C

ACORN SCIENTIFIC

REFERENCE MANUAL

C

ACORN SCIENTIFIC

PART NO 0410, 007
ISSUE NO 1
JULY 1985

In case of difficulty please contact your supplier. Deficiencies in software and documentation should be notified in writing, using the Acorn Scientific Fault Report Form to the following address:

Sales Department
Scientific Division
Acorn Computers Ltd
Fulbourn Road
Cherry Hinton
Cambridge
CB1 4JN

Published by Acorn Computers Limited,
Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.

Within this publication the term BBC is used as an abbreviation for the British Broadcasting Corporation.

NOTE: A User Registration Card is supplied with the hardware. It is in your interest to complete and return the card. Please notify Acorn Scientific at the above address if this card is missing.

ISBN 0 907876 40 4 Acorn Scientific

Contents

1 Introduction

32000 C refers to the implementation of C on Acorn Cambridge Series computers, under the Panos operating system. Note that this manual is not a tutorial; reference to such material may be found in Appendix B.

1.1 Standard C

The definition of 32000 C follows that of Kernighan and Ritchie which is described in *The C Programming Language.* Exceptions are noted in chapter 2 below. Throughout this manual, Kernighan and Ritchie's definition is called standard C , although it is not a formal national or international standard. Because most other implementations of C are also based on Kernighan and Ritchie's definition it is possible to move C programs quite freely between different computers, provided that extensions to the standard peculiar to individual machines are avoided. Although much of the power of C comes from the library routines for input and output of data, string handling etc. which are supplied along with most çompilers, the standard does not define a set of routines which all compilers must provide. However, most compilers agree about the definitions of the commonly used routines. The library routines supplied with 32000 C (see Chapter 4), with the exception of the low-level I/O routines, are common to almost all implementations of C; it is not guaranteed however that these routines will have exactly the same effect as their counterparts in other versions of C.

1.2 Installation

The installation procedure is described in the User Guide supplied with the system. C is provided on a DFS format disc, and must still be installed even if it is going to be used on this medium.

2 Using the compiler

Having created the source using an editor, giving it the file extension '-c', the command

```
cc <program name>
```

will compile the program. The compiler automatically searches for files with the '-c' extension, and it does not need to be quoted in the command.

2.1 Compiler Options

A number of compiler options are available which allow greater control over the input and output of the compilation, and over various debugging tools.

The overall argument string is:

```
{-source} name {{-list {name}} {{-aof {name}} {-module name}
{-decode} {-nodiags}  {-check} {-error name}  {-show}  {-identify} {-help}}
```

where name refers to any file name, and braces enclose optional items. Note that some of the file names will contain extensions. File extensions are an important Panos feature. The extensions '-lis' and '-aof' are automatically appended by the compiler for list files and Acorn Object Format (aof) files respectively. See the *Panos Guide to Operations* for further details about file extensions.

-source
: This argument is optional and does not need to be quoted when giving the source name. The source program must always have the file extension '-c' appended to the file name.

-list
: This option generates a listing which is sent to the file name specified. The file extension '-lis' is automatically appended. See figure 2 (section 5.1) for an example of an erroneous program compiled from within the editor, where the listing is sent to another file, also loaded into the editor.

-aof

Normally, the Acorn Object Format file which results from the compilation is given the same name as the source, with the extension '-aof' instead of the extension '-c'. This option allows the user to specify a different name. The '-aof' extension is added by the compiler.

-module

The module name field of the generated aof file is normally set to be the root name of the corresponding source program file (with the '-c' extension removed). Use -module<name> to override this default and set the module name field in the aof file to <name>.

-decode

If this option is used, the aof file generated by the compiler will contain extra information which allows the -decode command to produce a listing showing the machine code for each source statement in the program. To interpret this, a program will need to be written.

-nodiags

The code generated by the compiler contains some diagnostic information which allows a backtrace to be produced if an error occurs at run-time. This option disables diagnostic tables. The compiled code is smaller, but error messages are less helpful.

-check

This option causes the compiler to detect when a variable is used before a value has been assigned to it.

-error

This option allows the user to specify a file to which compiler errors are sent. By default, these are sent to the screen.

-show

If this option is used, the listing file will contain the expansion of any macro calls in the source program as well as the original source text. Macro expansion lines are marked by a double quote mark following the line number.

Examples of compiler commands

A. The minimal command

```
cc CProg
```

The source program ‘CProg-c’ is compiled with all default options: the object file is called ‘CProg-aof’, and it contains some diagnostic information, but no machine code listing for each source statement; no listing is generated; no checks are made for unassigned variables; errors are sent to the screen.

B. The aof file is specified, and diagnostic tables are disabled.

```
cc -source dfs::3.CProg -aof dfs::1.AFile -nodiags
```

The optional ‘-source’ argument is used to specify ‘dfs::3.CProg-c’; the object file is to be called ‘dfs::1.AFile-aof’, and is to contain no diagnostic information.

C. A listing of the compilation is specified.

```
cc nfs:$.CDir.Cprog -list adfs:$.Cprgl
```

The program ‘nfs:$.CDir.Cprog-c’ is compiled, and a listing placed in the file ‘adfs:$.Cprgl-lis’.

2.2 Panos Global Variables

To run C, a number of Panos global string variables must be setup. These are:

`cc$fe`	`C compiler front end`
`ll$be`	`C compiler back end`
`C$include`	`#include files`
`Link$Lib:C`	`C run-time library`
`Link$Lib:Pas`	`Pascal run-time library`
`LL$Prim`	`Primitive code procedures`

These variables must contain the full pathnames of the various files. See the *Panos Guide to Operations* for a description of global string variables.

3 Differences from Standard C

The differences between 32000 C and standard C are described here. Section numbers in the text refer to the section numbers in the *C Reference Manual* (Appendix A of *The C Programming Language*).

3.1 Restrictions

These features of standard C are not permitted in 32000 C.

3.1.1 Loose type checking of . and -> operators

Section 7.1 of the standard states that the left operand of the operators `.` and `->` must be a structure and the right must be the name of a member of that structure. Section 14.1 states however that the compiler allows any value as the left operand of `.` and any expression of pointer or integer type as the left operand of `->`. 32000 C follows the rule of section 7.1 in disallowing examples like the following:

```
int i;
struct { int p, q; } s; /* THIS IS ILLEGAL */

i.p = 0;
i->q = 17;
```

3.1.2 White space within compound operators

In 32000 C, assignment operators like += are single tokens whose parts (+ and =) may not be separated by white space. If + = is written instead of +=, the compiler will produce an error message.

3.1.3 Use of sizeof in array declarations

Constant expressions used in an array declaration may not contain the `sizeof` operator. This example is ILLEGAL: `char v [ sizeof(int) ] ;`

3.1.4 #line Ignored

The `#line` compiler control line is accepted but ignored by the 32000 C compiler.

3.1.5 Anachronisms not allowed

Both of the anachronistic forms '=op' and 'int x 3;' described in section 17 of the standard are illegal in 32000 C.

For `x=-1;` write either `x-= 1;` or `x = -1;` depending on which is meant.

For `int x 3 ;` you must write `int x = 3 ;`

3.2 Extensions

The following non-standard language features are allowed in 32000 C.

3.2.1 Dollar sign in identifiers

32000 C allows the dollar sign `$` to appear in identifiers. The dollar sign is treated as another letter. The following are all acceptable identifiers:

```
$
rate$
$_max9
```

3.2.2 More significant characters in identifiers

Two identifiers are deemed by the compiler to be the same if their first 31 characters match (standard C says 8 characters). Any additional characters are ignored. This rule also applies, while a file of routines is being compiled, to external identifiers, but the rules used to decide whether two external identifiers match when routines compiled separately are linked together depend on the linker program, not the compiler. Most linkers will treat two identifiers as being identical if their first 6 or 7 characters are the same, ignoring upper/lower case distinctions.

If C programs must be portable to many different compilers, they should only use identifiers which are distinct in the first 8 characters, except for external identifiers which should be distinct in the first 6 characters whether or not the distinction between upper and lower case letters is ignored.

3.2.3 Assignment to whole struct/union variables

In standard C, all that can be done with a `struct` variable is to create a pointer to it (using the & operator) or access one of its members (using the `.` operator).

