

***Introduction to the C
programming language***



PERSONAL COMPUTER

PB-2000C

CASIO®

Foreword

The C programming language is probably the most popular in use today. This is because C has a number of advantage over other languages.

One advantage is that, despite the fact that C is a relatively high-level language, it allows use of detailed processing notation that is very close to machine language. For this reason, C is the choice for a wide variety of programming applications, from scientific and technical calculations, to mechanical control.

The usefulness of a computer depends entirely upon the type of software it is programmed with. This means that the ideal situation is one in which the computer operator is able to program the computer to perform specific tasks. The best way to develop such an ability is through practice in writing and executing your own programs, moving from simple functions to more complex operations.

The PB-2000C comes with the C programming language built in, offering more power than ever before in a pocket computer. Now you can learn about all of the features and functions of the C programming language on a pocket computer that you can take along anywhere.

The procedures outlined in this manual are designed to give you some hands-on experience with the C programming language built into the Casio PB-2000C pocket computer. Chapter 2 and Chapter 3 provide a variety of actual programs that you can enter and execute, while gaining some insights into the functions of C. Chapter 4 contains even more sample programs covering everything from graphic animation to complex technical calculations. After you work with these samples for awhile, you should soon be well on the way to programming proficiency, with the ability to create a whole library of customized, original applications.

- | | |
|--|---|
| <ul style="list-style-type: none">• The contents of this manual are subject to change without notice.• Note that other than personal use of this manual without the permission of CASIO is prohibited under copyright laws.• Unlawful copying of this manual in part or in entirety is expressly forbidden.• Casio Computer Co., Ltd. assumes no responsibility for any damage or loss resulting from the use of this manual. | <ul style="list-style-type: none">• Casio Computer Co., Ltd. assumes no responsibility for any loss or claims by third parties which may arise through the use of the PB-2000C.• Casio Computer Co., Ltd. assumes no responsibility for any damage or loss caused by deletion of data as a result of malfunction, repairs, or battery replacement. Be sure to back up all important data on other media to protect against its loss. |
|--|---|

Contents

Foreword iii

Part 1 Introduction to the C Language	1
--	----------

Chapter 1 The Basics of C 3

1-1 Introduction to C 4

 Early history of C 4

 Why C? 4

 Features of C 4

1-2 Learning about C with the PB-2000C 5

1-3 Meet the PB-2000C interpreter 6

Chapter 2 C Interpreter Operation 7

2-1 About the C interpreter and editor 8

 About the C interpreter 8

 About the C editor 8

2-2 Activating the C function 9

 To activate the C function 9

 About the function key menus 9

 Function key menu commands used in C 10

2-3 Interpreter displays 13

 Interpreter 13

 Using command-line operation 13

2-4 Handy programming shortcuts 17

 Using the RECALL function 17

 Using one-key commands 17

2-5 Using the editor 19

 Entering the editor 19

 Moving the cursor 22

 Editing a C program 22

 Exiting the editor and loading an edited file 24

2-6 Checking and specifying memory status 25

 To check the memory status 25

 To specify the memory status 26

 Memory Map 28

 Memory contents 29

2-7	Using the LOAD command	30
	Creating a file.....	30
	Loading and executing programs.....	32
2-8	Execution using batch files	33
	To create a batch file.....	34
	Executing a batch file.....	35
2-9	Using the trace function	37
	To enter the TRACE mode.....	38
	To interrupt execution in the TRACE mode.....	39
	To exit the TRACE mode.....	40
Chapter 3	Introduction to C	41
3-1	Outputting characters	42
	Creating a program to output character strings.....	42
	Making your program easy to read.....	44
	Creating a program to output numeric values.....	44
	Integer notation in C.....	46
3-2	Variable types and operations	47
	Declaring variable types.....	47
	Assigning values to variables.....	48
	Using arrays.....	49
3-3	Entering characters and values	50
	Entering a single character from the keyboard.....	50
	Entering values.....	51
3-4	Using selection statements	52
	Using the “if” selection statement.....	52
	A program to solve quadratic equations.....	53
	Relational operators.....	54
3-5	Using loops	55
	Using the “while” loop.....	55
	Using the #define statement.....	57
	Incrementing and decrementing.....	57
	Using the “do~while” loop.....	58
	Using the “for” loop.....	58
	Nested loops.....	60
	Inputting and outputting character strings.....	62
	Standard functions.....	64
3-6	Defining functions	65
	Function definitions and program modules.....	65
	Sample functions.....	66
	Recursive function calls.....	68

3-7	Local variables and global variables	68
	Local variables.....	68
	Global variables.....	69
3-8	Pointers and variable storage locations	70
	Entering values.....	70
3-9	File input and output	71
Chapter 4	Sample Programs	73
4-1	Prime numbers	74
4-2	Memory display	75
4-3	Perpetual calendar	76
4-4	Since curve/cosine curve program	79
4-5	Simple Martian animation	80
4-6	More Martian animation	82
4-7	Pseudo-random number generator	85
4-8	Approximation of pi	87
4-9	Mean and variance	89
4-10	Solution of simultaneous linear equations	91
Chapter 5	C Interpreter	93
5-1	Comments	94
5-2	Reserved words	94
5-3	Data types and lengths	95
5-4	Assigning variable names and function names	95
5-5	Data expressions	96
	Character constants.....	96
	Integer constants.....	99
	Floating-point constants and double precision floating-point constants.....	100
	String constants.....	100
5-6	Operators	100
	Precedence.....	100
5-7	Control structures	103
	Statements.....	103
	Compound statements.....	103
	Control structures for jumps and repeats.....	104
5-8	Storage classes	105
5-9	Arrays and pointers	108
	Arrays.....	108
	Pointers.....	109

5-10	Functions	111
	main function.....	111
	Function type declaration.....	111
5-11	Structures and unions	113
5-12	Preprocessor	115
5-13	Standard functions	116
	File input/output functions.....	116
	String functions.....	118
	Numeric functions.....	118
	Memory functions.....	119
	Standard functions for display and keyboard control.....	119
	Other standard functions.....	120

Part 2 Reference 121

Chapter 6	Command Reference	122
	LOAD/RUN.....	122
	NEW/FLIST/EDIT.....	123
	TRON.....	124
	TROFF.....	125
Chapter 7	Standard Function Reference	126
	Input/Output function	127
	fopen.....	127
	fclose/getchar.....	130
	getc/fgetc.....	131
	putchar.....	132
	putc.....	133
	fputc/gets.....	134
	fgets/puts.....	135
	fputs/fread.....	136
	fwrite.....	137
	printf, fprintf, sprintf.....	138
	scanf, fscanf, sscanf.....	141
	ungetc.....	143
	fflush.....	144

feof/ferror	145
clearerr/remove	146
rename	147
Process function	149
exit/abort	149
breakpt.	150
Memory function	151
malloc	151
calloc	152
free	153
String function	153
strlen	153
strcpy/strcat	154
strcmp	155
strchr	156
Numeric function	157
abs	157
sin, cos, tan	158
asin, acos, atan	159
sinh, cosh, tanh	160
asinh, acosh, atanh	161
pow/sqrt	162
exp/log, log10	163
angle	164
Other functions	164
beep	164
clrscr/gotoxy	165
getch	166
line, linec	167
Chapter 8 Error Message Tables	168
1. Command Error Messages	168
2. Syntax Analysis Error Messages	168
3. Program Execution Error Messages	171
4. errno Error	173
5. General Error Messages	174
Index	176

Part 1

Introduction to the C Language

Chapter 1	The Basics of C	3
Chapter 2	C Interpreter Operation	7
Chapter 3	Introduction to C	41
Chapter 4	Sample Programs	73
Chapter 5	C Interpreter	93

Chapter 1

The Basics of C

This chapter provides you with the basics of the C programming language, and introduces a number of features. Also included is important advice for programming, making this chapter recommended reading for everyone.

1-1 Introduction to C

Early history of C

C is a programming language that incorporates such concepts as module programming and structural syntax along the lines of ALGOL. Another well-known offshoot of ALGOL is PASCAL, and forebears of C are the CPL and BCPL languages. Both CPL and BCPL were early innovations by Britain's Cambridge University in an attempt to make ALGOL 60 easier to use. This concept crossed over to the United States, resulting in such languages as B and Mesa.

B language was developed as the notation of the UNIX operating system, and the development of an improved UNIX*¹ notation produced today's C language.

Why C?

Up until a number of years ago, the main programming languages for mainframe computing devices were COBOL, FORTRAN and PL/I, while microcomputers were programmed using assembler or BASIC. Recently, however, there is a definite movement towards the use of C for programming a variety of computers. But what has made C the language of choice for programming?

One reason is that C was developed for the UNIX operating system. UNIX, on the other hand, served as the basis for the widely popular MS-DOS*², which means that the UNIX notation system is applied in a wide variety of software in use today.

Another reason is the wide appeal of C due to its distinctive features noted below.

Features of C

1. Wide applicability

C can be used in a wide range of programming applications, from numeric calculations to machine control.

2. Simple program portability

Since C is a high-level programming language, programs can be used on a wide variety of computing devices.

3. Compact programs

The abundant operators and control structures that are built into C greatly simplify complex processing. Compared with other languages, the rules that govern C are relatively simple, making program creation quick and easy.

4. Control structures based on structured programming

Structured programming allows easily mastered programs that are similar to human thought patterns. This means that conditional branching and repeat loop controls are all included.

5. A host of operators

C includes arithmetic operators, logical operators, relational operators, incremental/decremental operators, bit logical operators, and bit shift operators.

6. Pointer control

Unlike the memory addresses used in FORTRAN, COBOL, and BASIC, C employs a pointer to indicate memory locations.

7. Efficient use of memory

Memory and data management functions, as well as programs are very compact, for more efficient memory utilization.

Thanks to features such as these, C gives you the high-level programming capabilities of FORTRAN and Pascal, with all of the detailed processing of machine language.

*¹ UNIX is a registered trademark of AT&T

*² MS-DOS is a registered trademark of Microsoft, Inc.

1-2 Learning about C with the PB-2000C

There once was a time that C could be seen running on 16-bit or larger computers. A time when it was only within the realm of programmers and system engineers. Since C notation resembles that of machine language, it often appeared too difficult or confusing for the casual user.

All of that has changed with the introduction of the Casio PB-2000C pocket computer. The PB-2000C now makes it possible for everyone to enjoy the many benefits of programming in C. The full portability of the pocket computer format means that the PB-2000C can go along with you everywhere, so you can program and compute whenever and wherever you want.

Since C interacts with many of the computer's functions, it would be difficult to obtain a solid grasp of its workings by simply reading this manual. We highly recommend that you get as much hands on experience as possible with the PB-2000C, creating and running your own programs. This is the best way to learn about the power and versatility of C.

The PB-2000C actually makes a very good teacher. Should you make a mistake when inputting a program, error messages will appear on the display to guide you back to the correct path. With each error you will be moving one more step closer to the mastery of C. A trace function is also provided to give you valuable insights of programs as they are executed.

1-3 Meet the PB-2000C interpreter

With most computers, C acts as a *compiler language*. A *compiler* translates the program written in C into *machine language*, which can directly be understood by the computer. The program that the computer actually executes is the machine language program.

Since the compiler must perform the three steps of *compiling*, *linking* and *execution* for each program, they cannot be executed immediately after you enter them. The two steps of compiling and linking require some time to perform.

The Casio PB-2000C, on the other hand, features a C interpreter. The *interpreter* translates programs as you enter them, making it much easier to operate than a compiler. Once you enter the program, all you have to do is enter the RUN command to execute the program. Everyone who has ever worked with the BASIC programming language will find this a very familiar process. The PB-2000C C interpreter also supports floating point calculations, structures, and unions, for much of the power of larger computers with C compilers.

Of course, a wide variety of scientific functions found on today's scientific calculators can also be incorporated into C programs with the PB-2000C.

Chapter 2

C Interpreter Operation

This includes a number of simple programs to give you experience with the C editor and C interpreter. Especially important is the loading of files into the interpreter, and we recommend that you take the time to master this operation completely.

2-1 About the C interpreter and editor

The PB-2000C provides you with both a C interpreter and editor. The *interpreter* is used to execute programs. The *editor* is used for program creation and editing. Before getting into the actual operations of the PB-2000C, perhaps it might be a good idea to first look at these two functions in a little more detail.

About the C interpreter

There are two ways to process a C program: using a compiler or using an interpreter. With the compiler, the program is first translated into machine language before it is executed. This means that each time you write a program, all of the following processes must be performed:

Program input → compiling → linking → execution.

With an interpreter, the program is entered and executed in the same environment, for easier operation.

The PB-2000C features a C interpreter. After you enter a program, all you have to do is enter the RUN command to execute it.

Throughout the rest of this manual, the C interpreter will be referred simply as an “interpreter”.

About the C editor

An editor can be used to write everything from a few lines to an entire program. Unlike the interpreter in which simple writing cannot be performed easily, the editor lets you easily modify and correct programs.

