the result will have a leading **0x** (**0X**). For **e, E, f, g,** and **G** conversions, the result should always have a decimal point, even if there are no digits following the decimal point.

The conversion characters and their meanings are:

d,o,u,x,X Interpret the corresponding argument as a decimal, octal (no leading "0"), unsigned decimal or hexadecimal (no leading "0x"), integer, respectively. For **x** conversion, the letters **abcdef** are used, while for **X** conversion, the letters **ABCDEF** are used.

c Interpret the argument as a character (unpacked).

s The argument is a '\0'-terminated string: output characters until the correct precision has been achieved, or until '\0' is encountered, whichever comes first.

e,E Interpret the corresponding argument as a double-precision floating-point number and print it in the form

[-]m{m}.n{n}**e**±xx.

E format will cause the exponent to start with **E** instead of **e**.

f — Interpret the corresponding argument as a double-precision floating-point number and print it in the form

```
[-]m{m}.n{n}
```

with 'precision' digits to the right of the decimal point. If 'precision' is greater than 14, at most 6 significant digits will be printed.

g,G — Use the shortest of %e and %f formats. G format indicates %E instead of %e.

If the character "%" follows the initial "%" of the control specifier, the pair is taken as a literal character "%". 'Printf' returns the number of successfully printed characters or EOF (-1) if an error occurred.

Note that the old style (undocumented) capital letter conversion specifiers, which indicated 'long' arguments (e.g. %D for **long int**), are not supported. Use %ld (for example) instead of %D.

. **putc --- put a character on a stream**

. **putchar --- put a character onto 'stdout'**

Calling Information:

```
int putc (ch, stream)
char ch;
FILE *stream;

int putchar (c)
char c;
```

'Putc' puts the single character in 'ch' on 'stream'. If an error occurs, 'putc' returns EOF (-1); otherwise it returns the character just written.

'Putchar' is a macro; it is defined as 'putc(c, stdout)' in "=incl=/stdio.h".

. **fputc --- put a character on a stream**

Calling Information:

```
int fputc (c, stream)
char c;
FILE *stream;
```

'Fputc' is another function which does what 'putc' does. It was initially created to serve as a real function that one could take the address of etc., since under Unix, 'putc' is a macro. On the prime, both 'putc' and 'fputc' are func-

tions.

. **putw --- put raw words on a stream**

Calling Information:

```
int putw (w, stream)
int w;
FILE *stream;
```

'Putw' writes a single 16-bit word on 'stream'. After a successful write 'putw' returns the word written, while it returns EOF (-1) if an error occurred. If 'stream' is attached to "/dev/null", 'putw' always returns EOF.

. **puts --- put a string on 'stdout'**

Calling Information:

```
int puts (s)
char *s;
```

'Puts' appends a NEWLINE to the '\0'-terminated string 's' and prints it on standard output. If an errors occurs, 'puts' returns EOF (-1).

. **fputs --- put a string on a stream**

Calling Information:

```
int fputs (s, stream)
char *s;
FILE *stream;
```

'Fputs' puts the '\0'-terminated string 's' on 'stream'. Note that 's' need not contain a NEWLINE character and that 'fputs' will **not** supply one. 'Fputs' returns EOF (-1) if an error occurred, zero otherwise.

. **scanf --- formatted input conversion from 'stdin'**

. **fscanf --- formatted input conversion from stream**

. **sscanf --- formatted input conversion from a string**

Calling Information:

```
int scanf (control [, arg1, arg2, ..., arg10])
char *control;
char *arg1, *arg2, ..., *arg10;

int fscanf (stream, control [, arg1, arg2, ..., arg10])
FILE *stream;
char *control;
char *arg1, *arg2, ..., *arg10;
```

```
int sscanf (string, control [, arg1, arg2, ..., arg10])
char *string;
char *control;
char *arg1, *arg2, ..., *arg10;
```

'Scanf' reads characters from 'stdin', formats them according to conversion specifications in the control string, and stores the results in the variables pointed to by corresponding arguments 1-10. 'Fscanf' reads its input from the named 'stream'. 'Sscanf' reads characters from the named 'string'. In the following discussion, 'scanf' should be taken as a generic name for all three functions.

The control string may contain white space, which is skipped, literal characters (which **must** match corresponding characters from the input stream) and at most 10 conversion specifiers consisting of the following:

required	%	Begins conversion specification.
optional	*	Suppresses assignment of input field (does **not** skip argument).
optional	<number>	Numeric field width.
optional	l	Read variable as 'long'.
optional	h	Read variable as 'short'.
required		Conversion specifier:

'['[^]<char>[-<char>]{<char>[-<char>]}']'
: Input a string of characters until finding a character not included in the bracketed set. E.g., "[a-zA-Z]" stops reading when a non-alphabetic character is encountered. If the first character in the set is '^', input characters are read until finding a character included in the bracketed set. E.g., "[^]" reads until a blank is found.

d,o,u,x
: Input decimal, octal, unsigned decimal, or hexadecimal integers.

c
: Read single character(s) including blanks.

s
: Input a string delimited by white space, or until <width> characters have been read. The variable in which the string is to be stored must be long enough to contain the string followed by a '\0'.

e,f,g Read a floating point number of the form

[±]m{m}.n{n}[**E**[±]x{x}].

The value returned in both cases is a **float**. Use the "l" option to specify a **double** value.

If the character '%' follows the initial '%' of the control specifier, the pair is taken as a literal character '%'.

'Scanf' conversions stop when EOF is seen, when the control string is exhausted, or when an input character conflicts with the control string.

'Scanf' returns the number of successfully assigned input items, or EOF (-1) if none were found.

. **setbuf --- set buffering on a stream**

Calling Information:

```
setbuf (stream, buf)
FILE *stream;
char *buf;
```

Under Software Tools and Primos, 'setbuf' is a null (do nothing) function. Under Unix, it allows the user to associate a character array as the buffer for a given stream. 'Setbuf' is provided to make porting of programs easier. It is an actual function, just in case there is code which takes its address, or does something else strange of this nature.

. **system --- pass a command to the Software Tools Shell**

Calling Information:

```
int system (cmd)
char *cmd;
```

'System' passes a command 'cmd' to the Software Tools shell to be executed, and returns TRUE if the call was successful. If the call failed, 'system' returns FALSE. See the help on shell(2) in the Software Tools Subsystem Reference Manual for some caveats when dealing with the shell.

For programs that use "=lib=/nciolib", 'system' always returns FALSE.

NOTE: This routine has changed from the previous release of the C compiler. Before Version 9 of Software Tools, it was not possible to call the Software Tools shell, so the 'system' routine called the Primos command interpreter. If you still need to call Primos, see the help on sys$$(2) in the Software Tools Subsystem Reference Manual.

. **tmpfile --- create a temporary file**

Calling Information:

```
FILE *tmpfile ()
```

'Tmpfile' returns a pointer to a temporary file opened with "w+" access. The name of the file is inaccessible from inside a program. The file actually created bears the process unique name "=temp=/tm###" (where ### ranges from 1-999) and remains in the "=temp=" directory after the creating process terminates. If no file can be created, 'tmpfile' returns NULL.

. **tmpnam --- return a filename for a temporary file**

Calling Information:

```
char *tmpnam (s)
char *s;
```

'Tmpnam' returns a unique temporary file name "=temp=/ct=pid=###" where ### is a process unique number 0-999 and =pid= is the current process id. Names are recycled after all 1000 have been used.

. **tempnam --- return a filename for a temporary file**

Calling Information:

```
char *tempnam (dir, pfx)
char *dir, *pfx;
```

'Tempnam' is designed to give the user a little more control over the name of his temporary file. The directory for the tempfile will be taken from the environment variable TMPDIR, if it exists. Otherwise, if 'dir' is not NULL, 'dir' will be used. If 'dir' is NULL, the directory will be P_tmpdir (defined in "=incl=/stdio.h").