32000 C allows you to copy all of the members of a `struct` variable at once by using the assignment operator, =. If one operand of = is a `struct` then the other must be a `struct` of the same type.

For example:

```
struct { int p, q; } x, y ;

x.p = 3; x.q = 17;

y = x; /* struct assignment */
```

After this structure assignment, y.p has the value 3 and y.q has the value 17.

Both assignments in the example below are ILLEGAL because the types of the operands for = do not match.

```
struct { int p, q; } x ;
struct { int a, b; } y ;
int i;

x = i;      /* ERROR one integer, one struct ERROR */
x = y;      /* ERROR same size, but different types ERROR */
```

32000 C also allows function arguments to be `struct` types (standard C allows only pointers to `struct` as arguments). `Struct` arguments are declared and used in the same way as any other type.

For example:

```
struct tag { int p, q; }

clear(x)
struct tag x;
{
         x.p = 0; x.q = 0;
}

example()
{
         struct tag a;

         a.p = 3; a.q = 4;
         clear(a);
         return( a.p + a.q );
}
```

The result returned by the function `example` will be 7 because, like all other types of function argument in C, `struct` arguments are passed by value: `clear` cannot affect the contents of the structure `a` which is passed to it, since it works with a copy of `a` named `x`.

3.2.4 Long escape sequences

Standard C allows octal escape sequences of the form \ddd within string and character constants. 32000 C does not restrict the length of the digit sequence after the \ character to three digits, but programs which must be usable with other C compilers should observe the restriction.

3.2.5 Restrictions on struct member names

In standard C, the same member name may occur in different structures only if the fields identified by the member name and all preceding fields are the same. 32000 C makes no restrictions on the use of the same member name in different structures. Again, programs which must be usable with other C compilers should not make use of this fact.

3.2.6 Type-name syntax relaxed

Kernighan and Ritchie give the definition of the 'type-name' construct as

```
type-name:
type-specifier abstract-declarator
```

This allows only one 'type-specifier' before the 'abstract-declarator', disallowing expressions like:

```
sizeof(long int)
(unsigned short) e
```

Multiple 'type-specifier's like 'long int' are allowed in this context by other implementations of C, and by 32000 C.

3.2.7 Data Type void

The 32000 C compiler has an extra data type `void` which is not part of standard C. `Void` is a special data type with no values, used to indicate that a function returns no value. Expressions can be cast to type `void` in order to discard their value explicitly. The (non-existent) value of a `void` expression may not be used in any way, and neither explicit nor implicit conversations may be applied to such a value. Therefore a `void` expression may be used only as an expression statement, or as the operand of a comma operator.

3.2.8 Forward References to structure tags

Forward references to structure tags are allowed, e.g:

```
struct t1 (int i; struct t2 *p);
struct t2 (char x, y;);
```

Using a structure tag (like t2) before it has been declared is only allowed when pointers to the structure are being declared or manipulated: objects of that structure can only be declared after the structure has been fully declared. Similarly `sizeof, ->` and . will not work until the complete structure declaration is given.

3.3 System-dependent features

3.3.1 Data Type enum not allowed

32000 C does not allow the data type `enum`, which is permitted in some C compilers. The `enum` data type allows the programmer to construct new data types by enumerating the values which variables of that type may take. This restriction is permitted by standard C.

3.3.2 Pointers to Functions

Forward references to static functions are allowed. Static functions are called as external functions, but are defined to the linker as local rather than global symbols, so their names are not exported to other modules.

3.3.3 Standard locations for #include files

If the Panos global string variable `C$include` is defined, its value is taken as a list of directories to be searched for #include files, separated by commas or white space. If the form `#include "file name"` is used instead of `#include <file name>`, the current directory is searched before the `C$include` list. For example:

```
    -> set var C$include "$.Panoslib, $.CUser. Fred.CInc"
```

See the *Panos Guide to Operations* for more details about global string variables.

4 The C Runtime Library

4.1 The Purpose of the runtime library

The 32000 C runtime library is a collection of compiled functions which perform commonly-used operations not included in the C language itself: reading and writing data, and evaluation of mathematical functions like `sin` and `cos` are the most obvious instances.

This chapter covers the conventions used to describe the arguments of library routines, lists the available routines grouped by function (I/O, string handling etc.) in section 4.3, and then lists the available routines in alphabetic order, giving a description of the effects of each in section 4.4.

4.2 Conventions

This section describes how to use standard header files in calling library routines and how to interpret the notation used in section 4.4 to specify the number and types of arguments they require.

Runtime library functions are used in exactly the same way as user-defined functions (most are in fact just normal C functions). To use a library function, a program must first declare the name of the function to be used, and indicate that it is external to the program (storage class `extern`).

So that the declarations of library functions in user programs are always correct, standardised header files are provided with the system for each group of library functions. The programmer uses the C `#include` statement to access the contents of the header file before making use of any of the functions declared there.

`#include` files are located by declaring a Panos global string variable `C$include` to contain references to these files. The value of the global, variable is a list of directories to be searched for `#include` files. The directories are separated by commas or white space. If the form `#include "file name"` is used instead of `#include <file name>`, the current directory is searched before the `C$include` list. For example:

```
-> set var C$include "$.PanosLib, $.CUser. Fred.CInc"
```

See the *Panos Guide to Operations* for more details about global string variables. As well as containing the required function declarations, the header file will include declarations for any special data types required by its functions. For example, consider the standard input/output functions. These are declared in the header file `stdio-h`. Before the first use of any of the standard I/O functions, a program must contain the statement `#include <stdio-h>`. For compatibility with other C compilers, `#include <xxx.h>` is taken as equivalent to `#include <xxx-h>`.

This declares all of the standard I/O functions like `printf` and `getc` as well as defining the macros EOF and NULL which are used in communication between the I/O functions and user programs. EOF has the value -1; NULL has the value 0.

Programs should always use the header files provided with the compiler rather than attempting to provide their own declarations for library functions since the declarations of some functions will differ from the obvious declaration implied by the function synopses in this chapter.

These function synopses indicate how to call library functions. Information about required argument types and function result types is presented in the form of a C function declaration prefixed by `#include` statements which indicate which header files, if any, must be used in order to access the function. For example, the synopsis for the `fgets` function looks like this:

```
#include <stdio-h>

char *fgets(s, n, iop)
char *s;
int n;
register FILE *iop;
```

This means that `fgets` returns a result of type `(char *)` and has three arguments of types `(char *)`, `(int)` and `(FILE *)`, where FILE is a data type declared in the header file `stdio-h`. This header file must be included in all programs which use the function.

Ellipsis is used in function synopses to indicate that a function has a variable number of arguments, for example the `printf` function:

```
#include <stdio-h>

printf(format[,arg1[,arg2[,...]]])
char *format;
```

The synopsis shows that `printf` 's first argument must be a character pointer. The square brackets [ ] indicate that the enclosed arguments are optional; ellipsis "..." indicates repetition. Where argument types are not shown in the synopsis (e.g. arg1,arg2,... for `printf`) the allowed argument types are discussed in the text.

4.3 Library Modules

This section lists the library routines provided, divided into the following functional groups.

Stream input/output to files and devices, including facilities for random file access

Classification of ASCII characters (e.g. is a character an ASCII letter?)

String manipulation, including string copy and string comparison

Character conversion (e.g. convert uppercase letters to lowercase)

Numeric conversions between ASCII string and binary representations for integer and floating-point values

Mathematical functions, including logarithms and trigonometric functions

Dynamic ('heap') memory allocation and deallocation

Various other miscellaneous functions

Some background information common to all of the functions in a group is presented in this section rather than being repeated with the description of each individual function. In particular, the concepts on which the standard I/O system is based are presented here, such as `stream, file, pointer,` etc.

4.3.1 Input/Output Routines

The routines which are provided to read and write data fall into two groups: the low-level I/O routines and the standard I/O routines.