The C editor of the PB-2000C is for writing and modifying C programs. You should note, however, that you cannot execute programs while in the C editor. You can only execute from the interpreter.

Throughout the rest of this manual, the C editor will be referred simply as an “editor”.

Note

The only files that can be handled by the editor are those with the identifier “C”.

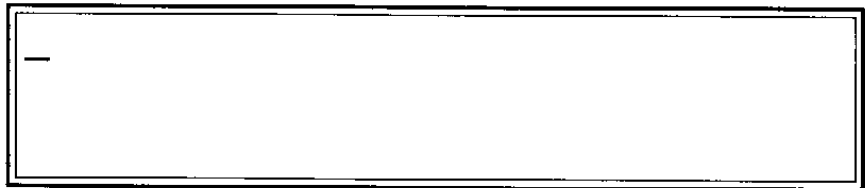
2-2 Activating the C function

Besides its C function, the PB-2000C also features calculator, formula memory, and data bank functions. Here we will see how to select the C function.


Important: Set the ROM card lock into the LOCK position when using built-in C even if the dummy card is not loaded in the computer.

To activate the C function

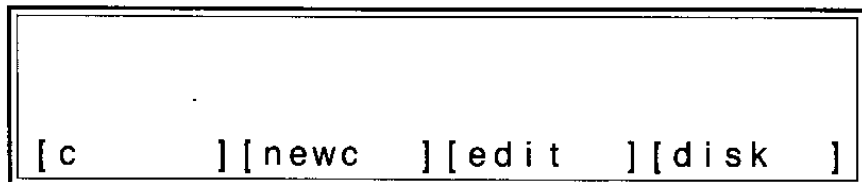
1. Switch the power of the PB-2000C ON.



This display indicates that the computer is in the CAL mode. This is the initial mode when you switch the computer ON, unless you have changed the initial mode to another using [set]. See the separate Owner's Manual for details.

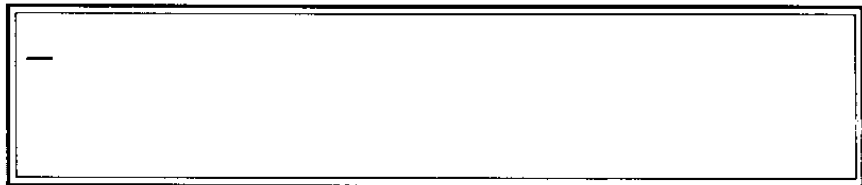
2. Press the  key.






3. Press the function key under [c] in the function key menu at the bottom of the display.

[c]

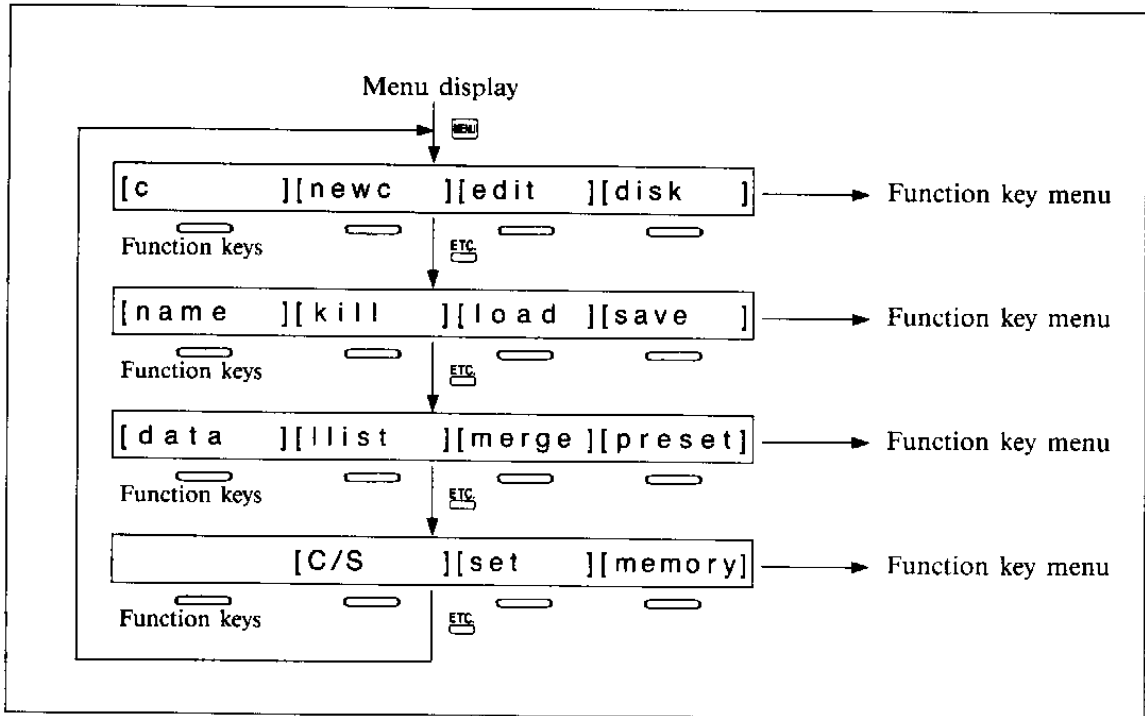


The computer is now in the interpreter, in which you can execute C commands and programs.

About the function key menus

The PB-2000C provides sets of function key menus to make operations quick and easy. Each function key menu includes four keys, and you can switch menus by pressing the  key. To execute any of the functions shown in the menu, simply press the function key underneath the menu selection you wish to select.

Menu display



Function key menu commands used in C

The function key menus make such operations as program file selection and entering the editor quick and easy. Press the **MENU** key to return to the first function key menu.

MENU

```
[c ][newc ][edit ][disk ]
```

You can select any of the currently displayed menu selections by pressing the function key below the function you wish to execute.

The following are descriptions of each of the 15 selections available in the C function of the PB-2000C. In each case, the currently selected file refers to the file whose name is highlighted on the display when a function is executed.

```
CASIO . S EXAMPLE1. S
ADDRESS . S math . C
INV . C
[c ][newc ][edit ][disk ]
```


[c]

Press the function key under this selection to enter the interpreter.

[newc]

Press the function key under this selection to create a new, unnamed C file. If an unnamed C file already exists in memory, the computer will enter the editor for that file.

[edit]

Press the function key under this selection to enter the editor or DATA EDIT mode for the currently selected file. For details on the DATA EDIT mode, see the Owner's Manual.

[disk]

This menu selection is only used when the optional MD-100 interface unit is connected to the PB-2000C. Press the function key under this selection to display the disk menu for the handling of files stored on the floppy disk currently loaded into the drive.

[name]

Press the function key under this selection to change the name of the currently selected file.

[kill]

Press the function key under this selection to delete the currently selected file.

[load]

Press the function key under this selection to load data or a program from a floppy disk or an external storage device. See the Owner's Manual for details.

[save]

Press the function key under this selection to save data or a program from the memory of the PB-2000C to a floppy disk or an external storage device, or to send data via the RS-232C interface. You can also make copies of files in the memory of the PB-2000C by saving from a file to another file with a different name. See the Owner's Manual for details.

[data]

Press the function key under this selection to enter the DATA EDIT mode to edit a sequential data file. See the Owner's Manual for details.

[list]

Press the function key under this selection to output the contents of a file to the printer. See the Owner's Manual for details.

[merge]

Press the function key under this selection to merge a file from a floppy disk or an external storage device to the currently selected file in PB-2000C memory. See the Owner's Manual for details.

[preset]

Press the function key under this selection to specify and cancel preset files. See the Owner's Manual for details.

[C/S]

Press the function key under this selection to change the identifier of the currently selected filename. A "C" identifier indicates a C file, while an "S" identifier indicates a sequential file. This selection will not appear if there are two unnamed files currently stored in memory, one with a "C" identifier, and the other with an "S" identifier.

[set]

Press the function key under this selection to display the current operational status of the computer. You can also make various changes in the operational status to suit your individual needs. See the Owner's Manual for details.

[memory]

Press the function key under this selection to check the memory status of the computer. See page 25 for details.

2-3 Interpreter displays

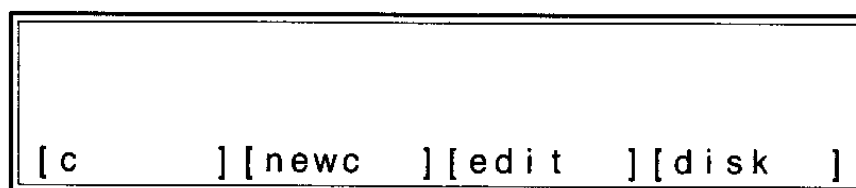
Interpreter

The interpreter is used to execute C commands and programs.

To enter the interpreter

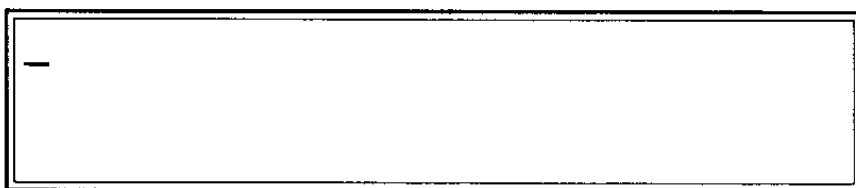
1. Press the  key.





2. Press the function key under [c] to enter the interpreter.

[c]



The display appears as illustrated above when you enter the interpreter. The cursor is flashing in the upper left corner of the display waiting for input.

Using command-line operation

The line in which the cursor is flashing on the display is called the *command line*. Each time you press a letter, number or punctuation key, the corresponding character appears at the current cursor location and the cursor moves to the right. You can enter any of the following seven commands directly into the command line to perform the functions noted.

1. LOAD — loads a program
2. RUN — executes the currently loaded program
3. NEW — deletes the currently loaded program
4. FLIST — displays the filename of the currently loaded program
5. EDIT — transfers to the editor
6. TRON — enters the TRACE mode
7. TROFF — exits the TRACE mode

Unlike BASIC and Prolog, C programs cannot be executed from the command line. You must first create a program using the editor.

To create a program

Let's create a simple C program. Follow the procedure outlined below.

1. Enter the word EDIT from the keyboard, and press **EXE**. The display will appear as illustrated below.

EDIT **EXE**

```
█  
  
[ c ] < 1 >
```

This is the editor display. Let's create a simple program that displays the word "HELLO!".

2. Enter "main()" from the keyboard and press **EXE**.

CAPS main() **EXE**

```
main(  
█  
  
[ c ] < 2 >
```

Note that the < 1> in the lower right of the display changed to < 2>. This value indicates the line in which the cursor is currently located.

3. Enter { (**SHIFT** **Y**) and press **EXE**.

SHIFT **Y** **EXE**

```
main(  
{  
█  
  
[ c ] < 3 >
```

4. Press **SPC** **SPC**, enter printf ("HELLO!");, and press **EXE**.

SPC **SPC** printf
(" **CAPS** HELLO!");
EXE

```
{  
printf("HELLO!");  
█  
  
[ c ] < 4 >
```

5. Enter) (SHIFT U) and press EXE.

SHIFT U EXE

```
printf("HELLO!");
}
[c ] < 5>
```

This gives us the following C program:

```
main()
{
    printf("HELLO!");
}
```

Now let's execute our program.

To execute a program

1. Press the function key under }c] to go from the editor to the interpreter. At this time, the display should appear as illustrated below.

[c]

```
NEW
LOAD""
Ready
```

2. Enter RUN and press EXE to actually execute the program.

RUN EXE

```
RUN
HELLO!
Ready
```

Success! Your computer has said "HELLO!" to you. Now it is time to store the program in a file in case we need it later.

To store a program in a file

1. Press the **MENU** key.

MENU

```
██████████ . C
[ c      ][newc  ][edit  ][disk  ]
```

Note that your program is actually already saved, but as an unnamed file.

2. Press **ETC** so that the **[name]** appears in the function key menu.

ETC

```
██████████ . C
[ name   ][kill  ][load  ][save  ]
```

3. Press the function key under **[name]** to change the name of our program.

[name]

```
██████████ . C
new name?_
```

4. Enter the name "main.c" in answer to the prompt, and press **EXE**.

main.c **EXE**

```
main .c C
[ name   ][kill  ][load  ][save  ]
```

2-4 Handy programming shortcuts

This section includes some valuable information on functions of the PB-2000C that will make your C programming quicker and easier.

Using the RECALL function

The recall function lets you instantly recall the last string of characters entered at the touch of a single key in the interpreter. This makes it easy to enter a series of identical or similar commands with fewer keystrokes.