If 'pfx' is not NULL, it will be used as the prefix for the file name. Otherwise, the prefix will be "ct".

The full file name will consist of the directory name, a '/', then the prefix, the process id number, and a number between 0 and 999. The number changes after each call to 'tempnam'. After all 1000 have been used, they will be recycled.

'Tempnam' uses 'malloc' to create space for the string containing the file name. The pointer returned by 'tempnam' can be used later in a call to 'free'.

'Tempnam' returns NULL if it could not allocate enough space for the string to hold the generated file name.

. **ungetc --- push a single character back on an input stream**

Calling Information:

```
int ungetc (ch, stream)
char ch;
FILE *stream;
```

'Ungetc' places 'ch' in a single-character buffer associated with 'stream'. The next call to 'getc' or 'fgetc' retrieves 'ch'. Attempting to push more than one character back onto the input stream or using 'ungetc' on a closed stream produces an error return of EOF (-1). Otherwise, 'ungetc' returns 'ch'.

. **ftrunc --- truncate a stream at the current position**

Calling Information:

```
int ftrunc (stream)
FILE *stream;
```

'Ftrunc' flushes all file and 'ungetc' buffers for the file associated with 'stream' and truncates the file at its current position. The file must be opened with write access. (If 'fopen' is used to open the file, then the only really useful values for 'mode' are "r+" and "a+", because read access is usually necessary to position the file correctly and because opening the file for write ("w", "w+") access always truncates the file.) 'Ftrunc' returns 0 if it succeeded, -1 otherwise.

'Ftrunc' is not part of the Standard I/O Library per se, but is provided in order to allow access to this capability of the Primos file system.

Unix Subroutines For C Programs

The following routines are those listed as "3C", i.e. the routines which are loaded along with every Unix C program, but which are not guaranteed to be on other non-Unix systems.

The character testing macros discussed below ('isalnum', 'isdigit', etc.) are valid on integers in the range -1 to 0377 (EOF to ASCII DEL). They merely return FALSE on characters in the range -1 to '\177'. The result of these macros on values less than -1 or greater than 0377 is undefined.

These macros do not necessarily return "true" values equal to the symbolic constant TRUE defined in "=incl=/stdio.h". Rather, they return logical true, i.e. non-zero, and logical false, i.e. zero. They should be used as conditions, not compared against TRUE and FALSE. In other words, use:

```
if (islower (c)) { /* stuff */ }
```

and not

```
if (islower (c) == TRUE) { /* stuff */ }
```

The routines are listed below in roughly alphabetical order. However, logically associated routines (and/or macros) are grouped together.

- **a64l --- convert base-64 string to long integer**
- **l64a --- convert long integer to base-64 string**

Calling Information:

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

'A64l' takes a '\0'-terminated string containing a base-64 representation, and returns the corresponding long. If the string has more than six characters, only the first six are used. 'L64a' takes a long integer, and returns a pointer to a string with the corresponding base-64 representation.

These routines use the following characters as digits in the base-64 notation. '.' for 0, '/' for 1, '0' through '9' for 2-11, 'A' through 'Z' for 12-37, and 'a' through 'z' for 38-63.

- **abort --- generate a "fault"**

Calling Information:

```
int abort ()
```

Under Unix, 'abort' generates a **SIGIOT** fault, which causes the program to exit and dump core. The user may catch this signal. Under Software Tools or Primos, this routine simply exits.

- **abs --- return integer absolute value**

Calling Information:

```
int abs (x)
int x;
```

'Abs' returns the absolute value of its integer argument. This is a fast, assembly language routine, local to the C library.

. **atof --- convert character string to double precision real**

Calling Information:

```
double atof (str)
char *str;
```

Converts a string of characters 'str' to a double precision real number. Conversion stops when 'atof' encounters a non-numeric character.

. **atoi --- convert character string to integer**

Calling Information:

```
int atoi (str)
char *str;
```

'Atoi' converts a string of characters 'str' to a base-10 integer. Conversion stops when 'atoi' encounters a non-numeric character. 'Atoi' uses 'gctoi', so it will recognize the Ratfor "radix notation" (e.g. 8r377).

. **atol --- convert character string to long integer**

Calling Information:

```
long atol (str)
char *str;
```

'Atol' converts a string of characters 'str' to a base-10 long (32-bit) integer. Conversion stops when 'atol' encounters a non-numeric character in 'str'. 'Atol' uses 'gctol', so it will recognize the Ratfor "radix notation" (e.g. 8r377).

. **strtol --- convert string to arbitrary base long integer**

Calling Information:

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;
```

'Strtol' takes a '\0'-terminated string in 'str', and returns the long integer it represents. 'Base' is the base of the string. If base is less than zero or greater than 36, 'strtol' will return. If base is zero, it will attempt to determine the base from the string itself. A leading '0' indicates octal, '0x' or '0X' indicates hexadecimal; otherwise, the string is assumed to be in decimal.

If the value of 'ptr' is not (char **) 0, the address of the character which terminated the string will be placed in *ptr. If no integer can be converted from the string, *ptr

is set to 'str', and 0 is returned.

. **getcwd --- get pathname of current working directory**

Calling Information:

```
char *getcwd (buf, size)
char *buf;
int size;
```

'Getcwd' returns a pointer to a string containing the SWT path name of the current directory. If 'buf' is not NULL, 'getcwd' will use 'buf' a buffer in which to place the name. Otherwise, it will use 'malloc' to dynamically allocate a buffer. In this case, the returned pointer can be used later in a call to 'free'. 'Size' is the size of the buffer to be 'malloc'ed, so it must include room for the trailing '\0'.

'Getcwd' returns NULL if size is less than or equal to 1, if 'malloc' could not allocate enough memory, or if one of the SWT routines 'follow' or 'gcdir$' failed.

. **getenv --- return value for "environment" variable**

Calling Information:

```
extern char **environ;

char *getenv (var)
char *var;
```

'Getenv' scans the environment list of name=value pairs pointed to by the external variable 'environ'. If 'var' is found, 'getenv' returns a pointer to its value. Otherwise, it returns NULL.

For programs which use "=lib=/nciolib", 'getenv' will always return NULL, and 'environ' is always equal to NULL.

. **getlogin --- get the login name**

Calling Information:

```
char *getlogin()
```

If the current process is a phantom, 'getlogin' returns NULL. Otherwise, it returns the user's login name, as obtained from 'cuserid'.

. **getopt --- get option letter from argument vector**

Calling Information:

```
extern char *optarg;
extern int optind;

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
```

'Getopt' returns the next option letter in 'argv' that matches a letter in 'optstring'. If a letter in 'optstring' is followed by a colon, then that option is supposed to have an argument, that may or may not be separated from it by white space. 'Optarg' is set to point to the beginning of the argument to the current option when 'getopt' returns.

'Getopt' sets 'optind' to the index of the next argument in 'argv' to be processed. Since 'optind' is external, it is initialized to zero.

When all options have been processed (i.e. when the first non-option is encountered), 'getopt' returns EOF. The special option -- can be used to delimit the end of the options. 'Getopt' will return EOF, and will skip the --.

'Getopt' returns a '?' and prints an error message on 'stderr' when it finds an option that is not in 'optstring'.

. **getpass --- read a password**

Calling Information:

```
char *getpass (prompt)
char *prompt;
```

'Getpass' disables echoing, and prints 'prompt' on 'stderr'. It then reads up to a newline or EOF from the terminal. It returns a pointer to a '\0'-terminated string of at most eight characters. If 'getpass' cannot use the TTY file descriptor, and if it cannot open "/dev/tty", it will read from 'stdin'. 'Getpass' turns echoing back on before returning.

. **isalnum --- indicate if a character is alphanumeric**

Calling Information:

```
int isalnum (ch)
char ch;
```

'Isalnum' returns TRUE if 'ch' falls in the range 'A' through 'Z', inclusive, in the range 'a' through 'z', inclusive, or if it lies between '0' and '9', inclusive. It

returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isalpha --- indicate if a character is alphabetic**

Calling Information:

```
int isalpha (ch)
char ch;
```

'Isalpha' returns TRUE if 'ch' lies between 'A' and 'Z', inclusive, or if it lies between 'a' and 'z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isdigit --- indicate if a character is a decimal digit**

Calling Information:

```
int isdigit (ch)
char ch;
```

'Isdigit' returns TRUE if 'ch' lies between '0' and '9', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isxdigit --- indicate if a character is a hexadecimal digit**

Calling Information:

```
int isxdigit (ch)
char ch;
```

'Isxdigit' returns TRUE if 'ch' falls between '0' and '9', inclusive, or if it falls between 'A' and 'F' inclusive, or 'a' and 'f' inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isupper --- indicate if a character is an upper case letter**

Calling Information:

```
int isupper (ch)
char ch;
```

'Isupper' returns TRUE if 'ch' lies between 'A' and 'Z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **islower --- indicate if a character is an lower case letter**

Calling Information:

```
int islower (ch)
char ch;
```

'Islower' returns TRUE if 'ch' lies between 'a' and 'z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isprint --- indicate if a character is printable**

Calling Information:

```
int isprint (ch)
char ch;
```

'Isprint' returns TRUE if 'ch' is a printable character. This includes all punctuation, letters, digits, and the space character ' '. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isgraph --- indicate if a character is printable and visible**

Calling Information:

```
int isgraph (ch)
char ch;
```

'Isgraph' is similar to 'isprint' above, except that it excludes the space character ' '. It returns TRUE if 'ch' has a graphic representation, FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **ispunct --- indicate if a character is punctuation**

Calling Information:

```
int ispunct (ch)
char ch;
```

'Ispunct' returns TRUE if 'ch' is neither an alphanumeric nor a control character. To use this macro, the file "=incl=/ctype.h" must be included first.

. **iscntrl --- indicate if a character is a control character**

Calling Information:

```
int iscntrl (ch)
char ch;
```

'Iscntrl' returns TRUE if 'ch' is an ASCII control character, i.e., if 'ch' falls in the range '\200' to '\237', inclusive, or if 'ch' equals '\377' (DEL). It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isascii --- indicate if character is within the ASCII character set**

Calling Information:

```
int isascii (ch)
char ch;
```

'Isascii' returns TRUE if 'ch' lies in the range '\200' to '\377', inclusive (Prime's ASCII representation). It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isspace --- indicate if a character is white space**

Calling Information:

```
int isspace (ch)
char ch;
```

'Isspace' returns TRUE if 'ch' is a space, a tab, a newline, a carriage return, a form feed or a vertical tab. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **malloc --- allocate memory**

. **alloc --- allocate memory (old name)**

Calling Information:

```
char *malloc (n)
int n;

char *alloc (n)
int n;
```

'Malloc' allocates 'n' words of memory and returns a pointer to the beginning of the storage block. If 64K words are requested or if zero words are requested, 'malloc' returns NULL.

'Alloc' is the name of the pre-Version 7 UNIX storage

allocator. It performs the same function as 'malloc'. Its use in new programs is strongly discouraged. It is provided to make porting code easier, and because it was in the first release of the C compiler.

. **calloc --- allocate memory for arrays or structures**

Calling Information:

```
char *calloc (n, size)
int n, size
```

'Calloc' allocates 'n' * 'size' words of memory for storing 'n' objects of 'size' words each. If successful, 'calloc' initializes all words to zero, and returns a pointer to the first word of the storage block. If 64K or more words are requested or if zero words are requested, 'calloc' returns NULL.

. **realloc --- change the size of previously allocated memory**

Calling Information:

```
char *realloc (ptr, size)
char *ptr;
int size;
```

'Realloc' reallocates a block of memory of 'size' words for a storage block previously allocated by 'malloc' or 'calloc' (0 < 'size' < 64K). The contents of the original storage block are preserved by copying to the newly allocated block. Therefore, the pointer 'ptr' passed as a parameter to 'realloc' **must** point to the beginning of a block allocated by 'malloc' or 'calloc' in order for the copy to work properly. Any existing pointers to the original data structure must be changed. (I.e. the contents of the old memory are preserved, but the same actual block of memory may not be used.)

'Realloc' returns a pointer to the first word of the new storage block or NULL if an error occurred.

. **free --- free allocated memory**

. **cfree --- free allocated memory (old name)**

Calling Information:

```
free (ptr)
char *ptr

cfree (ptr)
char *ptr;
```

'Free' frees a block of memory previously allocated by 'malloc', 'realloc', or 'calloc'. 'Free' will fail miserably if handed an arbitrary pointer; only pointers returned by 'mal-

loc', 'realloc' or 'calloc' are valid parameters.

'Cfree' is the name of the pre-Version 7 UNIX storage releaser. It performs the same function as 'free'. Its use in new programs is strongly discouraged. It is provided to make porting code easier, and because it was in the first release of the C compiler.

. **memccpy --- copy characters up to a character or some number**

Calling Information:

```
char *memccpy (s1, s2, c, n)
char *s1, *s2, c;
int n;
```

'Memccpy' copies characters (words, on the Prime) from the memory pointed to by 's1' into 's2' until it encounters 'c', or until 'n' characters have been copied. It returns a pointer to the character after the first occurrence of 'c', or NULL if 'c' was not found in the first 'n' characters of 's1'.

. **memchr --- return a pointer to a char within a memory area**

Calling Information:

```
char *memchr (s, c, n)
char *s;
int c, n;
```

'Memchr' returns a pointer to the first occurrence of 'c' within the first 'n' characters (words, on the Prime) of 's'. It returns NULL if 'c' does not occur.

. **memcmp --- compare arbitrary areas of memory**

Calling Information:

```
int memcpmp (s1, s2, n)
char *s1, *s2;
int n;
```

'Memcmp' looks only at the first 'n' words of its first two arguments. It returns an integer less than zero, equal to zero, or greater than zero, according as 's1' is lexicographically less than, equal to, or greater than 's2'.

. **memcpy --- copy arbitrary areas of memory**

Calling Information:

```
char *memcpy (s1, s2, n)
char *s1, *s2;
int n;
```

'Memcpy' copies 'n' characters from 's2' to 's1'. It returns 's1'.

. **memset --- initialize memory to a given value**

Calling Information:

```
char *memset (s, c, n)
char *s;
int c, n;
```

'Memset' sets the first 'n' characters (words) of 's' to 'c'. It returns 's'.

. **mktemp --- make a unique file name**

Calling Information:

```
char *mktemp (template)
char *template;
```

'Mktemp' replaces the contents of the string pointed to by 'template' with a unique file name, and returns 'template'. 'Template' should look like a file name with six trailing **X**s; 'mktemp' replaces the **X**s with a unique letter and the process id. (This implementation only requires four **X**s; six is recommended for portability to/from Unix systems.)

If there are no **X**s in the 'template', 'mktemp' returns NULL. The "unique" letter will recycled after 26 calls to 'mktemp'.

So that this routine does not conflict with the SWT 'mktemp', it is actually a macro, so "=incl=/stdio.h" must be included in order to use it.

. **rand --- return a random integer**

. **srand --- seed the random number generator**

Calling Information:

```
int rand()

srand (seed)
unsigned seed;
```

'Rand' uses 'rand$m' in the SWT Math Library. It returns a

number between 0 and 2^16-1. 'Srand' uses 'seed$m' to seed the random number generator. In keeping with the Unix semantics, if the user calls 'rand' before calling 'srand', the random number generator will be seeded with 1.

. **setjmp --- set up for non-local goto**

. **longjmp --- perform a non-local goto**

Calling Information:

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

longjmp (env, status)
jmp_buf env;
int status;
```

'Setjmp' saves the current stack frame in 'env' for later use. On every call, 'setjmp' returns 0. 'Longjmp' takes 'status' and performs a non-local "goto" to an environment ('env') saved by a previous call to 'setjmp'. The result of that operation allows execution to continue as if 'setjmp' had returned 'status' rather than 0 at the point of invocation. Use of these routines requires inclusion of "<setjmp.h>" before either of the routines is called. In particular "<setjmp.h>" does a **typedef** on the type 'jmp_buf', and contains a macro definition which is needed for 'setjmp' to return properly.