4.3.1.1 Standard I/O

The standard I/O functions provide a portable I/O interface for C programs. The standard I/O functions are available in the form described here in most implementations of C. The standard I/O functions also provide buffering between user programs and files or devices. This means that I/O transfers to/from files remain efficient even if data is transferred between the file and the user program in small units (e.g. one byte at a time). On output, user data are placed in a data buffer allocated 'behind the scenes' by the standard I/O routines, until the buffer becomes full, at which point the contents of the buffer are written en masse to the file, achieving an overall speed-up because disc devices are optimised for block transfers. The situation for input is similar.

Other standard I/O routines allow random file access and conversion of numeric data between internal (binary) and external (character string) representations.

All of the routines described in this section require the calling program to include the header file `stdio-h` before they may be called.

Before a user of the standard I/O package can read or write the data in a file, a path to the file must be opened by calling the `fopen` function. The name of the file is passed to `fopen`, which, if the file is accessible, returns a pointer to a structure of type FILE. This file pointer must be used by the calling program to refer to the file in subsequent I/O operations (`fputc`, for example, requires a file pointer argument to identify the file which is to be written). The data type FILE is declared in the header file `stdio-h`.

After performing I/O on an open file, the path to the file may be broken by closing the file. Files should be closed when they are no longer in use, since there is a limit on the number of files which may be open at once. The precise number of open files allowed depends upon which filing system you are using: the DFS allows 5 files, the ADFS allows 10, and the NFS permits 5 files to be open per file server. Files may, of course, be opened again after they have been closed. Having more than one path open to the same file at any point in a program should be avoided. Closing all files explicitly at the end of a program is, however, unnecessary; this is done automatically by the standard I/O system.

In this example, a file named fred is opened, some ASCII data is written out to it and the file is closed. For clarity, no error checking is performed.

```
#include <stdio-h>          /* standard I/O declarations */

main()
{
    FILE *fp;               /* file pointer variable */

    fp=fopen("fred",        /* file name */
      "w");                 /* open for writing */

    fprintf(                /* formatted output routine */
     fp,                    /* file pointer (identifies file) */
     "Hi!\n"              /* text string to be written */
     );

    fclose(fp);             /* disconnect file */
}
```

For convenience, three file pointers are always automatically opened. These are declared in `stdio-h` as follows:
`FILE *stdin;` this is the standard input stream. By default on most systems, stdin is connected to a terminal keyboard.
`FILE *stdout;` this is the standard output stream. `stdout` on most systems is the display device (VDU or printer) of a terminal.
`FILE *stderr;` this is the standard error stream, used by programs for outputting error messages. It too is normally opened on the terminal output device.

To simplify writing programs which simply read one sequential input file, process it and write another sequential output file, `stdin`, `stdout`, and `stderr` can be redirected by the use of keywords on the command line which invoke the program, e.g.

```
prog
```

uses default input/output streams.

```
Prog -input file1
```

initialises `stdin` to be connected to `file1` rather than the keyboard. The -output and -error keywords may be used in a similar fashion, and in any combination. This means that programs may be written and tested using the terminal for standard input and output, then run unchanged using files for input and output, yet the program itself need not open files.

Stream I/O

The model of I/O supported by the standard I/O package is known as stream I/O. In the stream I/O model, a file is considered as a sequence of char values. A notional file pointer, maintained by the I/O routines, indicates the character position within the file at which the next character will be read or written. The file pointer is advanced automatically as characters are read or written. Random file access is supported by allowing user positioning of the file pointer.

The basic operations provided by the standard I/O package in support of the stream I/O model are therefore 'read a character' (fgetc), 'write a character' (fputc), 'reposition file pointer' (fseek) and 'read file pointer' (ftell). Other, higher level, operations (e.g. write a string) are built up directly from these primitive operations. Because of this, calls on the character level functions and the higher level functions may be freely intermixed and characters will still be transferred in the expected order.

Devices such as terminals are included in the stream I/O model: characters may be read or written from them as appropriate (in principle, one at a time) but positioning operations are not supported.

Binary I/O

The basic units in the above discussion of stream I/O are 'characters': values of type char. These are integers which stand for graphic character representations in the encoding scheme of the host computer system (e.g. the ASCII encoding for A is 65). The C I/O system, however, does not require that the values transferred be valid character representations. In fact, any binary value which can be represented in a char variable may be written to a file (and later read back unaltered). For example, on most implementations of C, any value in the range 0..255 will fit in a char. Arbitrary binary data can be stored in files using the standard I/O system by recording it as sequences of char values.

Text I/O

Text I/O in C is simply a special case of the binary I/O discussed above where the values transferred are restricted to the valid character codes for the host system.

Human-readable text files are divided into lines. Line-breaks are represented in the stream I/O model by the newline character, '\n' (ASCII code 10). On output, newline characters may be included at arbitrary points in the text. On input, programs detect the end of a line by comparing characters being read with the value '\n'.

Standard I/O Functions

The library routines which form the standard I/O package are listed in this section. Functions implemented as macros are marked with a *.

	clearerr	resets the error and end of file indicators
	fclose	closes a file
	fdopen	creates a FILE structure and associates it with a file descriptor
*	feof	tests for end-of-file
*	ferror	returns a nonzero integer if an error occurs during read or write operations
	fflush	writes out any buffered information to the file
	fgetc	returns the next character from a file; generates a true function call
	fgets	reads a line from a file; the line is terminated by a NUL character
*	fileno	returns the low-level I/O file descriptor
	fopen	opens a file
	fprintf	performs formatted output to a specified file
	fputc	writes a single character to a file; generates a true function call

	fputs	writes a string to a file
	fread	reads a specified number of items from the file
	freopen	reassigns the address of a FILE structure and reopens the file
	fscanf	performs formatted input from a file
	fseek	places the file pointer at a specified byte offset relative to the beginning of the file, the end of the file or the current location in the file
	ftell	returns the current byte offset from the beginning of the file to the current location within the file
	fwrite	writes the specified number of items to a file
*	getc	returns the next character from a file; implemented as a macro
*	getchar	returns the next character from the standard input device
	gets	reads a line from the standard input device; the newline is replaced with a NUL character
	printf	performs formatted output to the standard output device
*	putc	writes a single character to a file; implemented as a macro
*	putchar	writes a single character to the standard output device
	puts	writes a string to the standard output device; terminates the string with a newline
	putw	writes a specified integer to a file
	rewind	resets the file pointer to the beginning of the file
	scanf	performs formatted input from the standard input device
	setbuf	associates a buffer with an input or output file
	sscanf	performs formatted input from memory
	sprintf	performs formatted output to a character string in memory
	ungetc	writes a character to a file buffer and leaves the file positioned before the character

4.3.1.2 Low-Level I/O

The low-level I/O functions transfer 'raw' user data to or from files or devices in variable length blocks (down to one byte). The low-level I/O routines are provided mainly for compatibility with other implementations of C; normally standard I/O should be used. In low-level I/O files are accessed via 'file descriptors', small integers returned by the system when a file is opened. Other functions are provided to create new files and directly control the position in a file where data transfers will take place. Standard I/O is implemented in terms of the low-level I/O functions (which in turn, call the Panos I/O library). The values returned by the standard I/O macro `fileno()` are low-level I/O file descriptors.

The low-level I/O routines are:

close	closes a file
creat	creates a new file
isatty	determines if a file descriptor is associated with a terminal
lseek	places you at a byte offset within a file and returns the new position as an integer
open	opens a file for reading, writing or both
read	reads a specified number of bytes from a file and places them in a buffer
write	writes a number of bytes from a buffer to a file

4.3.2 Mathematical Functions

The mathematical routines calculate various standard mathematical functions such as logarithms, sines, cosines etc.

The trigonometric functions operate on angles expressed in radians.