To use the RECALL function

MEM

```
main .c C
[c ][newc ][edit ][disk ]
```

[c]

```
—
```

ANS **EXE**

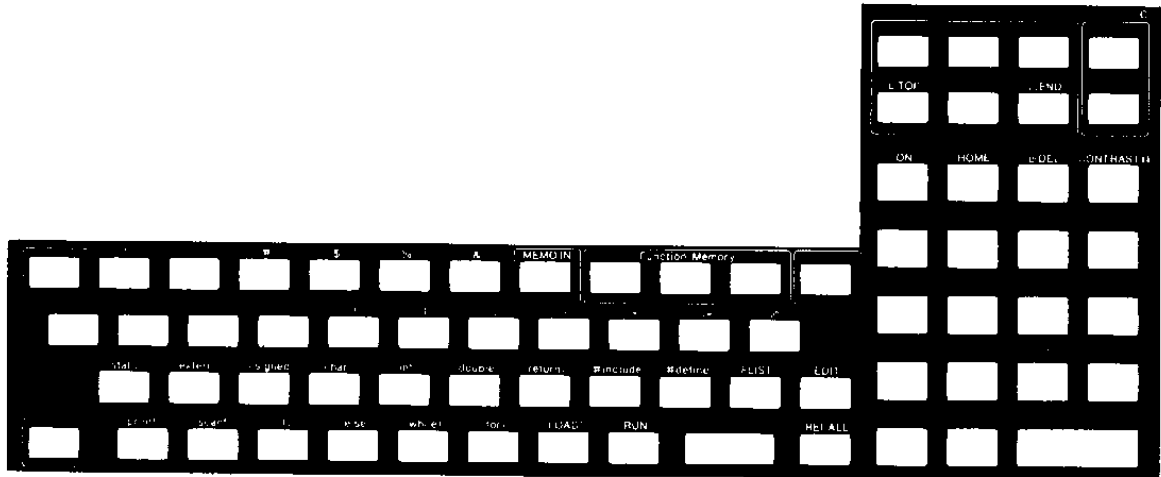
```
RUN
HELLO!
Ready
—
```

Using one-key commands

Programming is kept as simple as possible with the PB-2000C thanks to *one-key commands*. You can enter entire commands by simply pressing the applicable key following the **MEM** key. This not only makes input easier, it also cuts down on the chance of input errors for more efficient programming.

In order to help you keep track of the one-key commands, an overlay sheet is provided with the PB-2000C. Place the sheet on the keyboard during programming and you will be able to see the command assigned to each key.

Overlay sheet



Let's have a look at a simple example.

To use one-key commands

- 1. Press **SHIFT**, followed by **Z**.

SHIFT **Z**

```
printf(█
```

'printf(' immediately appears on the display because it is the one-key command assigned to the **Z** key.

- 2. Press **SHIFT**, followed by **B**.

SHIFT **B**

```
printf(while(█
```

This combination of commands actually does not have any meaning, we are just using it to illustrate the use of one-key commands.

Just as with the RECALL function, the 19 one-key commands of the PB-2000C make input easier while cutting down on the chance of input errors.

2-5 Using the editor

The editor of the PB-2000C is used for creating and editing programs, as we saw with our "HELLO!" program. This section takes you further along the path of using the editor effectively to create and edit useful C programs.

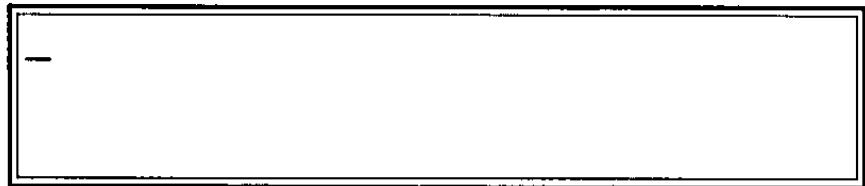
Entering the editor

You can enter the editor either from the MENU mode (by pressing the function key under [edit] or [newc]) or from the interpreter (by entering the EDIT command).

To create a new program from the MENU mode

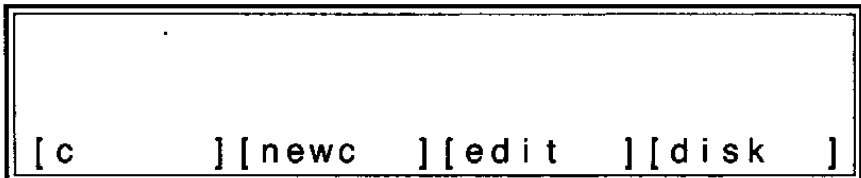
1. Confirm that the computer is in the CAL mode. If it isn't, press the **[CAL]** key or switch the power of the computer OFF and then ON again.

[CAL]



2. Press the **[MENU]** key to enter the MENU mode.

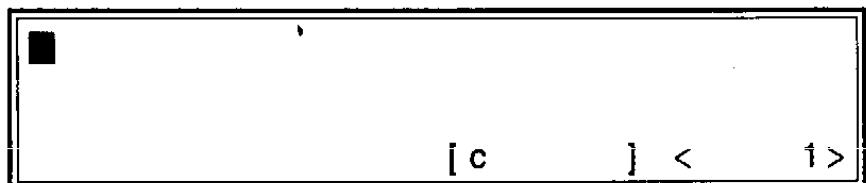
[MENU]



The display will appear as illustrated above when there are no files contained in memory.

3. Press the function key under **[newc]** to enter the editor. The block cursor blinks at the upper left of the display as the computer waits for your next input.

[newc]



The **[newc]** function is used when you wish to create or edit an unnamed file. After you create a new file, it is stored as an unnamed file, and you must use the **[name]** function to give it a name.

To enter an existing program from the MENU mode

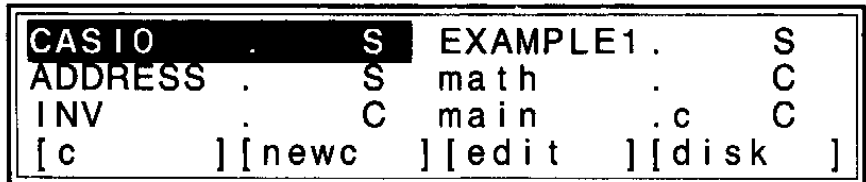
1. Confirm that the computer is in the CAL mode. If it isn't, press the **[CAL]** key or switch the power of the computer OFF and then ON again.

[CAL]



2. Press the **[MENU]** key to enter the MENU mode.

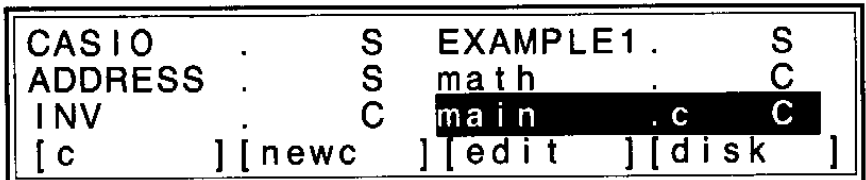
[MENU]



This display shows the names of all of the program files stored in the memory of the PB-2000C. The filename that is highlighted is the name of the currently selected file.

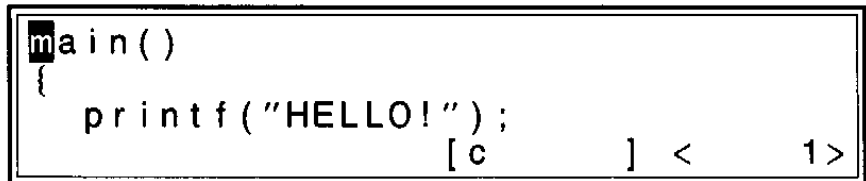
3. Use the cursor keys to change the currently selected file to "main.c".

[↓] [↓] [→]



4. Press **[edit]** to enter the editor for "main.c".

[edit]



To enter an existing program from the interpreter

1. Confirm that the computer is in the CAL mode. If it isn't press the **CAL** key or switch the power of the computer OFF and then ON again.

CAL



2. Press the **MENU** key to enter the MENU mode.

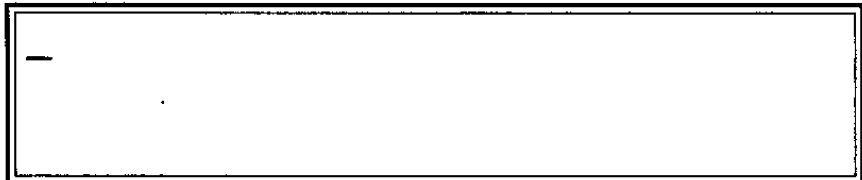
MENU

CASIO	.	S	EXAMPLE1.	S			
ADDRESS	.	S	math	C			
INV	.	C	main.c	C			
[c]	[newc]	[edit]	[disk]

This display shows the names of all of the program files stored in the memory of the PB-2000C.

3. Press the function key under **[c]** to enter the interpreter.

[c]



4. Manually enter EDIT"main.c" from the keyboard, and press **EXE**.

EDIT

"**CAPS** main. c" **EXE**

```





main( )
{
    printf("HELLO!");
}
[c ] < 1>

```

- If you do not include a filename after the EDIT command, the editor for the currently loaded file or the last file interrupted by an error will be entered.
- For further details on the EDIT command, see the command reference in Part 2 of this manual.

Moving the cursor

You can control cursor movement using the cursor keys in the editor as follows:

-  Cursor moves up
-  Cursor moves down
-  Cursor moves left
-  Cursor move right

Moving the cursor to the beginning of a line

Use the following operation to move the cursor to the beginning (left end) of the current line while in the editor:





Moving the cursor to the end of a line


Use the following operation to move the cursor to the end (right end) of the current line while in the editor:



Scrolling the screen

- Press the  cursor key while the cursor is at the top line of the display to scroll the display downward.
- Press the  cursor key while the cursor is at the bottom line of the display to scroll the display upward.

Editing a C program

To edit a C program, you must first enter the editor. There are two modes for editing with the editor, the INSERT mode and the OVERWRITE mode. Switch between these two modes by pressing the  key.

The INSERT mode

The INSERT mode is automatically specified when you enter the editor. The INSERT mode is indicated by a cursor that looks like “■”. While the computer is in the INSERT mode, any characters that you enter from the keyboard are inserted between any characters already on the display.

The OVERWRITE mode

You can specify the OVERWRITE mode by pressing the **[ms]** key. The OVERWRITE mode is indicated by a cursor that looks like “_”. While the computer is in the OVERWRITE mode, any characters that you enter from the keyboard replace any characters already on the display at the current cursor position.

Search function

The editor's search function lets you search through programs for specific character strings. We will demonstrate the search function using the following display.

[edit]

```
main()
{
    printf("HELLO!");
    [c      ] < 1>
```

1. Press the **[ETC]** key to advance to the next function key menu.

[ETC]

```
main()
{
    printf("HELLO!");
    [search][next ][delete] < 1>
```

2. Press the function key under **[search]**.

[search]

```
main()
{
    printf("HELLO!");
    search?_ < 1>
```

3. Enter the character string that you want to search for.

[CAPS] pr

```
main()
{
    printf("HELLO!");
    search?pr_ < 1>
```

4. Press the `EXE` key.

`EXE`

```

printf("HELLO!");
}
[search][next ][delete] < 3>
    
```

The specified character string is located and displayed, with the cursor located at the first letter of the string. The number in the lower right of the display indicates the program line number.

5. Press the function key under `[next]` to display the next occurrence of the specified string.

Exiting the editor and loading an edited file

You can exit the editor using one of two different methods:

1. By entering the interpreter (simultaneously executing the currently loaded program).
2. By returning to the MENU mode.

To exit the editor by entering the interpreter

You cannot directly execute a program while in the editor. You must first *load* the file from the editor into the interpreter, and then execute it. We will demonstrate using the following display.

```

main()
{
printf("HELLO!");
[c ] < 1>
    
```

1. Press the function key under `[c]` to enter the interpreter, and to simultaneously execute the commands `NEW` and `LOAD"main.c"`.

`[c]`

```

NEW
LOAD"main.c"
Ready
_
    
```

2. Enter the RUN command followed by **EXE** to execute the program "main.c".

RUN **EXE**

```
RUN
HELLO!
Ready
_
```

To exit the editor by returning to the MENU mode

1. Press the **MEN** key to exit the editor and return to the MENU mode.

MEN

```
CASIO . S EXAMPLE1. S
ADDRESS . S math C
INV . C main.c C
[c ][newc ][edit ][disk ]
```

2-6 Checking and specifying memory status

The PB-2000C lets you check and specify the sizes of various memory areas.

To check the memory status

1. In the MENU mode, press the **ETC** key until the display appears as illustrated below.

ETC

```
[C/S ][set ][memory]
```

2. Press the function key under **[memory]** to see the memory status display.

[memory]

```
<memory> c file work
26111 20992 4095 1024
free 1011 900
c,file?20992, 4095
```

The second line of this display shows the total user area capacity (in bytes), as well as a breakdown (from left to right) of the memory capacities of the C area, file area, and work area. The third line shows the amount of memory that is yet unused (free) in the file and work areas.