. **sleep --- sleep for the given number of seconds**

Calling Information:

```
int sleep (amount)
unsigned amount;
```

'Sleep' will sleep for 'amount' seconds. 'Sleep' simply calls the Primos 'sleep$' routine. It returns no value.

. **strcat --- concatenate two strings**

Calling Information:

```
char *strcat (t, s)
char *t, *s;
```

'Strcat' concatenates string 's' to string 't', terminating 't' with '\0'. The target string 't' is assumed to be large enough to accommodate all of the characters copied from 's'. 'Strcat' returns a pointer to 't', or NULL if either 's' or 't' is NULL.

. **strncat --- concatenate substring to string**

Calling Information:

```
char *strncat (t, s, n)
char *t, *s;
int n;
```

Concatenates at most 'n' characters of string 's' to string 't'. If 's' contains fewer than 'n' characters, then only "strlen (s)" characters will be copied. In any case, 'strncat' terminates 't' with '\0' and returns a pointer to 't'.

. **strcmp --- compare strings**

Calling Information:

```
strcmp (s1, s2)
char *s1, *s2;
```

'Strcmp' compares strings 's1' and 's2' and returns 0 if they are equal or if s1 = NULL and s2 = NULL. If *s1 > *s2 or if s2 = NULL, 'strcmp' returns a positive value; it returns a negative value if *s1 < *s2 or if s1 = NULL.

. **strncmp --- compare substrings**

Calling Information:

```
int strncmp (s1, s2, n)
char *s1, *s2;
int n;
```

'Strncmp' compares at most 'n' characters of 's1' and 's2'. It returns 0 if equal or if s1 = NULL and s2 = NULL; a positive value if *s1 > *s2 or if s2 = NULL; or a negative value if *s1 < *s2 or if s1 = NULL.

. **strcpy --- copy string**

Calling Information:

```
char *strcpy (t, s)
char *t, *s;
```

Copy string 's' to string 't'. 'Strcpy' assumes that 't' is large enough to receive all characters contained in 's'. If 't' is NULL 'strcpy' returns NULL; if 's' is NULL and 't' is non-NULL, 't' is set to the empty string ("").

. **strncpy --- copy substring to string**

Calling Information:

```
char *strncpy (s1, s2, n)
char *s1, *s2;
int n;
```

'Strncpy' copies at most 'n' characters from string 's2' to string 's1'. If 's2' contains more than 'n' characters, then 's1' will not be '\0'-terminated. If 's2' contains fewer than 'n' characters (including a terminal '\0'), then 's1' is '\0'-padded until it contains 'n' characters.

. **strlen --- return length of string**

Calling Information:

```
int strlen (s)
char *s;
```

'Strlen' returns the length of a string 's' excluding the terminating '\0' character.

. **strchr --- find character in string**

Calling Information:

```
char *strchr (s, c)
char *s, c;
```

Returns pointer to first occurrence of character 'c' in string 's'; if 'c' is not found 'strchr' returns NULL.

. **strrchr --- find character in string (last occurrence of)**

Calling Information:

```
char *strrchr (s, c)
char *s, c;
```

'Strrchr' returns a pointer to the last occurrence of the character 'c' in the string 's'. If 'c' does not occur in 's', NULL is returned. ('Strrchr' also works if 'c' = '\0'.)

. **strpbrk --- find one of a class of characters in a string**

Calling Information:

```
char *strpbrk (s1, s2)
char *s1, *s2;
```

Returns a pointer to the first character in 's1' matching any character in string 's2', or NULL if no character in 's2' is in 's1'. Both 's1' and 's2' must be non-NULL.

. **strspn --- find qualified substring**

Calling Information:

```
int strspn (s1, s2)
char *s1, *s2;
```

'Strspn' returns the length of the initial substring of 's1' that is made entirely of characters from 's2'. If either 's1' = NULL or 's2' = NULL, 'strspn' returns 0.

. **strcspn --- find qualified substring**

Calling Information:

```
int strcspn (s1, s2)
char *s1, *s2;
```

'Strcspn' returns the length of the initial substring of 's1' not having any characters contained in 's2'.

. **strtok --- find tokens in a string**

Calling Information:

```
char *strtok (s1, s2)
char *s1, *s2;
```

'Strtok' returns a pointer to the start of each token in 's1'. Tokens are defined as contiguous strings of characters delimited by separators. 'Strtok' skips over any leading separators. 'S2' contains 1 or more characters to be considered as token separators. When 'strtok' finds a token, it replaces the terminating delimiter with '\0'. If it can't find a token with the current set of delimiters, 'strtok' returns NULL; however, if 's1' has already been successfully searched for a token, the terminating '\0' in 's1' is considered a valid delimiter. Thus, the final token in 's1' can be retrieved without any special finagling (the '\0' of 's2' is never considered a valid separator during the scan of 's1' for delimiters).

On the first call to 'strtok', 's1' should point to a valid string, while on subsequent calls 's1' should be NULL so that the entire string is scanned. The characters in 's2' may change from call to call searching the same 's1'.

. **index --- find character in string**

Calling Information:

```
char *index (s, c)
char *s, c;
```

Same as 'strchr'. This is the V7 (and Berkeley) UNIX routine; the name was changed with UNIX System III.

. **rindex --- find the last occurrence of character in string**

Calling Information:

```
char *rindex (s, c)
char *s, c;
```

Same as 'strrchr'. This is the V7 (and Berkeley) UNIX routine; the name was changed with UNIX System III.

. **toascii --- convert a char/int to a valid ASCII value**

Calling Information:

```
char toascii (ch)
char ch;
```

'Toascii' returns its argument converted into a valid ASCII value. When unpacking packed character strings, 'toascii' can be used to obtain the high character after shifting the packed word right by 8 bits. The low character can be gotten by passing the whole integer to 'toascii'. To use this macro, the file "=incl=/ctype.h" must be included first.

. **toupper --- convert lower case character to upper case**

Calling Information:

```
char toupper (ch)
char ch;
```

If 'ch' is a lower case letter, 'toupper' returns the corresponding upper case letter. Otherwise it returns 'ch' unchanged. This is actually a fast, assembly language routine, declared in "=incl=/ctype.h".

. **tolower --- convert upper case character to lower case**

Calling Information:

```
char tolower (ch)
char ch;
```

If 'ch' is an upper case letter, 'tolower' returns the corresponding lower case letter. Otherwise it returns 'ch' unchanged. This is actually a fast, assembly language routine, declared in "=incl=/ctype.h".

. **_toupper --- blindly convert a character to upper case**

Calling Information:

```
char _toupper (ch)
char ch;
```

'_toupper' returns 'ch' converted to upper case. It does **not** check that its argument is indeed a lower case letter. This function performs the Unix Version 7 'toupper', which was changed to '_toupper' in Unix System III to accommodate those who may still want it. To use this macro, the file "=incl=/ctype.h" must be included first.

. **_tolower --- blindly convert a character to lower case**

Calling Information:

```
char _tolower (ch)
char ch;
```

'_tolower' returns 'ch' converted to lower case. It does **not** check that its argument is indeed an upper case letter. This function performs the Unix Version 7 'tolower', which was changed to '_tolower' in Unix System III to accommodate those who may still want it. To use this macro, the file "=incl=/ctype.h" must be included first.