Errors are handled by returning impossible or unusual result values and setting an error code in the external integer variable errno.

abs	returns the absolute value of the integer argument
atan	returns a value in the range -pi/2 to pi/2 which is the arc tangent of the radian argument

ceil	returns the smallest value which is equal to or greater than the argument
cos	returns the cosine of the radian argument
exp	returns the base e raised to the power of the argument
fabs	returns the absolute value of the floating point argument
floor	returns the largest integer which is less than or equal to the argument
log	returns the natural logarithm of the argument
rand	returns pseudorandom numbers
sin	returns a value that is the sine of the radian argument
sqrt	returns the square root of the argument
srand	reinitialises the random number generator

4.3.3 String Handling

The C language itself allows the manipulation of single characters. These library routines allow C programs to process variable-length strings of characters.

strcat	concatenates two strings
strcmp	performs lexicographic comparison of two ASCII strings
strcpy	copies one string to another
strlen	returns the length of a string
strncat	concatenates two strings up to a maximum number of characters
strncmp	performs lexicographic comparison of two ASCII strings (up to a maximum number of characters)
strncpy	copies a maximum number of characters from one string to another

4.3.4 Character Classification

The character classification functions described here are implemented as macros. They return a nonzero value if their argument meets the condition being tested and zero otherwise. The argument is a single integer.

isalnum	determines if the argument is alpha-numeric
isalpha	determines if the argument is alphabetic
isascii	determines if the argument is an ASCII character
iscntrl	determines if the argument is an ASCII control character
isdigit	determines if the argument is a digit
islower	determines if the argument is a lowercase letter
isprint	determines if the argument is a printing character
ispunct	determines if the argument is a punctuation character
isspace	determines if the argument is a space, horizontal or vertical tab, carriage return, form-feed or newline
isupper	determines if the argument is an uppercase letter

4.3.5 Conversions

These routines provide conversion operations between various representations of numeric values: binary integers, binary floating-point and character string. Some character mapping routines (upper/lower case mapping) are also provided.

atof	converts an ASCII string to a numeric value (double)
atoi	converts an ASCII string to a numeric value (int)
atol	converts an ASCII string to a numeric value (long)
tolower	converts uppercase characters to lowercase;
_tolower	returns lowercase characters unchanged
toupper	converts lowercase characters to uppercase;
_toupper	returns uppercase characters unchanged

4.3.6 Dynamic Memory Allocation

Building complex dynamically changing data structures requires a different class of storage from static or extern variables (which must be preallocated by the programmer when a program is written and are therefore not flexible enough) and auto or register variables (which disappear when the procedure which created them returns; some dynamic data structures must be operated on by many procedures).

This extra storage class is generally referred to as heap storage. In C, heap storage is allocated by calling a library function (malloc) and remains until it is explicitly released by calling another function (free).

calloc	allocates and clears an area of memory
free	deallocates the space allocated by malloc or realloc
malloc	allocates the specified number of contiguous bytes of memory
realloc	changes the size of an area previously allocated by malloc or realloc

4.3.7 Miscellaneous

Other useful library routines are provided for halting program execution, non local jumps, debugging etc.

assert	program debugging routine
_exit	stop program exit
getenv	access environment variables
longjmp	returns to the context saved by setjmp
perror	writes (to stderr) the most recent error encountered
rand	pseudo-random number generator
setjmp	saves the context of the calling function for a subsequent longjmp call
srand	change seed for rand
system	do operating system command string

4.4 Alphabetic List of Functions

This section lists all of the supported library functions supplied with 32000 C.

`abs` - integer absolute value

```
#include <math-h>

abs(i)
```

`abs` returns the absolute value of its integer operand.

`assert` - program debugging routine

```
#include <assert-h>

assert(expression)
int expression;
```

If the macro identifier NDEBUG is defined at the point in the source file where <assert-h> is included, use of the `assert` function will have no effect.

The `assert` function puts diagnostics into programs. The expression argument is any scalar expression. When it is executed, if expression is false (that is, evaluates to zero), assert writes the message "assertion failed" on the standard error file and does a diagnostic backtrace of the call stack. `assert` has no effect if its argument is true (non-zero).

No value is returned by `assert`.

`atan` - arc tangent

```
#include <math-h>

double atan(x)
double x;
```

`atan` returns the arc tangent of `x`.

`atof` - convert string to floating point

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

The string pointed to by `nptr` is converted to double-precision floating point representation. The first unrecognised character terminates the string.

`atof` recognises an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e` or `E` followed by an optionally signed integer.

`atoi` - convert string to integer

```
atoi(nptr)
char *nptr;
```

This function converts the string pointed to by `nptr` to integer representation. The first unrecognised character ends the string.

`atoi` recognises an optional string of tabs and spaces, then an optional sign, then a string of digits.

`atol` - convert string to long integer

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

This function converts the string pointed to by `nptr` to long integer representation. The first unrecognised character ends the string.

atol recognises an optional string of tabs and spaces, then an optional sign, then a string of digits.

In 32000 C, `atol` is equivalent to `atoi (int)` and `(long int)` have the same representation. since

`ceil` - ceiling function

```
#include <math-h>

int ceil(x)
double x;
```

`ceil` returns the smallest integer not less than `x`.

`clearerr` - clear stream errors

```
#include <stdio-h>

clearerr(stream)
FILE *stream;
```

clearerr resets any error indication on the named stream. It is implemented as a macro and therefore may not be redeclared.

close - close a file

```
close(fildes)
```

Given a file descriptor (fildes) as returned by open or creat, close closes the associated file, i.e. breaks the connection between the file descriptor (a small integer) and the file itself. A close of all files is automatic on exit, but since there is a limit on the number of files which may be open at once, close is necessary for programs which deal with many files.

Zero is returned if a file is closed, -1 is returned for an unknown file descriptor.

cos - cosine function

```
#include <math-h>

double cos(x)
double x;
```

cos returns the cosine of its radian argument.

creat - create a new file

```
creat(name, mode)
char *name;
```

creat creates a new file or prepares to rewrite an existing file called name, given as the address of a NUL-terminated string. The mode argument is currently ignored, but should be given by the caller for portability.

exit - terminate execution

```
exit(status)
int status;
```

exit is the normal means of terminating program execution. exit closes all the program's files, then stops.

This call never returns.

exp - raise e to a power

```
#include <math-h>

double exp(x)
double x;
```

exp returns the exponential function of x.

exp returns a huge value when the correct value would overflow; errno is set to "ERANGE".

fabs - floating absolute value

```
#include <math-h>

double fabs(x)
double x;
```

fabs returns the absolute value |x|.

fclose - closes a file

```
#include <stdio-h>

fclose(stream)
FILE *stream;
```

fclose causes any buffers for the specified stream to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

fclose is called automatically upon calling exit.

fclose returns EOF if stream is not associated with an output file, or if buffered data cannot be transferred to that file.

fdopen - open a stream

```
#include <stdio-h>

FILE *fdopen(fildes, type)
char *type;
```

fdopen associates a stream with a file descriptor obtained from open or creat.

type is a character string having one of the following values:

r open for reading

w create for writing

a append: open for writing at end of file or create for writing

The type of the stream must agree with the way the file was opened. the open file.

feof - is stream at end of file?

```
#include <stdio-h>

feof(stream)
FILE *stream;
```

feof returns non-zero when end of file is read on the named input stream, otherwise zero. It is implemented as a macro, and therefore cannot be redeclared.

ferror - tests for stream errors

```
#include <stdio-h>

ferror(stream)
FILE *stream;
```

ferror returns non-zero when an error has occurred reading the named stream, otherwise zero. Unless cleared by clearerr, the error indication lasts until the stream is closed. ferror is implemented as a macro.

fflush - flush stream buffer

```
#include <stdio-h>

fflush(stream)
FILE *stream;
```

fflush causes any buffered data for the named output stream to be written to the file or device associated with that stream. The stream remains open.

fflush is called automatically by close, and when all streams are implicitly closed by exit.

EOF is returned if `stream` is not associated with an output file or if buffered data cannot be transferred to that file.

`fgetc` - read a character from a stream

```
#include <stdio-h>

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

`fgetc` returns the next character from the specified input stream. Successive calls return successive characters from the stream. `fgetc` is a genuine function, unlike `getc` which is a macro.

EOF is returned at end of file or if a read error occurs.

`fgets` - read a string from a stream

```
#include <stdio-h>

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

`fgets` reads n-1 characters, or up to a newline character, whichever comes first, from the `stream` into string `s`. The last character read into s is followed by a NUL character. `fgets` returns its first argument.

fgets returns NULL on end of file or error.

Note that `fgets` behaves differently from `gets` (q.v.) with respect to any terminating newline character: `fgets` keeps the newline, `gets` deletes it from the string.

`fileno` - stream status enquiry

```
#include <stdio-h>

fileno(stream)
FILE *stream;
```

`fileno` returns the integer file descriptor associated with the stream, see open. It is implemented as a macro.