3. Press the **EXE** key to proceed to the C area memory status display.

EXE

<c>	code	symbol	stack
20992	5248	5248	10496
free	5138	5032	10488
code, symbol?_5248, 5248			

The second line of this display shows the total C area capacity (in bytes), as well as a breakdown (from left to right) of the memory capacities of the code area, symbol area, and stack area. The third line shows the amount of memory that is yet unused (free) in the code, symbol and stack areas.

To specify the memory status

1. Use the cursor keys to move the cursor left and right between the C area and file area values on the bottom of the general memory status display, and make changes in the values by entering values.
2. Press the **EXE** key to redisplay the general memory status display again with the newly changed values. At this time, you can make further changes as in Step 1, above.
 - Whenever you change the C area value, the C area contents are cleared.
 - The file area value cannot be made less than that required for storage of files currently in memory.
 - The size of the C area cannot be less than 4,096, and that of the work area cannot be less than 256.
 - All variables currently being used in the CAL mode are deleted when any change is made in the general memory status display.
3. Press the **EXE** key to proceed to the C area memory status display.

EXE

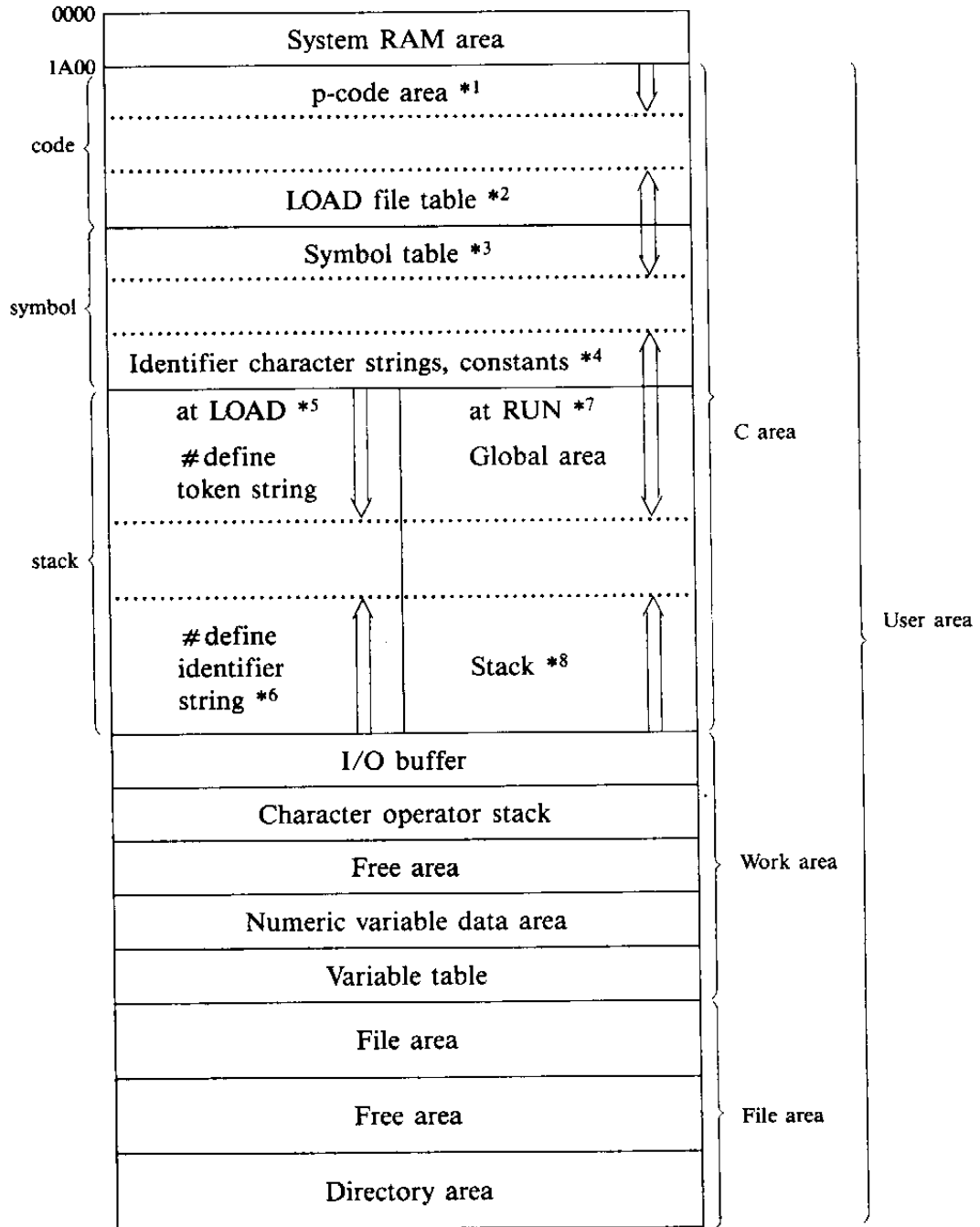
<c>	code	symbol	stack
20992	5248	5248	10496
free	5138	5032	10488
code, symbol?_5248, 5248			

4. Use the cursor keys to move the cursor left and right between the code area and symbol area values on the bottom of the display, and make changes in the values by entering values.
 - Whenever you change a value in this display, the C area contents are cleared.
 - The size of the code area and symbol area cannot be less than 512, and that of the stack area cannot be less than 1024.
5. Press the **EXE** key to redisplay the C area memory status display again with the newly changed values. At this time, you can make further changes as in Step 4, above.
6. Press **EXE** to return to the main menu display.
 - The following table shows the default values for the various memory areas following the NEWALL operation.

Memory area	32K byte RAM	64K byte RAM
User area (fixed)	26111	58879
work	1024	1024
file	4095	8191
C	20992	49664
code	5248	12416
symbol	5248	12416
stack	10496	24832

- The minimum value for the C area is 4,096 bytes.
- The work area requires at least 256 bytes.
- When you change the capacity of the C area, all of its current contents are cleared.
- When you change any of the values in the C area status display, all of the current C area contents are cleared.
- When you change the C area value, memory is reserved in the following ratio:
code=1 : symbol=1 : stack=2.
- The file area value cannot be made less than that required for storage of files currently in memory.
- All variables currently being used in the CAL mode are deleted when any change is made in the general memory status display.
- A "BS error" occurs if area values are outside of their specified ranges.
- The code area requires at least 512 bytes.
- The symbol area requires at least 512 bytes.
- The stack area requires at least 1,024 bytes.

Memory Map



Memory contents

***1 p-code area**

Statements of C program loaded using the LOAD command, stored as p-code. (2 bytes/1 token)

***2 LOAD file table**

Filename loaded by the LOAD command and filename specified by the #include. (16 bytes/1 filename)

***3 Symbol table**

Declaration portion of C program loaded using the LOAD command, stored as a symbol table. (17 bytes/1 symbol table)

***4 Identifier character strings, constants**

Identifier string and constants of C program loaded using the LOAD command. (Variable length)

***5 #define token string**

All #define token strings in C program loaded using the LOAD command. (2 bytes/1 token, initialized for each execution of LOAD)

***6 #define identifier string**

All #define identifier strings in C program loaded using the LOAD command. (Variable length, initialized for each execution of LOAD).

***7 Global area**

Reserved when the RUN command is executed as an allocated area for C program global variables and memory functions.

***8 Stack**

Reserved when the RUN command is executed as a parameter area for C program local variables and functions.

2-7 Using the LOAD command

The interpreter of the PB-2000C includes a function that is equivalent to separate compilation. This function is executed by the LOAD command.

Creating a file

First, let's create two files, one named "main.c" and the other names "sub.c". The following two displays show the contents of each file.

main.c

```
main()
{
    printf("Everybody!\n");
    sub();
}
```

sub.c

```
sub()
{
    printf("How are you?\n");
}
```

Enter the contents of each file using the following procedures.

main.c

Input: main () EXE
 { EXE
 SPC SPC printf("Everybody!\n"); EXE
 SPC SPC sub () ; EXE
 } EXE

```
sub();
}
[ c ] < 6 >
```

Now give the file a name and save it in memory.
Press **MENU**, **ETC**, and then the function key under **[name]**.

MENU **ETC**
[name]

```

main .c C
new name?_

```

The computer waits for you to enter a name.

Input: main. c **EXE**

```

main .c C
[name ][kill ][load ][save ]

```

Now let's input the second program.

sub.c

Input: sub() **EXE**
 { **EXE**
SPC **SPC** printf(" How are you? \n"); **EXE**
 } **EXE**

```

printf("How are you?\n");
}
[c ] < 5>

```

Now give the file a name and save it in memory.
Press **MENU**, **ETC**, and then the function key under **[name]**.

MENU **ETC**
[name]
sub.c **EXE**

```

main .c C sub .c C
[name ][kill ][load ][save ]

```

The two filenames now appear in the file menu.

Loading and executing programs

Here we will execute the LOAD command in the interpreter and load both of the files. Then we will execute the RUN command to execute both programs.

```
main .c C sub .c C
[c ][newc ][edit ][disk ]
```

Press the function key under [c].

Input: [c]
new **EXE**

```
new
Ready
_
```

This operation deletes an program currently stored in the interpreter and waits for input.

Input: LOAD "main.c" **EXE**

```
Ready
LOAD"main.c"
Ready
_
```

The computer loads the file "main.c" into the interpreter and awaits further input. Here, let's load the file "sub.c".

LOAD "sub.c" **EXE**

```
Ready
LOAD"sub.c"
Ready
_
```

The computer loads the file "sub.c" into the interpreter and awaits further input. Now let's execute the two files.

RUN **EXE**

```
Everybody!
How are you?
Ready
_
```

As the programs are executed the word “Everybody!” appears on the first line and “How are you?” appears on the second line. Let’s go back to our programs to find out why these words appeared as they did.

main.c

This program contains the standard library function `printf()`. A *standard library function* is a function that is built into C to perform a specific task (see Chapter 3 for further details). In this case, the `printf()` function outputs the string of characters that is included inside of its parentheses.

Note also that the string of characters is included inside of a set of double quotation marks. At the end of the string is “\n”. This tells the computer to include a carrier return at the end.

sub.c

This program is much the same as “main.c”. The string “How are you?” is produced by the `printf()` standard library function, and then a carrier return is output by “\n”.

2-8 Execution using batch files

A *batch file* lets you arrange a series of `LOAD` commands followed by a `RUN` command so that all of the filenames specified inside of the batch file are loaded and executed each time you execute the batch file.

Here, let’s create a batch file that loads and executes the “main.c” and “sub.c” files that we created previously. The program contained in the batch file will be the following:

```
NEW
LOAD“main.c”
LOAD“sub.c”
RUN
```

We will start from the MENU mode.

```
main .c C sub .c C
[c ][newc ][edit ][disk ]
```

To create a batch file

1. Press the function key under [newc] to enter the editor.

[newc]

```

[ c ] < 1 >
    
```

2. Enter the program as follows:

NEW [EXE]
 LOAD "main.c" [EXE]
 LOAD "sub.c" [EXE]
 RUN [EXE]

```

LOAD "sub.c"
RUN
[ c ] < 5 >
    
```

3. Now, press the [MENU] key to enter the MENU mode and then press [ETC] so that the following display appears.

[MENU]
 [ETC]

```

main .c C sub .c C
[ name ] [ kill ] [ load ] [ save ]
    
```

The highlighted file is the unnamed file that we have just created.

4. Name the file "greeting.BAT", and press [EXE]. Note that "BAT" must be all uppercase letters.

[name]
 [CAPS] greeting.
 [CAPS] BAT

```

main .c C sub .c C
new name?greeting.BAT_
    
```


EXE

```
main .c C sub .c C
greeting.BAT C
[name ][kill ][load ][save ]
```

Important

- The identifier for batch files must always be "C". The "C" identifier is assigned automatically when a new file is created using [newc].

Executing a batch file

There are a variety of ways to execute a batch file. We will describe all of them here, starting with the simplest method.

To execute a batch file from the MENU mode

1. Ensure that the filename menu is shown on the display and that the batch file filename is highlighted.

```
main .c C sub .c C
greeting.BAT C
[c ][newc ][edit ][disk ]
```

2. Press **EXE**.

This executes the file "greeting.BAT" and scrolls the display as each message appears. The following shows all of the messages that appear and scroll through the display.

```
NEW
LOAD"main.c"
LOAD"sub.c"
RUN
Everybody!
How are you?
Ready
—
```

To execute a batch file from the editor

1. While in the editor, press the function key under [c]. This will execute the batch file as long as its extension is "BAT".

To execute a batch file at power ON

You can specify a batch file to execute automatically each time you switch the power of the computer ON. All you have to do is name the desired file "AUTO.EXE".

```
main .c C sub .c C
AUTO .EXE C
[c ][newc ][edit ][disk ]
```

To execute a batch file as a preset file

First we have to specify the batch file as a preset file.