. **ttyname --- return the name of the terminal**

Calling Information:

```
char *ttyname (fd)
int fd;
```

'Ttyname' takes an integer file descriptor as an argument. If the device which is attached to the file descriptor is a terminal, it returns the string "/dev/tty", otherwise it returns NULL. (Under Unix, it would return the actual device name.)

The associated 'isatty' function already exists in SWT. It may used as is.

The C Math Library

The following routines are those listed as "3M", i.e. the C Math Library. These routines all take arguments of type **double**, and return type **double**.

You should include the file "=incl=/math.h" which declares these routines, before using them. This can be done with the line:

```
#include <math.h>
```

Most of these routines simply call the corresponding routine in the SWT math library, "vswtmath". See the SWT Math Library User's Guide for details on how these routines work, and under what condition(s) they will raise an error condition.

. **acos --- take arc cosine of a real**

Calling Information:

```
double acos (x)
double x;
```

This routine returns the value obtained from 'dacs$m' in SWT math library.

. **asin --- take arc sine of a real**

Calling Information:

```
double asin (x)
double x;
```

This routine returns the value obtained from 'dasn$m' in SWT math library.

. **atan --- take arc tangent of a real**

Calling Information:

```
double atan (x)
double x;
```

This routine returns the value obtained from 'datn$m' in SWT math library.

. **atan2 --- take arc tangent of x/y**

Calling Information:

```
double atan2 (x, y)
double x, y;
```

This routine first divides x by y. It passes the result of the division to the SWT math library routine 'datn$m', returning its result.

. **ceil --- return smallest integer not less than x**

Calling Information:

```
double ceil (x)
double x;
```

This routine uses the 'dint$m' routine in the SWT Math Library to remove the fractional part, and then adds 1.0 if its argument was positive.

. **cos --- take cosine of a real**

Calling Information:

```
double cos (x)
double x;
```

This routine returns the value obtained from the 'dcos$m' routine in the SWT math library.

. **cosh --- take hyperbolic cosine of a real**

Calling Information:

```
double cosh (x)
double x;
```

This routine returns the value obtained from the 'dcsh$m' routine in the SWT math library.

. **exp --- compute exponential (base e) of a real**

Calling Information:

```
double exp (x)
double x;
```

This routine returns the value obtained from the 'dexp$m' routine in the SWT math library (Raise e to 'x' power).

. **fabs --- compute the absolute value of a real**

Calling Information:

```
double fabs (x)
double x;
```

This routine returns the absolute value of its double precision argument. It is a fast, assembly language routine, local to the C library.

. **fmod --- do floating point modulus operation**

Calling Information:

```
double fmod (x, y)
double x, y;
```

'Fmod' returns 'x' if 'y' is zero. Otherwise, it returns a number f, of the same sign as x, such that x = iy + f for some integer i, and $|f| < |y|$.

. **hypot --- return Euclidean distance function**

Calling Information:

```
double hypot (x, y)
double x, y;
```

'Hypot' returns

```
sqrt (x * x + y * y)
```

It does not check for overflow of the multiplication and addition operations (although it should).

. **floor --- return largest integer not greater than x**

Calling Information:

```
double floor (x)
double x;
```

This routine uses the 'dint$m' routine in the SWT Math Library to remove the fractional part, and then subtracts 1.0 if its argument was negative.

. **log --- take the natural log (base e) of a real**

Calling Information:

```
double log (x)
double x;
```

This routine returns the value obtained from the 'dln$m' routine in the SWT math library.

. **log10 --- take the log base 10 of a real**

Calling Information:

```
double log10 (x)
double x;
```

This routine returns the value obtained from the 'dlog$m' routine in the SWT math library.

. **pow --- provide exponentiation for C programs**

Calling Information:

```
double pow (a, b)
double a, b;
```

'Pow' computes 'a' to the 'b'. 'Pow' calls 'powr$m' in the SWT math library. It may raise 'SWT_MATH_ERROR$' if the first argument is negative, or if the first argument is zero, and the second is negative or zero. The exception

will also be raised if the results of the calculation would cause an overflow.

. **sin --- take sine of a real**

Calling Information:

```
double sin (x)
double x;
```

This routine returns the value obtained from the 'dsin$m' routine in the SWT math library.

. **sinh --- take hyperbolic sine of a real**

Calling Information:

```
double sinh (x)
double x;
```

This routine returns the value obtained from the 'dsnh$m' routine in the SWT math library.

. **sqrt --- take square root of a positive real**

Calling Information:

```
double sqrt (x)
double x;
```

This routine returns the value obtained from the 'dsqt$m' routine in the SWT math library.

. **tan --- take tangent of a real**

Calling Information:

```
double tan (x)
double x;
```

This routine returns the value obtained from the 'dtan$m' routine in the SWT math library.

. **tanh --- take hyperbolic tangent of a real**

Calling Information:

```
double tanh (x)
double x;
```

This routine returns the value obtained from the 'dtnh$m' routine in the SWT math library.

Unix Special Library Routines

The following routines are those listed as "3X", i.e. the routines which require special libraries and/or include files.

- **assert --- put assertions into C programs**

  Calling Information:

  ```
  #include <assert.h>

  assert (expression)
  int expression;
  ```

  The 'assert' function is actually a macro which tests the boolean expression. If the expression is FALSE, 'assert' prints a message to 'stderr', and exits. The error message will contain the file name and line number of the 'assert' statement. The 'assert' macro is written in such a way that it can be used as a regular statement in a C program; it will not mess up **if-else** nesting, for instance.

  Defining NDEBUG before including "<assert.h>" or on the 'cc' (or 'ccl' or 'ucc') command line will turn 'assert' into a null (empty) macro.

- **logname --- return login name of user**

  Calling Information:

  ```
  char *logname()
  ```

  'Logname' returns a pointer to a string containing the user's login name. If the LOGNAME environment variable exists, 'logname' returns its value. Otherwise, if the template =user= has a value, that value is returned. If neither of those work, 'logname' returns NULL. Both of these methods are subject to counterfieting.

- **varargs --- portably write functions with a variable number of arguments**

  Calling Information:

  ```
  #include <varargs.h>

  function (va_alist)
  va_dcl
  va_list pvar;
  va_start (pvar);
  f = va_arg (pvar, type);
  va_end (pvar);
  ```

  The file "=incl=/varargs.h" contains a set of macros which allow you to write functions which will have a variable number of arguments in a portable (if slightly opaque) fashion.

va_alist — is used in place of the argument list inside the parentheses of a function header, to declare a **va**riable **a**rgument **list**.

va_dcl — declares the "type" of the variable argument list. Note that there is no semicolon after the "va_dcl".

va_list — is a "type" for declaring the variable pvar. Pvar is a variable which will be used to step through the argument list. One such variable must be declared.

va_start (pvar) — initializes pvar to the beginning of the argument list.

va_arg (pvar, type) — returns the next argument in the list pointed to by pvar. It will be a value of type type. Variables of different types may be mixed, but it is up to the called routine to determine their types, since this cannot be done at compile time.

va_end (pvar) — is used to finish up.

The list can be traversed multiple times, as long as each traversal starts with a 'va_start' and ends with 'va_end'.

While 'varargs' originated at Bell Labs, and is available with System V, it is not documented there. Instead, its use was popularized with the Berkeley versions of Unix (which do document it). In any case, you should be able to use the 'varargs' macros to portably write functions which take a variable number of arguments (like 'printf' does).

The current implementation of 'varargs' allows a maximum of ten arguments in the 'va_alist'.

Other Routines Not From Unix

The following routines are not routines found on Unix, but are supplied in "ciolib", since they are generally useful.

. **basename --- return the file name part of a path name**

. **dirname --- return all but the last part of a path name**

Calling Information:

```
char *basename (str)
char *str;

char *dirname (str)
char *str;
```

'Basename' returns a pointer to the last part of the SWT path name contained in 'str'. If there are no slashes in 'str', it returns 'str', otherwise it returns a pointer to somewhere in the middle of 'str'.