`floor` - floor function

```
#include <math-h>

int floor(x)
double x;
```

`floor` returns the largest integer not greater than `x`.

`fopen` - opens a file

```
#include <stdio-h>

FILE *fopen(filename, type)
char *filename, *type;
```

`fopen` opens the file named by `filename` and associates a stream with it. `fopen` returns a pointer to be used to identify the stream in subsequent operations.

`type` is a character string having one of the following values:

r open for reading

w create for writing

a append: open for writing at end of file or create for writing

`fopen` returns the pointer NULL if `filename` cannot be accessed.

`fprintf` - formatted output

```
#include <stdio-h>

fprintf(stream, format, [ , arg1 [ , arg2 [ ... ] ] ])
FILE *stream;
char *format;
```

`fprintf` writes its output on the specified stream (by calling `putc`). `fprintf` converts, formats and outputs the arguments `arg(i)` under control of its `format` argument. The `format` argument is a character string which contains two types of object: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and output of the next `arg` value.

Each conversion specification is introduced by the character %. Following the % there may be (in the given order)

- an optional minus sign '-', which specifies left justification of the converted value in the indicated field.

- an optional digit string specifying a field width; if the converted value has fewer characters than the field width it will be blank padded on the left (or right if left justification indicator '-' has been given) to make up the field width; if the field width begins with a zero, zero-padding will be performed instead of blank-padding.

- an optional digit string specifying a precision which specifies the number of digits to appear after the decimal point for e and f format conversion, or the maximum number of characters to be output from a string.

- the character l (lowercase L), specifying that a following d, o x or u corresponds to a long integer arg. (A capitalised conversion code has the same effect).

- a character which indicates the type of conversion to be applied.

A field width or precision may be specified as "*" instead of a digit string, in which case a corresponding integer arg is used as the field width or precision respectively.

The conversion characters and their meanings are:

dox

The integer arg is converted to decimal, octal or hexadecimal notation respectively.

f

The float or double arg is converted to decimal notation in the form "[-]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e

The float or double arg is converted into the form "[-]d.ddde[-|+]dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is not specified, 6 digits are produced.

g

The float or double arg is output in style d, f or e, whichever gives full precision in minimum space.

`c`

The (char) `arg` is printed. NUL characters are ignored.

`s`

`Arg` is taken to be a string (character pointer) and characters from the string are printed until a NUL character is reached or until the number of characters indicated by the precision specification is reached; however if the precision is zero or missing, all characters up to a NUL are printed.

`u`

The unsigned integer `arg` is converted to decimal and output.

`%`

Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by `fprintf` are printed by `putc` (q.v.).

Note that fields wider than 128 characters do not work.

`fputc` - write a character to a stream

```
#include <stdio-h>

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

`fputc` appends the character `c` to the specified output stream. It returns the character written. `fputc`, unlike `putc`, is a genuine function rather than a macro.

`fputc` returns `EOF` if an error occurs.

`fputs` - write a string to a stream

```
#include <stdio-h>

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

`fputs` copies the NUL-terminated string s to the specified output stream. The `NUL` character which terminates the string is not written to the stream.

Note that fputs is inconsistent with puts, which appends a newline to the output string.

fread - buffered binary input

```
#include <stdio-h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

fread reads into a block beginning at ptr, nitems of data of the type of *ptr from the specified input stream. It returns the number of items actually read. fread returns zero on end of file or error.

freopen - open a stream

```
#include <stdio-h>

freopen(filename, type, stream)
char *filename, *type;
FILE *stream;
```

freopen substitutes the named file filename in place of the open stream. It returns the original value of stream. The original stream is closed. freopen is typically used to attach the preopened constant names, stdin, stdout and stderr to specified files. type is a character string having one of the following values:

r open for reading

w create for writing

a append: open for writing at end of file or create for writing

freopen returns the pointer NULL if filename cannot be accessed.

fscanf - formatted input

```
#include <stdio-h>

fscanf(stream, format, [ , ptr1 [ , ptr2 [ ... ] ] ] )
FILE *stream;
char *format;
```

fscanf reads characters from the specified input stream, interprets them according to a format string and stores the results in the variables pointed to by its arguments. The format string usually contains conversion specifications which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.

2. An ordinary character (not %) which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character, '*', an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding `ptr` argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

`%`
: a single '%' is expected in the input at this point; no assignment is done.

`d`
: a decimal integer is expected; the corresponding argument should be an integer pointer.

`o`
: an octal integer is expected; the corresponding argument should be an integer pointer.

`x`
: a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

`s`
: a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

`c`

a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, use "%1s". If a field width is given, the corresponding argument should point to a character array, and the indicated number of characters is read.

`e,f`

a floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

`[`

Indicates a string not to be delimited by space characters. The left bracket is followed in the format string by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex (`^`), the input field is all the characters until the first character not in the set between the brackets; if the first character after the left bracket is `^`, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be capitalised or preceded by l (lowercase L) to indicate that a pointer to `long` rather than to `int` is in the argument list. Similarly, the conversion characters e or f may be capitalised or preceded by l (lowercase L) to indicate a pointer to double rather than to float. The conversion characters d, o and x may be preceded by h to indicate a pointer to `short` rather than to `int`. `fscanf` returns the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant `EOF` is returned on end of input; note that this is different from zero, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

fseek - reposition a stream

```
#include <stdio-h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;
```

fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, the current position or the end of the file, depending on whether ptrname has the value 0, 1, or 2. fseek undoes any effects of ungetc.

fseek returns -1 for improper seeks.

ftell - stream position enquiry

```
#include <stdio-h>

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

ftell returns the current value of the offset relative to the beginning of the file associated with the named stream. This offset is measured in bytes.

fwrite - buffered binary output

```
#include <stdio-h>

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

fwrite appends at most n items of data of the type of *ptr beginning at ptr to the specified output stream. It returns the number of items actually written.

Zero is returned on end of file or error conditions.

getc - read a character from a stream

```
#include <stdio-h>

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

getc returns the next character from the named input stream. Successive calls on getc return successive characters from the stream. getc is implemented as a macro.

EOF is returned on end of file or when a read error is detected.

getchar - read a character from standard input

```
#include <stdio-h>

int getchar()
```

Getchar() is identical to getc(stdin). (Getchar is implemented as a macro). It returns the next character from the standard input stream stdin. EOF is returned on end of file or read error conditions.

getenv - access environment variable

```
char *getenv(name)
char *name;
```

name is a pointer to a string which must be the name of a Panos string variable. If this global string is defined, getenv returns a pointer to the corresponding global string value, otherwise a null pointer is returned.

gets - read string from standard input

```
#include <stdio-h>

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

gets reads a string into s from the standard input stream stdin. The string is terminated by a newline character, which is replaced in s by a character.

gets returns its argument as result.

gets returns NULL on end of file or error.

Note that gets is inconsistent with fgets (q.v.) in its treatment of the terminating newline character: gets deletes the newline, fgets keeps it.

isalnum - is character alphanumeric? (macro)

```
#include <ctype-h>
isalnum(c)
```

`isalnum` returns 1 if the argument `c` is a letter or a digit, 0 otherwise.

`isalpha` - is character alphabetic? (macro)

```
#include <ctype-h>
isalpha(c)
```

`isalpha` returns 1 if the argument `c` is a letter, 0 otherwise.

`isascii` - is argument an ASCII character? (macro)

```
#include <ctype-h>
isascii(c)
```

`isascii` returns 1 if the argument `c` is an ASCII character (code less than 200 octal).

`isatty` - is file descriptor a terminal?