1. Ensure that the filename menu is shown on the display and that the batch file filename is highlighted.

```
main .c C sub .c C
greeting.BAT C
[c ][newc ][edit ][disk ]
```

2. Press **ETC** twice.

ETC **ETC**

```
main .c C sub .c C
greeting.BAT C
[data ][llist ][merge ][preset]
```

3. Press the function key under **[preset]** to change "greeting.BAT" to a preset file.

[preset]

```
main .c C sub .c C
greeting.BAT*C
[data ][llist ][merge ][preset]
```

Note that an asterisk appears before the identifier. This means that this file is a preset file. This completes the procedure to specify a file as a preset file, but let's continue and see how preset files are executed.

4. Press **CAL**.

CAL

```

_
[ greet i ]

```

This puts the computer into the CAL mode (see the Owner's Manual for details). Note that the name of the preset file that you have just created is shown at the bottom of the display as a preset filename menu selection.

5. To execute the "greetings" file, press the function key under **[greeti]**.

- You can have up to four preset files at any one time.
- Only the first six characters of the filename appear in the preset filename menu of the CAL mode.
- You can use other files (non-batch files) as preset files. When you execute a preset file with a C identifier from the CAL mode, the computer enters the interpreter and loads the file. If the preset file has an "S" identifier, the computer enters the editor for editing of the file.

2-9 Using the trace function

The *trace function* of the interpreter executes programs statement-by-statement to let you see on the display the exact flow of all of the commands. This function comes in very handy when debugging programs or when you wish to see the operation of the control structure. When the trace function is activated, the computer is in the *TRACE mode*.

To illustrate operation of the TRACE mode, we will use the files "main.c" and "sub.c" which we have already created. After loading the two files in the interpreter, the display should appear as follows:

```

Ready
LOAD"sub.c"
Ready
_

```

To enter the TRACE mode

1. Enter the TRON command and press **EXE**. "TRON" stands for *trace on*.

TRON **EXE**

```
Ready
TRON
Ready
_
```

The computer is now in the TRACE mode. Next we will execute the two programs which have been loaded to see how the TRACE mode operates.

2. Enter the RUN command and press **EXE**.

RUN **EXE**

```
Ready
RUN
[main.c(3)]    printf("Everybody!
\n"); ?_
```

This display shows that execution is halted at Line 3 of the file "main.c". The question mark in the bottom line of the display indicates that the computer is asking if it should execute Line 3. Note the following:

↓ File name (line number)

```
[main.c(3)]    printf("Everybody!
\n"); ?_
                ↑
                Statement
                ↑
                Waiting for input
```

3. Press **EXE** to execute Line 3 of "main.c".

EXE

```
[main.c(3)]    printf("Everybody!
\n"); ?
Everybody!
[main.c(4)]    sub(); ?_
```

This display shows the result of the execution of Line 3 of "main.c", and stands by waiting for you to press **EXE** and execute Line 4.

4. Press **EXE** to execute Line 4 of "main.c".

EXE

```
Everybody!
[main.c(4)]   sub(); ?
[sub.c(3)]   printf("How are you
?\\n"); ?
```

After Line 4 of "main.c" is executed, the computer enters file "sub.c". Execution stops at Line 3 of "sub.c" and the computer waits for you to press **EXE**.

5. Press **EXE** to execute Line 3 of "sub.c".

EXE

```
?\\n"); ?
How are you?
Ready
_
```

The result of the execution of Line 3 of "sub.c" is displayed. Since this completes execution of both programs, the interpreter stands by waiting for your next input.

To interrupt execution in the TRACE mode

1. To interrupt execution of a program in the TRACE mode, press the **BRK** key.

BRK

```
RUN
[main.c(3)]   printf("Everybody!
\\n"); ?
Break ?_
```

2. At this point, you can do one of the following:

- Press **EXE** or **C** to resume execution in the TRACE mode.
- Press **BRK** or **A** to interrupt execution in the TRACE mode.

To exit the TRACE mode

There are four different methods that you can use to exit the TRACE mode.

- Enter the TROFF command and press **[EXE]**. “TROFF” stands for *trace off*.

TROFF **[EXE]**

```
Ready
TROFF
Ready
_
```

- Exit the interpreter by entering the editor or the MENU mode. This automatically exits the TRACE mode.
- Switch the power of the computer OFF. This automatically exits the TRACE mode.
- Press the **[N]** key while TRACE mode execution is interrupted after you press the **[BRK]** key (see above).

Chapter 3

Introduction to C

This chapter tells you about the important points and rules to remember when creating C programs. Simply work with the example programs presented in the chapter to become familiar with proper procedure. This chapter puts you on the road to becoming a C programmer.

3-1 Outputting characters

Creating a program to output character strings

We have already had some experience in Chapter 2 with a few simple programs that output single lines of characters to the display. Here, let's take this to the next logical step and see how to output multiple lines.

Before we begin with this exercise, be sure that you have assigned names to all of the files that we have created until now.

1. Press the function key under **[newc]** to enter the editor and create a new unnamed file.

```

[ c ] < 1 >

```

2. Enter the following:

```

main( ) EXE
{ EXE
SPC SPC printf("How are you?\n"); EXE
SPC SPC printf("Fine, thank you\n"); EXE
} EXE

```

```

printf("Fine, thank you\n");
}
[ c ] < 6 >

```

The display should appear as illustrated above when you complete entering the program. The following shows how the program is stored in the file.

```

main( )
{
printf("How are you?\n");
printf("Fine,thank you\n");
}

```


As you can see here, C programs are generally written using lower case characters. Also note the following characters in lines 1, 2 and 5 of the program:

```
main( )
{
.....
}
```

These are called the *function* that makes up the C program. The word “main” defines the function name, and the statements between the braces are actually executed. Actually you can assign your function any name you wish, but “main” is special in that the computer will always begin execution from the function named “main”.

Lines 3 and 4 contain two printf statements. The printf statement is actually a function of C language that is used to output characters to the display. Such functions are called *standard functions*. The character strings included within the parentheses following printf are what is output to the display. Note that a *character string* is defined as such by being included within double quotation marks. A character string or anything else that is included within the parentheses is called an *argument*.

The “\n” at the end of the two printf character strings is called a *newline* character, and it advances output to the left margin of the next line. Here the newline character is at the end of the string, but you can also include it within the string to tell the computer to go to the beginning of the next line.

Now let’s enter the interpreter to execute this program. To do this, press the function key under [c] while in the editor.

After you performed the procedure described above, the computer will enter the interpreter, load the unnamed file, and stand by for further input.

<pre>NEW LOAD"" Ready _</pre>	<p>← Initializes</p> <p>← Loads unnamed file</p> <p>← Stands by for input</p>
-------------------------------	---

Now enter the RUN command and press [EXE].

RUN [EXE]

<pre>How are you? Fine, thank you Ready _</pre>

The screen scrolls as results are output to the display, and the computer stands by for further input following execution.

Making your program easy to read

You have probably noticed by now that we have been writing and editing our C programs in a certain format. The previous program was written and entered as:

```
main( )
{
    printf("How are you?\n");
    printf("Fine, thank you\n");
}
```

We could have just as easily written and input it as:

```
main( )
{
    printf("How are you?\nFine, thank you\n");
}
```

The computer would execute the program identically in either case. The former format is preferred, however, because it is much easier to read. Though you may not see the need for such a format at this point, but when you start dealing with longer programs you will greatly appreciate an easier to read format.

Creating a program to output numeric values

PB-2000C programs can output octal, decimal and hexadecimal values.

Here, let's create a program that displays the value 65 in its decimal, hexadecimal and floating point format, using a few more new techniques.

First, enter the editor and input the following:

```
/* Output Value */
main( )
{
    printf("D=%d H=%0x F=%f\n", 65, 65, 65.0);
}
```

```
65, 65.0);
}
[ c ] < 6 >
```

The following is the program list for our "Output Value" program.

```

/* Output Value */
main()
{
    printf("D=%d H=%x F=%f\n", 65, 65, 65.0);
}

```

The first line is what we call a *comment line*. The computer reads everything between `/*` and `*/` as a comment, and so they are ignored. You can use comment lines inside of programs to point out certain features, or as memos for later reference.

The remainder of the program is quite similar to the one that we created to output character strings to the display, but the argument for `printf` looks a bit different. The commas inside of the parentheses are there to separate arguments. This means that the `printf` statement in this program has a total of four arguments.

```

printf("D=%d H=%x F=%f\n", 65, 65, 65.0);

```

└─ 1st argument
└─ 2nd argument
└─ 3rd argument
└─ 4th argument

The arguments inside of the parentheses of `printf` tell the computer to print the following:

D= <65 as decimal integer> H= <65 as hexadecimal> F= <65.0 as floating point>

Note that there is a direct relationship with the 1st `%` construction and the 2nd argument, the 2nd `%` construction and the 3rd argument, and the 3rd `%` construction and the 4th argument. This sounds more confusing than what it actually is, and you might better understand it as illustrated below:

```

printf("D=%d H=%x F=%f\n", 65, 65, 65.0);

```

You just have to be careful to ensure that the proper `%` construction is matched with the proper value, or else you will get strange results. Here is a list of some other `%` constructions in addition to those noted above.

<code>%</code> construction	Output
<code>%d</code>	decimal integer
<code>%x</code>	hexadecimal integer
<code>%f</code>	floating point
<code>%s</code>	string
<code>%c</code>	single character

For further details, see the command reference in Part 2.

Now let's enter the `RUN` command to execute our program and look at the result.

`RUN` `EXE`

```

RUN
D=65 H=41 F=65.000000
Ready
_

```

Here we can see that the decimal equivalent of 65 is 65, the hexadecimal equivalent is 41, and the floating point equivalent is 65.000000.

Integer notation in C

In the previous program, the decimal value 65 was converted in accordance with three corresponding `%` constructions. In some instances, however, you might want to include the actual values as they are in the `printf` arguments, without conversion when they are displayed. To do this, you have to specify the value as an *integer constant*. With C, you can specify decimal, octal, and hexadecimal values, as well as characters as integer constants. In the case of a character, its 8-bit ASCII code is used when it is specified as an integer constant. Use the following formats to specify values or characters of integer constants.

- `65` — displayed as decimal integer 65.
- `0101` — displayed as octal value 101. Include 0 in the first digit of an octal value to specify it as an integer constant.
- `0x41` — displayed as hexadecimal value 41. Include `0x` or `0X` in the first two digits of a hexadecimal value to specify it as an integer constant.
- `'A'` — displayed as a single character. Include the character in single quotes.

If you go back to our last program and change the printf arguments as noted below:

```
main ( )
{
    printf ( "D=%d O=%o H=%x C=%c \n" , 65 , 0101 ,
            0x41 , 'A' );
}
```

The following results would be displayed:

```
D=65 O=101 H=41 C=A
```

3-2 Variable types and operations

Declaring variable types

With C programs, you have to declare the type of data that will be assigned to each variable before you can use that variable. The following statement, for example, tells the computer that the variable "x" will be used for the storage of integers:

```
int x;
```

The following table shows the other declarations that can be made for variables.

Declaration	Meaning
char	8-bit integer
short	16-bit integer
int	16-bit integer
long	32-bit integer
float	32-bit, single-precision floating point
double	64-bit, double-precision floating point

Besides these, you can also include the declaration "unsigned" in front of any of the integer declarations (except long) to indicate that the integer is unsigned (no positive/negative sign).

Assigning values to variables

To see how to declare variables and also how to assign values, let's write a program that performs various operations using the values 49 and 12.

Enter the editor and input the following program.

```
/* ARITHMETIC OPERATIONS1 */
main()
{
    int a,b,c,d,e;

    a=49+12;printf("%d  ",a);
    b=49-12;printf("%d  ",b);
    c=49*12;printf("%d  ",c);
    d=49/12;printf("%d  ",d);
    e=49%12;printf("%d\n",e);
}
```

The first four operations are addition, subtraction, multiplication and division. The value for 'e' will be the modulus (remainder) of 49 divided by 12.

When you execute the program, the display should appear as follows:

```
RUN
61  37  588  4  1
Ready
_
```

The following statement in the first line of the program declares that all five variables will be for the storage of integers.

```
int a, b, c, d, e;
```

As you can see, you can use a single declaration for multiple variables by separating the variables by commas.

In each of the following lines, the computer performs the calculation and assigns the result to the corresponding variable. Then the result is displayed by including the variable as an argument in the printf statement.

Using arrays

An *array* is a variable with depth. With an array, you use a single letter followed by a value to indicate the variable name. For example, a[0], a[1], a[2], a[3], a[4] represent five memory areas within an array called "a[5]". Note that the values following the letters must be enclosed within brackets.