'Dirname' returns a pointer to the directory part of the SWT path name contained in 'str'. It copies 'str' into a private buffer (of length MAXPATH), and then replaces the final slash with a '\0'. If there are no slashes, it changes no characters in the buffer. In all cases, it returns the address of the buffer; the original 'str' is not modified.

The following example should clariy what these routines do:

```
basename ("path/file");   returns   "file"
dirname ("path/file");    returns   "path"
```

. **c$ctov --- convert C string to PL/I string**

Calling Information:

```
int c$ctov (dest, src)
int *dest;
char *src;
```

Converts the C string in 'src' to a PL/I varying string in 'dest'. (A PL/I string is an array of integers. The first element contains the number of characters in the string. The rest of the array contains the characters, packed two to a word.) Conversion terminates when a '\0' is encountered in 'src'. The function return is the number of characters converted to 'var'. Like other C string routines, no bounds checking is performed (see ctov(2) in the Software Tools Subsystem Reference Manual, though).

NOTE: This routine has been changed from the previous release of the C compiler.

. **c$vtoc --- convert PL/I string to C string**

Calling Information:

```
int c$vtoc (dest, src)
char *dest;
int *src;
```

Converts a PL/I varying string 'src' to a C string 'dest'. The function returns the number of characters copied into to 'str'. Again, no bounds checking is done (see vtoc(2) in the Software Tools Subsystem Reference Manual).

NOTE: This routine has been changed from the previous release of the C compiler.

Conversion

The Georgia Tech C compiler is based on the specifications contained in The C Programming Language by Kernighan and Ritchie. However the C compiler environment is not totally compatible with the Unix C implementation. Simulation of a Unix environment under Primos can be done only with an unreasonable loss of performance. Therefore, Unix C programs require some conversion to execute on Prime systems. (Programs that depend intimately upon the Unix process mechanism or the Unix file system layout are more difficult to convert. Likewise, programs that make heavy use of Unix inter-process 'signal' interfaces will be difficult to convert.)

C Program Checker

There exist the beginnings of a "C Program Checker" to flag possibly dangerous C program constructs when it encounters them; e.g. type mismatches. The "C Program Checker" can be called by using the "-y" option with 'cc', 'ccl', or 'ucc'. It currently reports on mismatched formal/actual parameters and misdeclared function return values.

Incompatabilities With PDP-11 C

The C compiler is compatible with PDP-11 C where possible. The following list enumerates those features of the Georgia Tech C compiler which are not compatible with PDP-11 C.

Include Statements

The compiler will complain about semicolons appearing at the end of include statements.

Note that the Georgia Tech C compiler automatically includes the standard definitions in "=cdefs=" so that the typical Unix-style "#include <stdio.h>" is optional. The compiler will search for an include file starting with the the current working directory, through the directories listed with the "-I" compiler option in the order listed, and ending with the system include directory "=incl=". Use of angle brackets (e.g., <filename>) rather than double quotes (e.g., "filename") in the include statement directs the compiler to skip the search of the current working directory.

Pointers

It is currently not possible to make pointers and **int**s the same length. Pointers are 32 bits, **int**s are 16 bits. The compiler tries to warn of pointer truncation, but cannot always detect it.

If NULL pointers are to be passed as arguments, they must be of type pointer (e.g. you cannot pass 0 or 0L as a NULL pointer. Use the symbolic constant NULL which is defined in "=incl=/stdio.h" to be "(char*) 0").

Pointers to dynamically linked objects cannot be compared. Pointers to dynamically linked objects (currently only functions are dynamically linked) are actually faulting pointers to character strings. At run time, these pointers are filled in with the correct linkage address (the links are "snapped") the first time the pointer is referenced indirectly. The C compiler must generate a constant pointer to each external object in each C object file. If relocatable files are linked together, during execution it is possible to have one file's constant pointer snapped, and the other's untouched. The object code generated by the compiler to compare these pointers does not reference through the pointers; it merely treats them as 32-bit integers. Because of this, comparisons of pointers to dynamically linked objects may give inconsistent results. A significant performance penalty would be required to guarantee consistent results in such a limited case.

Program and Data Object Size Restrictions

No source file may require more than 65536 words of static data. The static data for each C source file is compiled into a single linkage frame, and the linkage frame size restriction is imposed by the system architecture.

If you do require very large data objects, you may be able to get around this restriction with some work. You must declare the data object as an **extern** and write a Fortran subroutine that declares the data object name as a common block. Then when accessing the contents of this large block you must somehow insure that an object **never** crosses a segment boundary (start it at the beginning of the next segment just as Fortran does). If you attempt to address an object (such as a **double**) across a segment boundary, part of your reference simply wrap around to the beginning of the segment you are trying to reference beyond.

No source file may require more than 65536 words of procedure text. The compiler generates all procedures in the same PMA (Prime Macro Assembler) module. Currently PMA restricts the module size to 65536 words.

No function may generate more than 65536 words of internal-format PMA (currently around 16K statements). This is a code-generator workspace restriction. It has only been encountered

with output from YACC -- functions this huge are just not normally found around PDP-11s. (YACC is an LALR(1) parser generator. Its reads a BNF grammar, and produces a C function which will parse the grammar. This generated output has many large tables.)

Functions

In C, all arguments are passed by value. In Georgia Tech C, as long as arguments match in type they are, in all outward appearances, also passed by value. However, the internal mechanism for parameter passing is different from Unix C and will give different side effects if arguments do not match in type and in number.

The Prime architecture maintains a stack for local variables and provides a 64V mode procedure call argument transfer primitive for passing pointers, but not data values. We have used this mechanism to take advantage of its speed. Therefore, pointers are passed by value, just as in Unix C, but data values are not passed by value; a pointer to the data value is passed into the stack frame of the called procedure; the data value is then copied into the local stack frame by the procedure initialization code. This scheme is transparent as long as there are no type mismatches. For this reason, an attempt to cast a pointer argument to a non-pointer type will fail.

A variable number of arguments can be used, but not in the same manner as in Unix. The strategy is to declare as many arguments as you will ever need (make them pointers so that the compiler does not try to copy them). You will actually ignore all but the first of these names in the function. This trick forces the compiler to leave enough room for your arguments in the procedure's local stack frame. When the function is called, you will find the first argument pointer at the address of the first argument, the second argument pointer at the address of the first argument plus 3, the third at the address of the first argument plus 6, etc. Note that because of software conventions, i.e., the procedure initialization code, functions that are declared with zero arguments must be called with exactly zero arguments; and functions that are declared with one or more arguments must <u>not</u> be called with zero arguments.

Programs that depend on the order of parameter evaluation will fail.

You cannot call a function with single precision floating point arguments nor can you ever expect a function to return a single precision floating point argument. Remember, C turns them into double precision.

If a structure is to be a return value, the compiler adds on an additional first argument through which it passes a pointer to a temporary area in the calling procedure for the return value. Needless to say, type or length mismatches could cause

significant nastiness.

The side effects of type mismatches are quite predictable and can be useful for calling non-C procedures. For example, if you pass a non-pointer argument to a pointer argument, it will behave exactly as if a pointer had been passed (i.e. possibly allowing the supposed "value" argument to be modified). If you pass a pointer argument to a simple variable argument it behaves just like you passed the value of the argument instead.

Be wary of non-C routines which modify their arguments (particularly Subsystem routines like 'ctoi'); if you pass a constant, the "constant" might end up with a different value in it than it had before the routine was called!

Arrays

Although it is possible to index outside of array bounds, doing so is very dangerous. In 64V mode, indexed instructions are much faster than 32-bit pointer arithmetic. As a consequence, the compiler generates 16-bit indexed instructions wherever possible. The only side effect of this performance improvement is that indexing outside the bounds of arrays may not give the expected results.