```
#include <stdio-h>

isatty(fildes)
```

`isatty` returns 1 if the file descriptor `fildes` is associated with a terminal device, 0 otherwise.

`iscntrl` - ASCII control character? (macro)

```
#include <ctype-h>
iscntrl(c)
```

`iscntrl` returns 1 if the argument `c` is an ASCII control character, 0 otherwise.

`isdigit` - is argument a digit? (isdigit)

```
#include <ctype-h>
isdigit(c)
```

`isdigit` returns 1 if the argument `c` is a digit, 0 otherwise.

`islower` - is character lowercase? (macro)

```
#include <ctype-h>
islower(c)
```

`islower` returns 1 if the argument `c` is a lowercase letter, 0 otherwise.

`isprint` - printing ASCII character? (macro)

```
#include <ctype-h>
isprint(c)
```

`isprint` returns 1 if the argument `c` is a printing character, codes 40 octal (space) through 176 octal (tilde). Returns 0 otherwise.

`ispunct` - punctuation character? (macro)

```
#include <ctype-h>
ispunct(c)
```

`ispunct` returns 1 if the argument `c` is a punctuation character (neither control nor alphanumeric), 0 otherwise.

`isspace` - white space character? (macro)

```
#include <ctype-h>
isspace(c)
```

`isspace` returns 1 if the argument `c` is a space, horizontal or vertical tab, carriage return newline or formfeed character, 0 otherwise.

`isupper` - is character uppercase? (macro)

```
#include <ctype-h>
isupper(c)
```

`isupper` returns 1 if the argument `c` is an uppercase letter, 0 otherwise.

`log` - natural logarithm

```
#include <math-h>

double log(x)
double x;
```

`log` returns the natural logarithm of `x`. `log` returns 0 when `x` is zero or negative; the external `int` variable `errno` is set to `EDOM` (defined by `#include <errno-h>`).

`longjmp` - non-local goto

```
#include <setjmp-h>

longjmp(env, val)
jmp_buf env;
```

This function is used to deal with errors encountered in a low-level subroutine of a program.

`longjmp` restores the stack environment saved by the last call of `setjmp` with `env` as its argument. It then returns in such a way that execution continues as if the call of `setjmp` which saved the environment `env` had just returned the value `val` to the function which called `setjmp`, which must not itself have returned in the interim. All accessible data still have their values as of the time `longjmp` was called.

`lseek` - move read/write pointer

```
long lseek(fildes, offset, whence)
long offset;
```

The file descriptor refers to a file open for reading and writing. The read (resp. write) pointer for the file is set as follows:

> If `whence` is 0, the pointer is set to `offset` bytes. If `whence` is 1, the pointer is set to its current location plus `offset`. If `whence` is 2, the pointer is set to the size of the file plus `offset`.

The returned value is the resulting pointer location. -1 is returned for an undefined file descriptor or a seek to a position before the beginning of the file. `lseek` is a no-op on devices (e.g. the vdu or keyboard) which are not disc files.

`open` - open for reading or writing

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

`open` opens the file `name` for reading (if `mode` is 0), writing (if `mode` is 1) or for both reading and writing (if `mode` is 2). `name` is the address of a string of ASCII characters representing an filing system file name, terminated by an ASCII NUL character.

The file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

The value -1 is returned if the file does not exist or is unreadable or if too many files are already open.

perror - print error message

```
#include <stdio-h>

char *perror(s)
char *s;
```

The perror function converts the value in the global variable errno into a textual message. If s is not null, perror writes a line to the standard error file thus: first the string pointed to by s then a colon and a space, then the message and a newline. If the argument s is a null pointer, the perror function returns a pointer to the message string and performs no output.

The message string is of the form:

PANOS error code 16_xxxxxxxx

printf - formatted output on stdout

```
#include <stdio-h>

printf(format, [ , arg1 [ , arg2 [...] ] ] )
char *format;
```

printf writes output to the standard output stream, stdout. The arguments of printf have the same meaning as the fprintf arguments of the same name. See the description of fprintf.

```
printf(...);
```

is equivalent to

```
fprintf(stdout, ...);
```

putc - append a character to output stream

```
#include <stdio-h>

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

putc appends the character c to the specified output stream. It returns the character written. EOF is returned on error. Because it is implemented as a macro, putc treats a stream argument with side-effects improperly. In particular,

```
putc(c, *f++);
```

does not work sensibly.

putchar - write a character to standard output

```
#include <stdio-h>

putchar(c)
```

putchar(c) is a macro defined as putc(c,stdout) i.e. the character c is written to the standard output stream, stdout (normally the vdu).

EOF is returned on error.

puts - write string to standard output

```
#include <stdio-h>

puts(s)
char *s;
```

puts copies the NUL-terminated string s to the standard output stream stdout and appends a newline character. The terminating NUL character is not copied. stdout is normally the vdu.

puts appends a newline to the output string but fputs (q.v.) does not.

putw - write an integer to standard output

```
#include <stdio-h>

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

putw outputs an integer value to the standard output stream in a format which can be read in again by the standard input function getw.

Putw returns the word written. putw neither assumes nor causes special alignment in the file.

EOF is returned if a write error occurs.

rand - pseudo-random number generator

```
int rand()
```

The rand function returns successive pseudo-random integers in the range 0 to 32767. See also srand.

`read` - read from file

```
read(fildes, buffer, nbytes)
char *buffer;
```

A file descriptor is an integer returned by a successful call on `open` or `creat`. `Buffer` is the location of `nbytes` contiguous bytes into which the input will be placed. It is not guaranteed that all `nbytes` bytes will be read; for example if the file descriptor refers to the keyboard at most one line will be returned. In any event, the number of characters actually read is returned.

Zero is returned when the end of the file has been reached. If the read was unsuccessful for any other reason, -1 is returned. Many conditions may cause errors: physical I/O errors, bad buffer address etc.

`rewind` - reposition stream to beginning

```
#include <stdio-h>

rewind(stream)
```

`rewind(stream)` is equivalent to `fseek(stream,0L,0)`. It repositions `stream` to the first byte of the associated file (byte 0). It is a no-op if the stream is associated with a device rather than a file (e.g. the keyboard or the vdu).

`rewind` returns -1 on failure.

`scanf` - formatted input from stdin

```
#include <stdio-h>

scanf(format, [ , ptr1 [ , ptr2 [ ... ] ] ] )
char *format;
```

`scanf` reads input from the standard input stream `stdin`. It reads characters (via `getc`), interprets them according to the given format and stores the resulting values in the locations pointed to by the `ptr` arguments.

The exact meaning of the arguments to `scanf` is the same as that of the arguments of the same name to the function `fscanf`. In fact, the call

```
scanf(format, ...);
```

is equivalent to

```
fscanf(stdin, format, ...);
```

scanf returns EOF on end of input, and a short count for missing or illegal data items.

setbuf - assign buffering to a stream

```
#include <stdio-h>

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

setbuf is used after a stream has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer.

Note: if buf is the constant pointer NULL, input/output will be performed without any buffering being interposed by the stdio package.

A macro BUFSIZ tells how big an array is needed: char buf[BUFSIZ]; A buffer is normally obtained from malloc upon the first getc or putc on the file, except that output streams directed to the vdu and the standard error stream stderr are normally not buffered.

setjmp - save environment for longjmp

```
#include <setjmp-h>

setjmp(env, val)
jmp_buf env;
```

setjmp saves the current stack context of the program in env and returns zero. longjmp (q.v.) can be used to transfer control back to the call of setjmp with the same auto variables accessible as when setjmp was called. setjmp returns a non-zero value (supplied by the call on longjmp if control is transferred in this way).

sin - sine function

```
#include <math-h>

double sin(x)
double x;
```

sin returns the sine of its radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

sprintf - formatted output to a string

```
#include <stdio-h>

sprintf(s, format, [ , arg1 [ , arg2 [ ... ] ] ] )
char *s, *format;
```

sprintf writes formatted output into a character array via a pointer s supplied by the caller. The meaning of format and the arg values is as for fprintf. The output string s is automatically terminated by a NUL character.

sqrt - square root

```
#include <math-h>

double sqrt(x)
double x;
```

sqrt returns the square root of x. sqrt returns zero when x is negative; the extern int variable errno is set to EDOM (defined by #include <errno-h>).

srand - new seed for rand function

```
srand(seed)
unsigned int seed;
```

The srand function uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand.

sscanf - formatted input from string

```
#include <stdio-h>

sscanf(s, format [ , ptr1 [ , ptr2 [ ... ] ] ] )
char *s, *format;
```

sscanf reads input from the string s. It interprets the characters it reads according to the given format string and stores the resulting values in the locations pointed to by the ptr arguments. The exact meaning of the arguments to sscanf is the same as for fscanf.