With arrays, declaration of data type becomes very simple. The following statement at the beginning of a program declares an array from a[0] through a[4] for integers:

```
int a[5];
```

Let's go back to our original program where we used variables A through E to store calculation results, and use an array instead.

```
/* ARITHMETIC OPERATIONS2 */
main()
{
    int a[5];

    a[0]=49+12; printf("%d  ", a[0]);
    a[1]=49-12; printf("%d  ", a[1]);
    a[2]=49*12; printf("%d  ", a[2]);
    a[3]=49/12; printf("%d  ", a[3]);
    a[4]=49%12; printf("%d\n", a[4]);
}
```

Though the type of variable used is different, the results produced by this program are identical to those obtained by the previous program.

Note that an array may also have width. The array a[3][3] would represent as total of nine values:

```
a[0][0] a[1][0] a[2][0]
a[0][1] a[1][1] a[2][1]
a[0][2] a[1][2] a[2][2]
```

Array a[3][3] is called a 3 × 3 2-dimensional array.

3-3 Entering characters and values

Entering a single character from the keyboard

Here we will create a program that outputs a character and its corresponding character code in hexadecimal format. If you press the key for the letter "B", for example, the format will be:

```
Char=B Hex=0x42
```

The standard function `getchar()` is used to tell the computer to get one character input from the keyboard.

The following is the program that will accomplish our task:

```
/* #include "stdio.h" */
main( )
{
    int c;

    c=getchar( );
    printf("Char=%c Hex=0x%x\n", c, c);
}
```

The `getchar()` function only returns the character code, and so no argument is used inside of the parentheses. In the `printf()` function, `%c` specifies character format, while `%x` specifies hexadecimal format.

Try inputting and executing this program.

When you enter the RUN command, the interpreter executes the program until it comes to the `getchar()` function. At that time execution stops and the cursor flashes at the bottom of the display waiting for further input. At this time, you should enter a letter, number or symbol. The following shows the display which should result if you enter the letter "Q".

```
RUN [EXE]
Q [EXE]
```

```
Q
Char=Q Hex=0x51
Ready
_
```

This shows that the character code for "Q" is hexadecimal 51.

Entering values

Now let's try a program that calculates the sine and cosine of values that you enter from the keyboard. Since we can be expecting decimal values for input and output, we will be defining the variable as floating point. The program to accomplish this task would appear as follows:

```
/* #include "stdio.h" */
/* #include "math.h" */
main( )
{
    float x;
    printf("Input Value(DEG)");
    scanf("%f", &x);
    angle(0);
    printf("sin(%f)=%f\n", x, sin(x));
    printf("cos(%f)=%f\n", x, cos(x));
}
```

First we must specify the variable *x* as floating point.

The next line tells the computer to print the message "Input Value (DEG)" as a prompt. The `scanf()` function is used to get the entry of a value from the keyboard. Exactly the opposite of the `printf` function, the first argument of `scanf()` determines the type, and the value is assigned to the second argument. In the program here, we can see that the first argument of `scanf()` is `%f`, to indicate floating point. The second argument is `&x` which is a *pointer* which indicates an address in memory. The second argument of the `scanf()` function must be a pointer.

The statement "angle (0);" specifies the unit of angular measurement. You can specify the unit by changing the value following "angle" as follows:

- angle (0) — degrees
- angle (1) — radians
- angle (2) — grads

The next statements are the now-familiar `printf()`. Note that the 1st argument contains two `%f` (floating point) values which correspond to the 2nd argument (*x*) and the 3rd argument (`sin(x)` or `cos(x)`).

After you input this program, enter the interpreter and execute it. When the counter reaches the scanf() function, it displays the message "Input Value (DEG)" and waits for input of a value. Here, let's enter 60 and press the **EXE** key. This value is assigned to variable "x", and the following appears on the display.

RUN **EXE**
60 **EXE**

```
sin(60.000000) = 0.866025  
cos(60.000000) = 0.500000  
Ready  
_
```

Why don't you try modifying this program so that it calculates in radians or grads?

3-4 Using selection statements

Using the "if" selection statement

You can use the "if" statement to tell the computer to shift the flow of control if certain conditions are met. The "if" statement has two basic formats.

1. if (condition) statement

Here, the statement is executed if the condition is met (true=any value other than 0), and not executed if the condition is not met (false=0).

**2. if (condition) statement 1
else statement 2**

In this case, statement 1 is executed if the condition is met (true=any value other than 0), while statement 2 is executed if the condition is not met (false=0).

The if~else statement may contain multiple statements for the "if" and the "else" statements. In the following example, please also note the proper format for easy reading. Note that the statements are aligned and also note the position of the braces.

```
if (condition) {
    Statement 1
    Statement 2
}
else {
    Statement 3
    Statement 4
}
```

A program to solve quadratic equations

Here we will create a program that produces two solutions for the following quadratic equation:

$$ax^2 + bx + c = 0 \quad (a \neq 0)$$

In the above equation, it is assumed that a , b , and c are real numbers. The solution formula is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Also, the following is the discriminant which determines the following concerning the solutions of the above equation:

$$D = b^2 - 4ac$$

$D > 0$ — two different real numbers

$D = 0$ — multiple solutions (real numbers)

$D < 0$ — two different imaginary numbers (conjugate complex numbers)

In the program listed below, the discriminant is used in the condition of a selection statement, with one operation being performed if $D \geq 0$ and another being performed when $D < 0$.

```

/* #include "math.h" */
main()
{
    double a,b,c,d,q,r;
    scanf("%lf %lf %lf", &a,&b,&c);
    d=b*b-4.0*a*c;
    if (d>=0) {
        q=(-b+sqrt(d))/a/2.0;
        r=(-b-sqrt(d))/a/2.0;
        printf("%lf, %lf\n",q,r);
    }
    else{
        r=sqrt(-d)/a/2.0;
        q=-b/a/2.0;
        printf("%lf+%lfi",q,r);
        printf("%lf-%lfi\n",q,r);
    }
}

```

The variables "a", "b", and "c" correspond to the "a", "b", and "c" in the quadratic equation, while the "d" variable is the "D" of the discriminant. Variables "q" and "r" are used for the two solutions. Note that all of the variables are specified as "double", meaning that they are for double-precision floating point values.

Note that the % construction in the scanf() function is "lf%". The "f" portion specifies a floating point value, while the "l" stands for *long*, and means that the integer part of the value is longer than normal.

Relational operators

The conditions "d >= 0" is called a *relational operator*. This tells the computer to compare the value assigned to variable "d" with 0. If the value of "d" is greater than or equal to 0, a value of 1 is returned to indicate true. If it is less than zero, a 0 is returned to indicate false. The following are the other relational operators that can be used with C.

Operator	Example	Meaning
>	i > j	True (1) if i is greater than j, false (0) if not.
<	i < j	True (1) if i is less than j, false (0) if not.
>=	i >= j	True (1) if i is greater than or equal to j, false (0) if i is less than j.
<=	i <= j	True (1) if i is less than or equal to j, false (0) if i is greater than j.
==	i == j	True (1) if i is equal to j, false (0) if not.
!=	i != j	True (1) if i is not equal to j, false (0) if it is.

Enter the program and then go to the interpreter to load the file. Enter the RUN command and enter values for variables “a”, “b”, and “c”, pressing `EXE` after each entry. Here, we will use the following values as an example:

RUN `EXE` 1 `EXE` 5 `EXE` -14 `EXE`

```
-14
2.000000, -7.000000
Ready
_
```

We can confirm that these values are correct by performing the following manual calculation:

$$1 \times 2^2 + 5 \times 2 + (-14) = 0$$

$$1 \times (-7)^2 + 5 \times (-7) + (-14) = 0$$

For the next example, lets enter values that will produce imaginary numbers.

RUN `EXE` 2 `EXE` 3 `EXE` 4 `EXE`

```
-0.750000+1.198958i   -0.750000-1
.198958i
Ready
_
```

3-5 Using loops

Using the “while” loop

The “while” loop makes it possible to repeat execution of statements until a specific condition is met. The format of the “while” loop is as follows:

```
while (condition)
    statement
```

The “while” loop first executes the condition. If the condition is met (true, returning a value other than 0), the statment is executed, and execution loops back to the “while” loop to evaluate the condition again. Whenever the condition is not met (false, returning 0), execution of the “while” loop is ended.

You can have multiple statements in a “while” loop, but note the following format that you should use in order to make the program easier to read:

```
while (condition)    {
    statement 1
    statement 2
    :
    :
    statement n
}
```

The braces in the above list define the statements that are performed in the “while” loop, and so the closed brace should be directly under the “w” of the “while”. Keep the statements indented.

Now let’s create a program that uses the “while” loop. The program will output the character codes for the characters “0” through “Z”, as illustrated below.

```
Char(0) = Hex(0x30)
Char(1) = Hex(0x31)
Char(2) = Hex(0x32)
:
:
Char(X) = Hex(0x58)
Char(Y) = Hex(0x59)
Char(Z) = Hex(0x5A)
```

The following shows the program that produces such results.

```
#define STR '0'
#define END 'Z'

main()
{
    char ch;
    ch = STR;
    while(ch<=END) {
        printf("Char(%c)=Hex(0x%x)\n",ch,
            ch);
        getchar();
        ch++;
    }
}
```

Using the #define statement

Line 1 and Line 2 of the program contain the #define statement. The #define statement defines a *name* for a particular string of characters. In the above program “#define STR '0' ” tells the computer that anytime it comes across the name “STR”, it should replace it with the character “0”. Likewise, “#define END 'Z' ” tells the computer that the name “END” is to be replaced with the character “Z”.

The reason for using the #define statement is to make programs easier to read and to change later. If, in the above program, we used the character “0” a number of times throughout the program and we wished to change all of them to “A”, it would be much easier to change the #define statement once, rather than making multiple changes.

Incrementing and decrementing

The program here establishes the initial value for variable “ch” as “'0'”, and tells it to keep repeating the loop “while (ch <= END)”, in which END is a name that represents the character “Z”. As long as “ch” is less than or equal to “Z”, the following printf() statement is executed and then the getchar() statement is executed. This program does not really require input of any characters, but if we did not include a getchar() statement here the data produced by this program would simply scroll across the display so quickly that we would not be able to use it. The getchar() statement causes the computer to stop here and wait for input, so you can read the last line produced by the program. When you are ready to continue, simply press the **ENTER** key.

Once execution continues past the ch++; statement, 1 is added to the value of “ch”. This is accomplished by the statement “ch++” in which “++” is called the *increment operator*. The following table shows how it is used, as well as its opposite, the *decrement operator*.

Operator	Example	Meaning
++	++i	Increment i by 1 and use the value.
--	--i	Decrement i by 1 and use the value.
++	i++	Use i and then increment it by 1.
--	i--	Use i and then decrement it by 1.

As you can see here, the increment and decrement operators can be used either before or after the value, with different results. For other useful operators, see Chapter 5 of this manual. Now let's enter the program using the editor, and then execute it in the interpreter.

Using the “do ~ while” loop

The “do ~ while” loop is another method that you can use for repeat execution. The format of the “while” loop is as follows:

```
do
    statement
while (condition);
```

Unlike the “while” loop, the “do ~ while” loop executes the statement first and then checks whether or not the condition has been met or not. Note also that the semicolon at the end of the “while” line cannot be omitted.

Let’s use the “do ~ while” to find the Greatest Common Measure for two values.

```
main ( )
{
    int gcm, x, y;

    x = 56;
    y = 63;
    printf ( "\nGCM (%d , %d) = " , x , y );
    do {
        gcm = x;    x = y % x;    y = gcm;
    } while ( x != 0 );
    printf ( "%d\n" , gcm );
}
```

When you execute this program, the following result should be produced:

```
GCM (56, 63) = 7
```

Using the “for” loop

You can also use the “for” loop for repeat execution of statements. The format of the “for” loop is as follows:

```
for (expression 1; expression 2; expression 3)
    statement
```


Note that there are three expressions inside of the parentheses of the “for” loop. Expression 1 initializes the counter variable, and it is executed only once, before the first pass of the loop. Expression 2 is the condition of the loop, so the loop continues to execute until this condition is not met. Expression 3 performs an operation on the counter variable before the statement in the loop is executed. Note the following:

```
for (i=0; i<10; i++)
    printf( . . . . . );
```

This tells the computer to execute the `printf()` statement starting from a count of 0 which is incremented by 1 with each pass of the loop, until the count reaches 10.

As with the “if” and “while” loops, multiple statements within a “for” loop are enclosed in braces.

Let’s write a program that squares all of the integers from 1 through 100. Note the following list:

```
main ( )
{
    int i;

    for ( i=1; i<=100; i++)
        printf( "%8d %8d\n", i, i*i );
}
```

First, let’s look at the expressions contained in the parentheses following the “for” statement and see what each of them means.

expression 1	<code>i=1</code>	Initial value of counter is 1.
expression 2	<code>i<=100</code>	Repeat as long as <code>i</code> is less than or equal to 100.
expression 3	<code>i++</code>	Increment <code>i</code> by 1 with each pass of loop (<code>i=i+1</code>).

In effect, this tells the computer to keep repeating the loop starting with a counter value of 1, and repeat the loop as long as the counter value is 100 or less.