Identifiers --- Naming Restrictions

Because the C compiler originally generated symbolic assembly language which was then processed by PMA, the Prime Macro Assembler, variable and function names had to follow PMA's naming conventions which require that names begin with an alphabetic character. To achieve the necessary compatibility, variable and function names beginning with an underscore are prefixed with "z$". Even though 'vcg' now generates object code directly, this naming restriction is still in effect.

Field names within **struct**s must be unique since the C compiler does not maintain a separate symbol table for each **struct**. This behavior is in accordance with K&R and the V7 Unix C compiler. (Berkeley Unix, System III, and System V, all keep separate symbol tables for each structure.)

Character Representation and Conversion

Character values run from 128-255, not 0-127.

Characters are not sign extended when promoted to integers.

Numerical

Programs that use data of type **double** may lose precision in trade for increased magnitude.

See the SWT Math Library User's Guide for more details on Prime's floating point hardware and software.

Library Incompatibilities

The Unix call 'fork' cannot be efficiently implemented because of operating system restrictions, and is therefore not available with Georgia Tech C.

'Read' and 'write' calls that do not use 'sizeof' to compute the buffer length will probably have to be changed.

Programs that open other users' terminals can not be supported.

Unix File System Incompatibilities

Programs that depend intimately on the Unix directory structure ('..', directory layout, links) will not be easily converted.

Programs that depend on the order and behavior of Unix file descriptors will not be easily converted.

You cannot depend on file descriptors 0, 1, and 2 always being connected to standard input, standard output, and standard error respectively. Instead, use the macros STDIN, STDOUT, and STDERR (defined in "=incl=/swt.h", which is automatically included by "=cdefs=").

Tabs

Tabs are not supported in exactly the same manner on the Prime as in Unix. C programs which produce tabs in their output should be run piping their output into the Subsystem program 'detab' ('detab +8' is recommended).

Static Initializers

Initializers for static data objects which involve the "address-of" operator may only consist of "&objectreference". For example, while the statement "static char *x = &A" is okay, the statement "static char *x = &A+1" cannot be handled by the Georgia Tech C compiler. The restriction arises from the inability of PMA/SEG to handle address expressions of external symbols when forming 32-bit pointers.

Registers

In 64V mode, the Prime is essentially a single accumulator machine. Thus, while the compiler recognizes the **register**

keyword, there is no effect on the size or speed of the generated code.

The Type void

Berkeley and System III Unix introduced the new type **void** into the C language. A **void** function is one which is guaranteed not to return a value (i.e. a true procedure). Only functions may be declared to be of type **void**, although you may also cast a function call to **void**. Georgia Tech C does not directly support **void**, but you may get around it with the simple statement:

```
#define void int
```

which should allow you to port practically any code which uses **void**. Admittedly, this defeats some of the type checking that the new type provides, but it will allow you to port code, without having to modify it.

Bugs

Known Bugs

The following is a list of known problems with the C compiler, as well as important enhancements that need to be made.

1. In certain instances, the compiler's attempt at parsing error correction fails to accept an input token. This can result in an infinite loop in the parser as it encounters and reports the same error repeatedly. A good example is placing an extra semicolon after the right brace of the statement in an **if**, before the corresponding **else**. The compiler will halt after reaching a limit of 50 such messages.

2. Bit fields must be initialized by execution time assignments; compile time initialization does not work correctly.

3. If "f" and "g" are type float, then "f*=g" is performed in single precision, whereas "f=f*g" is performed in double precision. We have not made a detailed analysis of the ramifications of this situation; it may be that no loss of precision can be detected. Regardless, because of the structure of the code generator, it will be very difficult to alter this situation.

4. The preprocessor does not support the "#if" construct.

5. The parameter-checking option ("-y") does not check calls to the C library.

6. A duplicate **case** in a **switch** is not detected by the first pass of the compiler. It will cause an error (with no information regarding the location of the error!) message to be reported by 'vcg'. In some cases, no error is reported, but 'vcg' generates unreasonable code.

7. There are several problems involving duplicated declarations for an external/global identifier (e.g. "extern a; int a;"). Most reasonable redeclarations are handled correctly, but some of the more obscure cases are probably not handled the way the Unix compilers handle them. In general, correct handling of these odd cases is not described explicitly--to find out how they "should" be handled, you have to ask a Unix compiler (and they often give different answers).

8. The sequence of declarations "extern int a[]; int a[5];" generates a warning message that "a" is being redeclared improperly. This is caused by the differing array bounds confusing the compiler into thinking that the second declaration is unreasonable. This is a definite bug (as

opposed to a question of interpretation).

9. The constant -2^-31 (smallest 32-bit negative number) is mishandled in all bases ('gctoi' goofs on it). For the time being, instead of using "0x80000000", use "(1<<31)" or "(~0x7FFFFFF)". These will give identical results because the constant folder gives correct results.

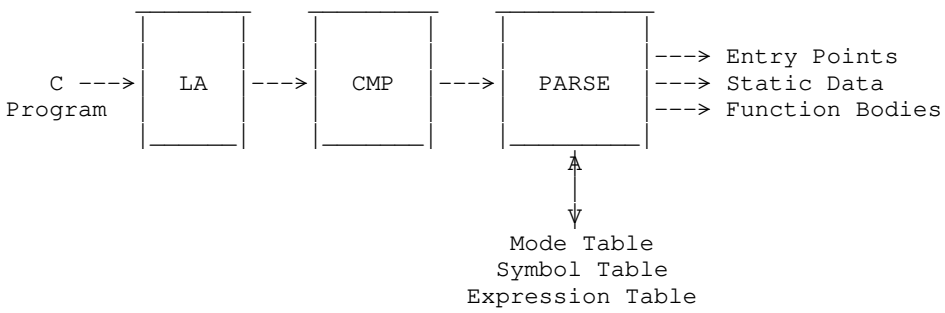
10. The construct "p++->x" confuses the compiler and causes it to complain about missing parentheses. This is because "->" is of higher precedence than "++" and thus confuses the recursive-descent parser. You should write the expression as "(p++)->x".

11. The new version of the code generator still has some bugs in it. If it produces an object file which causes an error from the loader, you may wish to compile the program with the "-s" option, to generate PMA in a ".s" file. Then use 'pmac' to assemble it, and load the new binary. This will usually work; it will simply take longer to compile.

Technical Information

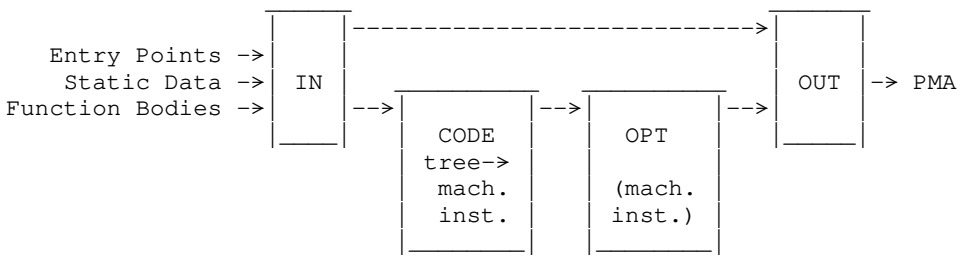
Implementation

The C compiler accepts C source code as input and in two "passes" produces 64V-mode relocatable object code as output. The first compilation pass is implemented by a Ratfor program known as the "front end," and the second pass by a Ratfor program known as the "back end." The front end is the Subsystem program 'c1' and the back end is 'vcg'.



The "Front End." LA is the lexical analyzer, or scanner. CMP is the C macro preprocessor. PARSE is the parser and intermediate code generator.

The front end is a classical recursive-descent compiler, employing a lexical analyzer (to break the stream of input characters into tokens), a preprocessor (to handle macro definitions and source file inclusion), and a parser (to analyze the program, diagnose syntactic and semantic errors, and produce an "intermediate form" output stream).