`strcat` - concatenates two strings

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

`strcat` appends a copy of string `s2` to the end of string `s1`. A pointer to the NUL-terminated result is returned.

`strcmp` - string compare

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

`strcmp` compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether `s1` is lexicographically greater than, equal to or less than `s2`.

`strcpy` - string copy

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

`strcpy` copies string `s2` to `s1`, stopping after the NUL character has been moved. `s1` is returned.

`strlen` - string length

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

`strlen` returns the number of non-NUL characters in `s`.

`strncat` - string concatenate

```
char *strncat(s1, s2, n)
char *s1, *s2;
```

`strncat` appends a copy of string `s2` to the end of string `s1`. It copies at most `n` characters. A pointer to the NUL-terminated result is returned.

`strncmp` - string compare

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

`strncmp` compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether `s1` is lexicographically greater than, equal to or less than s2. At most n characters are looked at.

strncpy - string copy

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

strncpy copies string s2 to s1. Exactly n characters are copied: s2 is truncated or NUL-padded as required. The target may not be NUL terminated if the length of s2 is n or more. s1 is returned.

system - call command interpreter

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

The string is passed to the Panos command line interpreter interface procedure InterpretString and processed as if it had been entered as a Panos command. The Panos ResultCode value is left in the global variable errno (for inspection by perror if required).

system always returns a non-zero value to indicate that it is not a null command processor.

tolower, _tolower - convert char to lower case

```
int tolower(c)
int c;

#include <ctype-h>
int _tolower(c)
int c;
```

If c is the ASCII code for an upper case letter, tolower returns the code for the corresponding lower case letter, otherwise the value of c is returned unchanged. _tolower behaves like tolower but is implemented as a macro.

toupper, _toupper - convert char to upper case

```
int toupper(c)
int c;

#include <ctype-h>
int _toupper(c)
int c;
```

If c is the ASCII code for a lower case letter, toupper returns the code for the corresponding upper case letter, otherwise the value of c is returned unchanged. _toupper behaves like toupper but is implemented as a macro.

ungetc - push character back into input stream

```
#include <stdio-h>

ungetc(c, stream)
FILE *stream;
```

ungetc pushes the character c back on an input stream. That character will be returned by the next getc call on that stream. ungetc returns c.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

fseek (q.v.) erases all memory of pushed back characters.

ungetc returns EOF if it can't push a character back.

write - write on a file

```
write(fildes, buffer, nbytes)
char *buffer;
```

A file descriptor is the integer returned by a successful call on open or creat. buffer is the address of nbytes contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested. write returns -1 on error: bad descriptor, buffer address or count; physical I/O errors.

5 Debugging

This chapter shows how error conditions are reported by the compiler, outlines ways of dealing with errors, and gives general information about the kinds of error which may occur once a program has been successfully compiled and is being run.

5.1 Compiler error message format

This section describes the error reports displayed by the compiler when it has detected errors in a program.

If an attempt is made to compile a program which does not obey all of the rules of C, the compiler will display a message indicating the nature of the fault and showing where in the program the error was trapped. For example, in the following program, the brackets which must surround the expression following `while` have been omitted:

```
main()
{
            int i = 0;

            while i++ < 10
             printf("hello, world\n");
}
```

The compiler will discover the error and display a message like this:

```
* "Ctest-C" line 5 while i++ < 10
                           ^
                           ( expected
```

The upward arrow character ^ points to the place where the error was found.

Notice the format of the message: all of the messages which the compiler produces appear in a similar form. The first character in the message is a marker which indicates how bad the error was - an asterisk * is the usual sort of error; it means that the compiler has detected a fault but is able to continue trying to compile the rest of the program. The other markers which can appear are described later.

Following the marker character is the name of the file in which the error occurred followed by a number. This is the line number in the original C program at which the error was detected. Here the error is on line five. Wherever possible the compiler displays the text of the offending line after the line number, as in the example program, but because the original text is stored in a fixed size memory area, this cannot always be done. If the source text is no longer in memory it is omitted from the error message.

The general form of compiler messages is therefore:

```
   <marker> "<filename>" line <number>: <source-line>

                                        ^

                                        <message-text>
```

Where ^ points to the part of the source-text in error, <message-text> is an English description of the fault, and <marker> may be any of `*`, ? or !.

Marker	Meaning
*	Error: compilation continues
?	Warning: a part of the program is strictly correct, but is dubious in some way. For example, if some part of a program can never be reached.
!	Fatal error. Compilation cannot continue after a fatal error. Fatal errors indicate either that a program is too large or complicated to be compiled in the amount of memory available or that there is a fault in the compiler itself which makes it unable to compile this program.

The <line-number> is filled out with spaces on the left-hand side to be at least four characters long; it shows which line of the source file being compiled the error was found in. The line number information can be used to locate the incorrect line quickly with a text editor even when a program contains `#include` statements, because each `#include` counts as a single line, no matter how many lines the included file contains. See the two C program files called `main` and `error` below.

```
#include "error"

main()
{
while i++ < 10
 printf("hello n");
}
```

```
/* error included text */

auto int i;
```

Figure 1

If we compile `main`, we will get the following error messages:

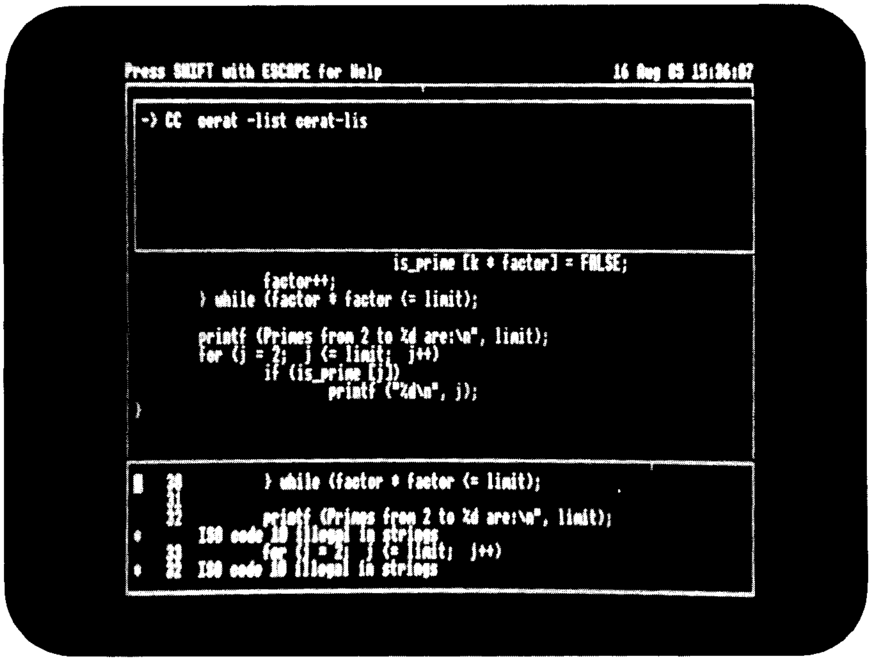
```
* "error-h" line 3: auto int i;

                           ^

an external data definition may not have storage class auto

* "main-c" line 5  while i++ < 10

                   ^

                   ( expected
```

These messages indicate that in line three of the included file `error` the declaration of `i` is not allowed (because only static or extern declarations are allowed at the outermost level of a program), and in line five of `main` an opening bracket must follow the keyword `while`. Once the compiler has read the complete program, it displays a summary message indicating how many errors were detected:

```
nnnn errors detected
```

('nnnn' is a decimal number).

Normally, error messages are sent to the screen, but by the use of the compiler '-error' option (see section 2.1), these messages can be redirected. See figure 2 for an example of an erroneous program compiled from within the editor. The error messages have been sent to a listing file which has then been copied into an editor window.



```
Press SHIFT with ESCAPE for Help                    16 Aug 85 15:36:07

 -> CC  erat -list erat-lis

                                is_prime [k * factor] = FALSE;
              factor++;
        } while (factor * factor <= limit);

        printf (Primes from 2 to %d are:\n", limit);
        for (j = 2;  j <= limit;  j++)
              if (is_prime [j])
                    printf ("%d\n", j);
 }

 ■  30          } while (factor * factor <= limit);
    31
    32          printf (Primes from 2 to %d are:\n", limit);
 *     150 code 10 illegal in strings
    33          for (j = 2;  j <= limit;  j++)
 *  33 150 code 10 illegal in strings
```

Figure 2 Compiling a C program from within the editor

5.2 Fixing errors detected by the compiler

This section contains information about how the compiler handles errors in the program which it is trying to compile. This information should make it easier to understand the messages displayed by the compiler, and so make it easier to correct programs.