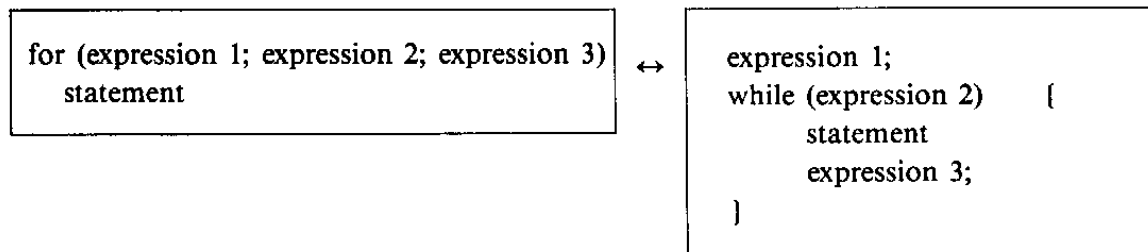
In the next line we have a `printf()` statement. The “%8d” in the first argument tells the computer to display the value flush right in an 8-character block reserved on the display. Flush left would be “%-8d”.

The `getchar()` statement causes the computer to stop and wait for input, so you can read the last line produced by the program. When you are ready to continue, simply press the **ENTER** key.

Enter this program, load its file into the interpreter and enter the `RUN` command to see how it works.

1	1
2	4
3	9
	:
	:
98	9604
99	9801
100	10000
Ready	

The following shows how the same loop would be written as a “for” loop and a “while” loop.



As you can see, the “for” loop is much easier to write and read.

Nested loops

The term *nested loop* means simply “loops inside of loops”. To better understand how nested loops work, let’s have a look at a very short, simple representative program.

```
main()
{
  float    a[3][3];
  int      i, j;

  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      scanf("%f", &a[i][j]);
}
```

The program uses a pair of “for” loops to read values from the keyboard and assign them to the 3×3 2-dimensional array `a[3][3]`. Remember that such an array looks something like the following:

<code>a [0] [0]</code>	<code>a [1] [0]</code>	<code>a [2] [0]</code>
<code>a [0] [1]</code>	<code>a [1] [1]</code>	<code>a [2] [1]</code>
<code>a [0] [2]</code>	<code>a [1] [2]</code>	<code>a [2] [2]</code>

Considering just the loop counter (`i` and `j`) values, the above program executes as follows:

First pass (<code>i</code>)	Second pass (<code>j</code>)	Third pass (<code>i</code>)
<code>i=0</code>	<code>i=1</code>	<code>i=2</code>
<code>j=0</code>	<code>j=0</code>	<code>j=0</code>
<code>j=1</code>	<code>j=1</code>	<code>j=1</code>
<code>j=2</code>	<code>j=2</code>	<code>j=2</code>

This means that values entered from the keyboard are read into “%f” by `scanf()` and assigned to array `a[3][3]` in sequence by “&a[i][j]” as “i” and “j” change values as noted above.

Let’s expand on this program so that a second nested loop reads the values out of the array and displays them on the screen.

```
main()
{
    float a[3][3];
    int i, j;

    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            scanf("%f", &a[i][j]);
    for (i=0; i<3; i++) {
        for (j=0; j<3; j++)
            printf("%8.2f", a[i][j]);
        printf("\n");
    }
}
```

The first line of the program declares a 3×3 2-dimensional array to hold floating point values. The next line specifies the values of “i” and “j” as integers.

Next, we have the value assignment nested loops we saw before, followed by a similar set of loops to read and display the values after they are stored. The only new item is the “%8.2f” which specifies that each value will be displayed in an area of at least 8 characters, with two decimal places (right flush).

Enter the program and then execute it. As an example, enter the following values as noted.

RUN

1 2 3 4 5 6 7 8 9

Scrolled off the display →

Last line →

1.00	2.00	3.00
4.00	5.00	6.00
7.00	8.00	9.00
Ready		
—		

Inputting and outputting character strings

Lets use a “while” loop to read input of a character string from the keyboard and then output the string to the display. This program will also limit input to 128 characters.

```

/* #include "stdio.h" */
#define LINESIZE 128
main()
{
    int i;
    char s[LINESIZE];

    i=0;
    while((s[i]=getchar())!='\n')
        i++;
    s[i]='\0';
    for (i=0; s[i]!='\0'; i++)
        putchar(s[i]);
}

```

Variable "s" is declared to be an array for storage of character strings, and the depth of the array is defined by LINESIZE, which is the name assigned to the value 128.

Let us now look inside of the parentheses of the "while" statement.

```
while ((s[i]=getchar( ))!='\n')
```

With the initial pass of the loop, the inner statement is executed first to assign a character entered from the keyboard to s[0]:

```
(s[i]=getchar( ))
```

Then the computer is instructed (by !=) to check whether or not the character assigned to the array is a newline character:

```
while ( ..... !='\n')
```

This gives us the entire "while" statement:

```
while ((s[i]=getchar( ))!='\n')
```

In review, it says "get a character from the keyboard and assign it to the array". Keep doing this unless the last character was a newline (**EXE**).

The "i + +" in the next line increments "i" so that the input progresses through the array (s[0], s[1], s[2], s[3], etc.).

When the **EXE** key is pressed, the computer detects a newline character and exits the "while" loop. The next line of the program replaces the newline character with 0, by:

```
s[i]='\0'
```

The "for" loop in the next line recalls the characters from the array and puts them on the display.

If you run this program and enter the word "Good", the display should appear as illustrated below:

```
RUN EXE  
Good EXE
```

```
Good  
Good  
Ready  
_
```

The following shows the contents of the array s[i] following the above execution.

Input	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
G	'G'	
o	'G'	'o'	
o	'G'	'o'	'o'	
d	'G'	'o'	'o'	'd'	.	.	.	
EXE	'G'	'o'	'o'	'd'	'\n'	.	.	
	'G'	'o'	'o'	'd'	0	.	.	

Standard functions

The *standard functions* have been preprogrammed into C to make various complex functions as easy as including the corresponding function within the program. The following table shows the most commonly used standard functions and their operation. See the function reference for further details.

Standard Function	Operation
putchar(c)	Outputs character c.
puts(s)	Outputs character string s.
printf(.....)	Outputs expression.
getchar()	Reads character from standard input.
gets(s)	Returns entered string into character string s.
scanf(.....)	Inputs expression
strlen(s)	Returns length of string s.
strcat(s1,s2)	Appends string s2 to string s1.
strcmp(s1,s2)	Compares string s1 with string s2.
strcpy(s1,s2)	Copies string s2 to string s1.
fopen(file,mode)	Opens a file.
fclose(filep)	Closes a file.

Now let's see what happens if we rewrite our previous program using the standard function `gets(s)` for input, and `put(s)` for output.

```
main( )
{
    char  str[128];

    gets(str);
    puts(str);
}
```

As you can see here, we can use standard functions to make our programs much simpler. For further details on using standard functions, see the function reference in Part 2 of this manual.

3-6 Defining functions

Function definitions and program modules

Besides the standard functions, the C programming language lets you define your own routines as functions to be called by later programs. We have seen a hint of this already in the "main()" first line of our programs to define them as main functions.

Using the procedures presented here, you will be able to break large programs down into *modules* and then call up each module from the main() function. This means that you will eventually build up your own library of functions that you can call up from various programs as you need them.

Creating functions

The following is the format for function definition:

```
function type declaration  function name (arguments)
argument type declaration;
{
    declaration of variables used in function;
    statements
    .....
}
```

function type

This part of the function declares the type of data that will be returned by the function. This line may be omitted if the value to be returned is an integer (int) or if there is no data returned.

function name

You cannot use reserved words as function names, and you cannot use names already used for standard function names. The following is a table to reserved words.

auto	(default)	float	register	struct	(volatile)
break	do	for	return	(switch)	while
(case)	double	goto	short	(typedef)	
char	else	if	(signed)	union	
(const)	(enum)	int	sizeof	unsigned	
continue	extern	long	static	void	

arguments

Arguments are used to pass values to the function when it is called. If no values are passed, the arguments can be omitted. If the arguments are omitted, the following argument type declaration is also omitted.

Argument type declaration

Declares the types of the arguments specified above.

Statements

The portion of the function between the braces is executed when the function is called. Use the same format as that which we have seen for our main() programs.

Sample functions

For illustrative purposes, lets write a program that squares all integers from 1 through 10, with the actual squaring calculation being performed by a function. We will write one program that returns integer values and another that returns floating point values.

Returning integers

```
main( )
{
    int    i;

    for(i=1; i<=10; i++)
        printf(" (%d)^2=%d\n", i, isquare(i));
}

isquare(x)
int x;
{
    return (x*x);
}
```


Returned values

The function `isquare` that we have defined receives the argument, squares it, and then returns the squared value. The line `return(x*x)` instructs the computer to square the value of `x` and return it to the `main()` function.

When the `main()` function executes `isquare(i)`, the value stored in variable `i` is passed to function `isquare` as argument `x`. The `return` statement in the `isquare` function then returns the squared value to `isquare(i)` in the `main()` function, and execution precedes to the `printf` statement for output.

This execution is repeated for values 1 through 10, and since this program declares variable `i` with `int` the returned value is treated as an integer.

Returning double precision values

```
main()
{
    double    d, dsquare();

    for(d=1.0; d<=10.0; d+=1.0)
        printf("(%.1f)^2=%.1f\n", d, dsquare(d));
}

double dsquare(x)
double x;
{
    return (x*x);
}
```

This program is identical to the previous one, except that variables are declared to be double precision. Always be careful to declare variables properly in order to ensure that your program produces the desired results.

Recursive function calls

The recursive capabilities of C functions mean that a function may call itself, either directly or indirectly. The following program uses a recursive function call to calculate the factorials of entered values, according to the following formulas:

$$n! = n * (n - 1)!$$

$$= n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$$

```
double fact();
main()
{
    int n;
    scanf("%d", &n);
    printf("%2d! = %e\n", n, fact(n));
}
double fact(x)          /* FACT function */
int x;
{
    if(x==0) return(1);
    return(x*fact(x-1));
}                          /* Recursive call */
```

The following shows a sample execution of the above program:

```
RUN 
20           ← Value input
20! = 2.432902e+18          ← Result
```

3-7 Local variables and global variables

Local variables

A *local variable* is one that is declared within a function for use within that particular function only.

Since such variables are local, you can use the same variable name in multiple functions without any problem — each variable is treated independently, even though they have the same name.

Have a look at the program listed below.

```
main()
{
    int i;
    for(i=0; i<10; i++)
        pr();
}
pr()
{
    int i;
    for(i=0; i<2; i++)
        printf("%d ", i);
}
```

This program displays ten sets of the numbers 0 and 1. Both the `main()` function and the `pr()` function utilize variable "i" but the computer treats these as two entirely different variables. In the `main()` function the value of "i" ranges from 0 through 9, while in `pr()` it is 0 or 1 only.

Global variables

A *global variable* is one that is commonly applied throughout all functions of a program. Global variables are declared in the line preceding the `main()` line. Have a look at the program listed below.

```
int i;
main()
{
    for(i=0; i<10; i++)
        pr();
}
pr()
{
    printf("%d", i);
}
```

Here, variable "i" is declared before the `main()` line, so "i" will be treated as a global variable. Consequently, it will retain its current value regardless of whether execution is in `main()` or `pr()`. In this program, variable "i" is assigned a value by the "for" loop in `main()`, and then the value of "i" is printed by function `pr()`.

As a general rule, it is recommended that you do not use global variables. This is because global variables restrict the flexibility of variables in general, and they make the program difficult to comprehend. If you do decide to use global variables, it is probably best to use long, descriptive names such as ‘array_x’, ‘xcord’ or ‘ycord’ to make obvious the type of value that is stored.

3-8 Pointers and variable storage locations

Entering values

Here we will write a calculation program that performs specific integer arithmetic operations on values entered via the `scanf()` standard function. With this program, entry of ‘33+21 **EXE**’ for example, would produce the result ‘=54’.

```
main ( )
{
    char   op;
    int    x, y, xy;

    xy = 0;
    scanf ( "%d%c%d" , &x , &op , &y ) ;
    if ( op == ' + ' ) xy = x + y ;
    if ( op == ' - ' ) xy = x - y ;
    if ( op == ' * ' ) xy = x * y ;
    if ( op == ' / ' ) xy = x / y ;
    if ( op == ' % ' ) xy = x % y ;
    printf ( "= %d \n" , xy ) ;
}
```

Variable ‘op’ is a character type, while ‘x’, ‘y’, and ‘xy’ are all integer type. Since we are using the `scanf()` function, arguments are expressed preceded by ampersands, becoming ‘&x’, ‘&op’, and ‘&y’. In C, an argument such as ‘&x’ represents the location in memory that the contents of variable ‘x’ are stored. It is called an *address*.