The "Back End." IN reads the intermediate code produced by the front end. CODE attempts to "intelligently" generate machine instructions. OPT performs some simple peephole optimizations to remove redundant loads and stores. OUT converts the internal instruction form to 64V-mode object code, or, optionally, to Prime Macro Assembly Language.

The back end is a reusable general-purpose code generator. It accepts the linearized intermediate form tree produced by the front end, rebuilds the tree internally, converts the tree to a linked list of machine instruction representations, performs peephole optimizations on that list, and then produces 64V-mode object code, ready for link editing and subsequent execution. 'Vcg' has an option for producing symbolic assembly language instead of object code. The assembly language that 'vcg' produces is suitable for processing by the Prime Macro Assembler, PMA.

For those of you wishing to supply your own front-ends to the code generator, there is a V-mode Code Generator User's Guide (use the Subsystem command 'guide') and a Reference Manual entry for 'vcg'.

Performance

The C compiler requires parts of 5 segments to run. The previous version of the C compiler, which used to call PMA, ran almost twice as fast as Prime's Fortran 77 and Pascal compilers (700 lines per minute vs 400 lines/minute on a PRIME 550 running under Primos 18.1). Hand inspection and informal benchmarks indicate that the code produced is superior to that produced by Pascal, PL/1 and Fortran 77; in particular, fewer base register loads are generated, and operations on packed data structures are performed without resorting to the field manipulation instructions.

The compile time requirements for each phase were approximately as follows: 'c1': 23%, 'vcg': 27%, 'pma': 50%. Roughly half of 'vcg's time was spent in the assembly-language output routines.

For the second release, 'vcg' has been changed to produce 64V-mode object modules directly. This substantially reduces compile time. We have not measured the new version of the compiler, but the compile time requirements for each phase are about equal. The total compile time is now approximately half of what it was, since PMA is not involved in the process.

Subsystem Managers Section

The machine-readable text of the User's Guide for the Georgia Tech C Compiler is in the file "=doc=/fguide/cc" (already formatted) and in the directory "=doc=/guide/cc" (unformatted) (assuming that you have already installed the C compiler according to the directions below).

Installation Procedure

The C compiler and its support programs are intended to be part of the Subsystem. Source, documentation and executable versions "drop in" to appropriate Subsystem directories so that they are accessible as standard Subsystem tools. This section covers the procedures necessary for installation of the C compiler.

Georgia Tech C Installation Package

The C Installation Package as sent from Georgia Tech contains the following items:

1 Release Tape
1 Copy of the C User's Guide

Release Tape Contents

The C Release Tape contains all files and directories necessary for proper operation of the C compiler under the Software Tools Subsystem. It is in standard MAGSAV/MAGRST format and contains 5 "logical tapes." Each logical tape contains a number of files that "drop in" to Subsystem directories.

Logical Tape 1

The first logical tape contains executable files that are to be placed in "=bin=",

cc ccl compile ucc vcg vcgdump

'Cc' is the Subsystem C compiler, 'ccl' is a shell file that compiles and loads a C program, 'ucc' is a 'Unix-like' C compiler and 'vcg' is the V-mode code generator. 'Vcg' is used by the C

compiler but can also be used separately by those users who have their own "front ends." 'Vcgdump' reads the intermediate files produced by 'c1', and prints a human-readable version of the intermediate form tree. 'Compile' is a general purpose compiler interlude.

Logical Tape 2

The second logical tape contains libraries for "=lib=",

```
ciolib   c$main   nciolib   vcglib   vcg_main
```

"Ciolib" contains the executable version of the C run time library; "c$main" is a small startup program that must be loaded with every C main program. "Nciolib" is the version of the C run time library for programs which are to run under bare Primos.

'Vcglib' is a library of regular and shortcall routines for range testing and other purposes. 'Vcg' generates calls to these routines for their operations, instead of generating code. 'Vcg_main' is a small general purpose start off routine. These are not used by C programs, but are necessary if you wish to provide your own "front end" for 'vcg'.

Logical Tape 3

The third logical tape contains files for "=extra=",

```
bin/(c1   cc   cck1   cck2   compile   ucc)

incl/(swt_def.c.i    ascii.h    assert.h    ctype.h
      debug.h        lib_def.h  keys.h      math.h
      memory.h       setjmp.h   stdio.h     swt.h
      swt_com.h      varargs.h)

incl/(vcg_defs.h   vcg_defs.p.i   vcg_defs.r.i)
```

'C1' is the "front end" for the C compiler and is called by 'cc', 'ccl', and 'ucc'. 'Compile' is a general purpose compiler interlude. 'Ucc' calls it. 'Cck1' and 'cck2' are the "trouble spotters" for C programs. They will flag potentially dangerous constructs in a C program and are invoked by compiling a program with "ucc -y". Subsystem definitions for the C compiler are contained in "swt_def.c.i". The ?*.h files are other header files, discussed above in the chapter on the compile time environment.

The vcg_defs.?* files contain constant definitions for use in writing "front ends" for 'vcg'.

Logical Tape 4

The fourth logical tape contains documentation for "=doc=",

```
man/s1/(cc.d    ccl.d       ucc.d
        vcg.d   vcgdump.d   compile.d)

man/s5/(c1.d   cck1.d   cck2.d)

fman/s1/(cc.d    ccl.d       ucc.d
         vcg.d   vcgdump.d   compile.d)

fman/s5/(c1.d   cck1.d   cck2.d)

guide/(cc vcg)

fguide/(cc vcg)
```

Logical Tape 5

The fifth logical tape contains source files for "=src=",

```
std.sh/(cc.sh   ccl.sh   ucc.sh   compile.sh)

ext.c   # new directory with source files for C interludes

ext.r/(cck1.r   cck2.r   cck2_com.r.i   cck2_def.r.i)

lib/(cio   c$main    nc$main   vcg    vcg_main)

spc/(c1.u   vcg.u)

std.r/(vcgdump.r   vcgdump_com.r.i)
```

If you do not have a source license then you will not recieve any of the source files. In fact, the "src" directory will not be on the tape.

Loading the Tape

To load the release tape, follow the instructions below:

1. Assign a tape drive:

   ASSIGN MT0

2. Mount the release tape on the assigned drive.

3. Attach to directory "=bin=":

   ATTACH BIN <owner-password>

or if the tape is being restored to an ACL or priority ACL protected partition, type

```
ATTACH BIN
```

4. Load the contents of the first logical tape with MAGRST:

```
MAGRST
Tape Unit (9 Trk): 0
Enter logical tape number: 1
<tape label information>
Ready to Restore: yes
```

(This loads the files "cc", "ccl", "ucc", "compile", "vcg", and "vcgdump".)

5. Attach to directory "=lib=":

6. Load the contents of the next logical tape (i.e., reply "0" to the "Enter logical tape number:" prompt) with MAGRST. (This loads the library files for 'cc' and 'vcg'.)

7. Attach to directory "=extra=":

8. Load the contents of the next logical tape with MAGRST. (This loads the support programs for the compiler interfaces.)

9. Attach to directory "=doc=":

10. Load the contents of the next logical tape with MAGRST. (This loads the formatted and unformatted 'vcg' and C compiler guides, and the formatted and unformatted Reference Manual (help) entries.)

11. Attach to directory "=src=":

12. Load the contents of the next logical tape with MAGRST. (This loads the source code for the C compiler, the run_time library, the compiler interfaces, the V-mode code generator, and the vcg support routines.) If you do not have a source license, and/or you have received a demonstration tape, this logical tape will not be present.

This completes the loading of the C compiler from tape.

Installation

Once you have loaded the tape, the C compiler is ready to use. However, for the C compiler programs to appear in the "help" index, you must rebuild it by executing 'man_index' in "=doc=/build".

Finally, for the 'locate' and 'source' commands to work correctly, you have to rebuild the =srcloc= file. To do this, 'cd' to "=src=/misc", and execute the file "make_srcloc". This completes the integration of the C compiler with the rest of the Subsystem.