The compiler can detect two classes of error: errors in the syntax of a program (missing semicolons, misspelt keywords etc.) and errors where an identifier of a particular type is used in the wrong context (such as attempting to multiply a struct variable by a float value or use an identifier that has not been declared). Syntax errors are detected when the compiler discovers that the piece of program it is reading does not fit in the context of the program it has already processed. When this happens, the compiler displays a message and starts reading on from the point of the error, ignoring everything until it finds a symbol which could fit in at this point in the program; compilation then continues as though there had been no error.

Because the compiler may ignore vital parts of the program (like declarations) in recovering from an error, the best policy when fixing errors reported by the compiler is to deal with them one by one, starting with the first. Look at the part of the program indicated by the error message and try to find out what is wrong with it, then fix the problem and recompile the program. If errors are dealt with sequentially like this, you will not waste time hunting for spurious errors caused by the compiler skipping over some declarations and then complaining about "undeclared identifiers" in the rest of the program. Examine the example below, where a comma in a declaration has been mistyped as a dot.

```
main()
{
            int i . j;

            i = j ;
}
```

This compiler will display the following messages:

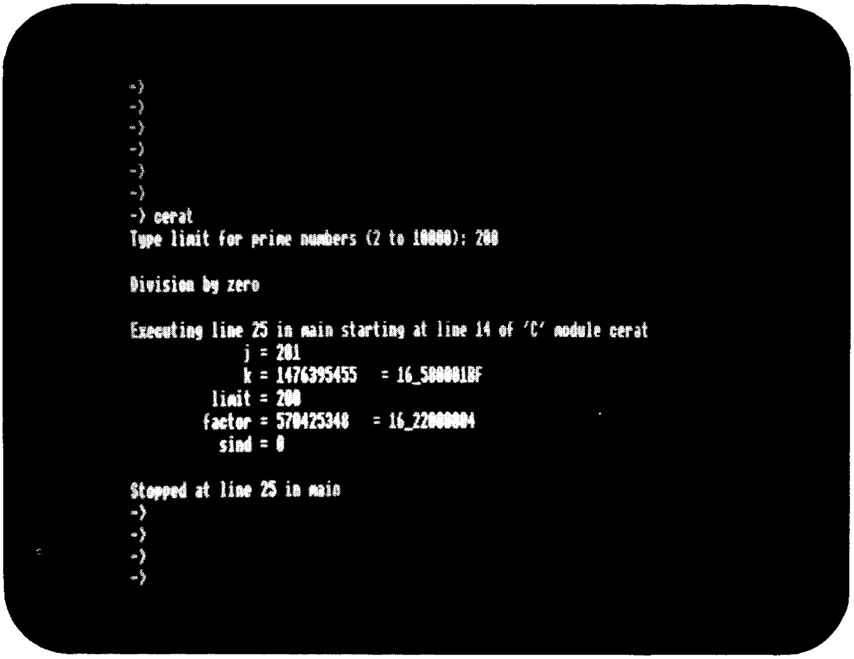
```
* "Ctest-c" line 3:  int i . j;
                            ^
                            ; expected
* "Ctest-c" line 5:  i = j ;
                           ^
                           j not declared
```

The first message indicates that a semicolon or comma must follow each identifier in a declaration (a dot is not allowed). Because the compiler has skipped the declaration of `j` in order to get back in step with the program, `j` appears not to be declared in line five resulting in the second error message.

If you correct this program as suggested above, by starting with the first error, fixing it and then recompiling the program then you will never have to worry about fixing the second error: it will go away automatically when the first error is fixed.

5.3 Errors detected during execution

In general, errors such as attempting to divide by zero or use an illegal pointer value (e.g. NULL) will be detected and will cause diagnostic routines in the C runtime library to display information about what the program was doing when the error occurred, including a backtrace through all the active function calls giving line number information and showing the values of the variables in each function. See figure 3 for an example of a run-time error caused by dividing by zero.



```
-)
-)
-)
-)
-)
-)
-) cerat
Type limit for prime numbers (2 to 10000): 200

Division by zero

Executing line 25 in main starting at line 14 of 'C' module cerat
              j = 201
              k = 1476395455   = 16_580001BF
          limit = 200
         factor = 570425348   = 16_22000004
           sind = 0

Stopped at line 25 in main
-)
-)
-)
-)
```

Figure 3 Example run-time-error

Appendix A

Error Messages

The messages listed here may be output by the compiler while a program is being compiled. Section 5.1 describes the format of complete messages.

Some messages contain special character sequences like !1, !2 etc. These do not appear in the actual message displayed by the compiler, rather they are replaced by the appropriate text from the program. For example, take the message

```
"!1" not declared
```

If it is the identifier `foo` which is not declared, the message actually displayed will be:

```
foo not declared
```

List of Error Messages

```
unimplemented feature !1
too many names
ISO code !1 illegal in strings
bad escape code \'!1'
expression syntax fault
'!1' character not allowed here
sizeof operand must be a type name or unary expression
missing operand
missing )
identifier expected
!1 expected
declaration syntax fault
expression expected
an empty enumerator list is not allowed
an empty structure is not allowed
```

```
identifier or {struct-decl-list} required after
'struct'/'union'
'{' function-body '}' expected here; could be missing ; after ) above?
statement expected here
a compiler-control (#) line may not begin with "!1"
format is #include "file" or #include <file>
syntax error in compiler-control (#) line
too many nested #include files
include stack underflow
closing '>' expected
macro text store full
macro expansion stack full
number of macro actual parameters does not agree with definition
too many macro parameters
array dimension table full
internal error
corrupt syntax tree
storage class incompatible with a previous declaration
"!1" incompatible with type "!2"
constant integer expression required here
"!1" and "!2" are incompatible operand types for the "!3" operator
"!1" not declared
left operand of "!1" is not an lvalue
"%" must have integer operands
operand of unary "!1" must be an lvalue
operand of unary * must have pointer type
unary "!3" may not have an operand of type "!1"
"!1" is not in the parameter list of "!2" and so may not appear here
declarator may only contain a single formal parameter list
function declarator required before "{"
a function result of type "!1" is not allowed
expression of type "!1" cannot be used as a function
this statement is inaccessible
switch expression must have integer type case and default are only
allowed inside a switch statement
only one default statement is allowed per switch statement
left operand of "." must be a structure
structure of this type has no "!1" field
a parameter may not have storage class "!1"
an external data definition may not have storage class "!1"
```

```
a parameter declaration may not be initialized
unexpected colon in statement context
both operands for pointer "-" must have the same type
applying sizeof() to a function is illegal
too many initializers for object of type "!1"
auto/register arrays may not be initialized
original and result types for cast must be scalar or pointers
initializer string longer than array
#endif/#else without matching #if
one or more #endif lines inserted before extra #else here
union type objects may not be initialized
a bit-field must have an integer type
a field may not exceed !1 bits
a constant integer expression is required here
not a constant
"!1" operator not allowed in a constant-expression
attempt to divide by zero
implementation restriction: sizeof not allowed in this context
implementation restriction: structs with bit-fields may not be
initialized
only extern or static functions are allowed
type "!1" may not be unsigned
implementation restriction: pointers to functions cannot be initialized
label "!1" is used in function "!2" above but is not defined there
label "!1" has already been defined in this function
constant integer value too large
expression of type "!1" used instead of int
expanded macro line too long
cc fails -- not enough memory
```

Appendix B

Bibliography

Standard C is defined in the "C Reference Manual" part of Kernighan and Ritchie's book *The C Programming Language* [Prentice-Hall, 1978].

In order to use 32000 C you will need access to the information in this book, which contains an excellent tutorial introduction to computer programming in general as well as the definition of C.

Index

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200