The `scanf()` function understands arguments to be addresses, and represents the value stored at the address within the expression. See the Command Reference for detailed information on the `scanf()` function.

The line ‘`scanf("%d%c%d", &x, &op, &y);`’ accepts input of integers for variables ‘x’ and ‘y’, and a character code for variable ‘op’.

The execution of the next statement depends on the characters assigned to variable ‘op’. If it is a plus sign, the first ‘if’ statement is executed, if it is a minus sign, the second ‘if’ statement is executed, etc.

Finally, the `printf()` statement displays the value of ‘xy’ which is the result, no matter which arithmetic operation is performed.

3-9 File input and output

Here we will write a program that writes the character string "abcdefg\nABCDEFG\n" to a file named "a.dat", and then displays the string, character-by-character.

```

/* #include "stdio. h" */
main ( )
{
    int c;
    FILE *fp;

    fp=fopen("a.dat","w");
    fprintf(fp,"abcdefg\nABCDEFG\n");
    fclose(fp);
    fp=fopen("a.dat","r");
    while((c=fgetc(fp)) !=EOF)
        putchar(c);
    fclose(fp);
}

```

The line "FILE *fp;" in this program declares "fp" points to a FILE.

Unlike standard C, the interpreter of the PB-2000C allows all standard functions and built-in functions to be used without declaration. Therefore, the fopen() statement which is generally required in the FILE declaration is not necessary with the PB-2000C.

The line "fp=fopen("a.dat", "w");" specifies a filename of "a.dat" and a mode of "w". Here, the "a.dat" file is a write file which will be written to via variable "fp". The following shows the three modes that can be specified.

- w — *write* from the beginning of the file, erasing any previous file contents.
- r — *read* from the beginning of the file.
- a — *append* at the end of a file

"fprintf" in Line 8 of the program writes a character string specified by the first argument to the file.

"fclose(fp)" in Line 9 of the program closes the open file.

Line 10 reopens the "a.dat" file, but this time in the read mode.

Lines 11 and 12 employ a "while" loop to read data via "fp" character-by-character from the file and assign it to "c".

The putchar(c) function displays the characters.

The loop is continued until EOF (end of file) is reached. At that time the "while" loop is exited and fclose(fp) is executed to close the file and end the program.

Chapter 4

Sample Programs

This chapter includes a number of short programs that show you the graphics and mathematical capabilities of the PB-2000C. Try out some of these programs and you gain valuable experience in programming debugging operations.

4-1 Prime numbers

This program displays all of the prime numbers for an entered value. The program does this by applying what is known as the "Eratosthenes method". The following shows some of the essential formats and their functions.

Format	Function
<pre>nc = ert(n, array); int nc, n; int array[];</pre>	<pre>nc = prime numbers in values less than n. Enters prime numbers into array. This program is limited to the following range: 1 < n < 2000.</pre>

```
main()
{
    int prime[1000];
    int i, n, nc;

    printf("N(2<N<2000)=?"); scanf("%d", &n);
    nc=ert(n, prime);
    printf("N prime=%d\n", nc);
    for(i=0; i<nc; i++)
        printf("%8d", prime[i]);
}

ert(n,p)                                /* Eratosthenes method */
int n,p[];
{
    int i,j,k,nn;

    nn=0; p[0]=2;
    for(i=1,j=3;j<=n; i++,j+=2)
        p[i]=j;
    for(j=1; j<=i; j++)
        if(p[j]){
            p[++nn]=p[j];
            for(k=j+p[j]; k<=i;k+=p[j])
                p[k]=0;
        }
    return(nn);
}
```

The following shows a sample execution.

Input	Display
RUN <input type="checkbox"/> EKE	N(2<N<2000) = ?_
10 <input type="checkbox"/> EKE	N prime = 4 2 3 5 7 Ready —

4-2 Memory display

This program displays the memory contents from a beginning address to an ending address, both entered from the keyboard. Memory contents are displayed in hexadecimal format. In this program, an integer is used instead of a pointer, but the integer is treated as a pointer thanks to a forced conversion known as a cast. If "x" is an integer variable, and "p" is a character type pointer, the following would substitute the value of "x" for pointer "p":

```
p=(char *)x;
```

In addition, the contents at address 0x4000 are assigned to pointer "c".

```
c=*(char *)0x4000;
```

The format "`*(char *)i&0xff`" stands for 1 byte of data at address "i". Bit AND is performed on 0xff to extract the lower 8 bits.

```
main()
{
    int i, st, ed;

    printf("Start address(hex)?");
    scanf("%x", &st);
        /* Entry of beginning address in hexadecimal format */
    printf("End address(hex)?");
    scanf("%x", &ed);
        /* Entry of ending address in hexadecimal format */
    for (i=st; i<ed; i++){
        if(i%8==0)printf("\n%4X|", i);
        printf("%02X ", *(char*)i&0xff);
        /* Address display */
        /* Contents display */
    }
}
```


The following shows a sample execution.

Input	Display
RUN <input type="button" value="EXE"/>	Start address(hex) ?__
2000 <input type="button" value="EXE"/>	End address(hex) ?__
2010 <input type="button" value="EXE"/>	2000 FF FF FF FF FF FF FF FF 2008 FF FF FF FF FF FF FF FF Ready —

Important

The actual display will differ in accordance with memory contents.

4-3 Perpetual calendar

With this program, you enter a month, date, and year, and the computer tells you what day of the week that particular date falls on. This is accomplished by assuming that January 1, 1987 falls on Thursday, with all other dates being determined on that basis. The following is a summary of the execution of the calendar program.

1. Computer initializes the days of the week and the number of days in each month.
2. Operator inputs month, date, and year.
3. Computer determines whether or not the entered year is a leap year, and calculates the number of days from January 1 to the specified date.
4. Computer determines what day of the week January 1 falls on for the specified year, and calculates the day of the week for the entered date.
5. Computer displays the day of the week for the specified date.

The following table shows the variables and functions used in this program.

Variables	Functions
x=wday (year);	Returns the day of the week that January 1 of "year" falls on (0=Sunday; 1=Monday; ... 6=Saturday).
l=leap (year);	Returns a leap year flag (0=normal; 1=leap year).

The following is a list of the calendar program.

```

#define INITYEAR 1987      /* Initial Year number for Calc. */
#define WDAY1987 4        /* Week day Number of 1987.1.1 */

char *wday[7];            /* Week day name */
int mo[12];               /* Days of month */

main()
{
    int year, month, day, wdayn, totalday=0, i;

    wday[0]="Sun"; wday[1]="Mon"; wday[2]
    ="Tue"; wday[3]="Wed"; wday[4]="Thu";
    wday[5]="Fri"; wday[6]="Sat";
    mo[0]=mo[2]=mo[4]=mo[7]=mo[6]=mo[9]=
    mo[11]=31;
    mo[3]=mo[5]=mo[8]=mo[10]=30;
    printf("Input YEAR ?"); scanf("%d",
    &year);
    printf("Input MONTH?"); scanf("%d",
    &month);
    printf("Input DAY ?"); scanf("%d", &day);

    if(leap(year)) mo[1]=29;
    else mo[1]=28;

    for(i=0; i<month-1; i++) /* Total days to month 1st */
        totalday+=mo[i];
    totalday+=day;          /* Total days from Jan. 1st */
    wdayn=(totalday+wday1_1(year)-1)%7;
                          /* Set weekday number */

    printf("%d.%d.%d=\ \"%s\" ", year, month, day,
    wday[wdayn]);
}

int wday1_1(year)          /* Return the week day
int year;                 number of Year of 1st */
{
    int y, wlliday;
    long tday =0;

```

```

    if (year >= INITYEAR) {
        for (y = INITYEAR; y < year; y++) {
            if (leap(y)) tday = tday + 366;
            else tday = tday + 365;
        }
        wlliday = (tday + WDAY1987) % 7;
    }
    else {
        for (y = INITYEAR - 1; y >= year; y--) {
            if (leap(y)) tday = tday + 366;
            else tday = tday + 365;
        }
        wlliday = ((tday + 6) / 7 * 7 + WDAY1987 - tday) % 7;
    }
    return ( wlliday );
}

leap(y) /* Return Leap Year (0 : not LEAP) */
int y;
{
    return ((y % 4 == 0 && y % 100 != 0) || y % 400 == 0);
}

```

The following shows a sample execution of the program.

Input	Display
RUN <input type="checkbox"/> EXE	Input YEAR ?__
1989 <input type="checkbox"/> EXE	Input MONTH ?__
6 <input type="checkbox"/> EXE	Input DAY ?__
19 <input type="checkbox"/> EXE	1989.6.19 = "Mon" Ready —

4-4 Sine curve/cosine curve program

The program presented in this section displays a sine curve and cosine curve in graphic form. The y-coordinate values for the sine curve and cosine curve are calculated according to the following formulas:



$$\begin{aligned} ys &= 15.5 - 15.0 * \sin(3.0 * x); && \text{sine curve coordinate values} \\ yc &= 15.5 - 15.0 * \cos(3.0 * x); && \text{cosine curve coordinate values} \end{aligned}$$

The x-axis is established at $y=15$ on the display, making this the central axis of the graph, with the minimum value being 0 and the maximum 30. To each of these values, 0.5 is added because C simply cuts off decimal portions when substituting integers for floating point values. Adding 0.5 makes it possible to obtain more accurate values through rounding.

```
main()
{
    int  x, ys, yc;

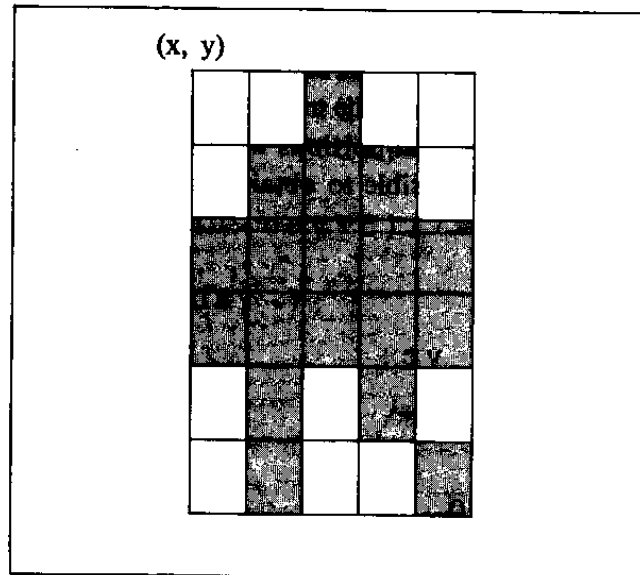
    angle(0);
    clrscr();
    line(10, 0, 10, 31);          /* Vertical axis */
    line(10, 15, 180, 15);       /* Horizontal axis */
    for(x=30; x<180; x+=30)
        line(x+10, 13, x+10, 17); /* Step marker display */
    for(x=0; x<150; x++) {
        ys=15.5-15.0*sin(3.0*x); /* Calculation of sine */
        yc=15.5-15.0*cos(3.0*x); /* Calculation of cosine */
        line(x+10, ys, x+10, ys); /* Display of sine wave */
        line(x+10, yc, x+10, yc); /* Display of cosine wave */
    }
}
```

The following is a sample execution of the above program.

Input	Display
RUN 	

4-5 Simple Martian animation

The program presented here creates a “Martian” figure on the display and then moves it up, down, left and right in response to input from the keyboard. The Martian is drawn using a 5 × 6-dot matrix, the position of which on the display is defined by the location of the upper left dot of the matrix.



The following are the variable functions used in the program.

disp1 (x, y); Displays Martian figure.
 disp2 (x, y); Displays Martian's feet.
 delp (x, y); Erases Martian.

The following shows the complete program.

```
main()
{
    int xx, yy, d, inx, iny;
    clrscr();                               /* Clears display */
    xx=50;yy=8;
    disp2(xx,yy); disp1(xx,yy);           /* Initial Martian display */
    for( ; ; ) {
        inx=iny=0;
        d=getch();                          /* Key code input to d */
        if(d=='4') inx=-5;
        if(d=='6') inx=5;
        if(d=='8') iny=-5;
        if(d=='2') iny=5;
```

```

        delp(xx, yy);                /* Erase Martian */
        xx+=inx; yy+=iny;           /* Next Martian position */
        disp2(xx, yy);              /* Feet display */
        disp1(xx, yy);              /* Martian figure display */
    }
    clrscr();
}

disp1 (x, y)
int x, y;
{
    line(x+2, y, x+2, y+1);    y++;
    line(x+1, y, x+3, y);      y++;
    line(x,y, x+1, y);
    line(x+3, y, x+4, y);      y++;
    line(x,y, x+4, y);
    line(x+1, y, x+1, y+1);
    line(x+3, y, x+3, y+1);
}

disp2 (x, y)
int x, y;
{
    linec(x, y+5, x+6, y+5);
    line(x+1, y+5, x+1, y+5);
    line(x+4, y+5, x+4, y+5);
}

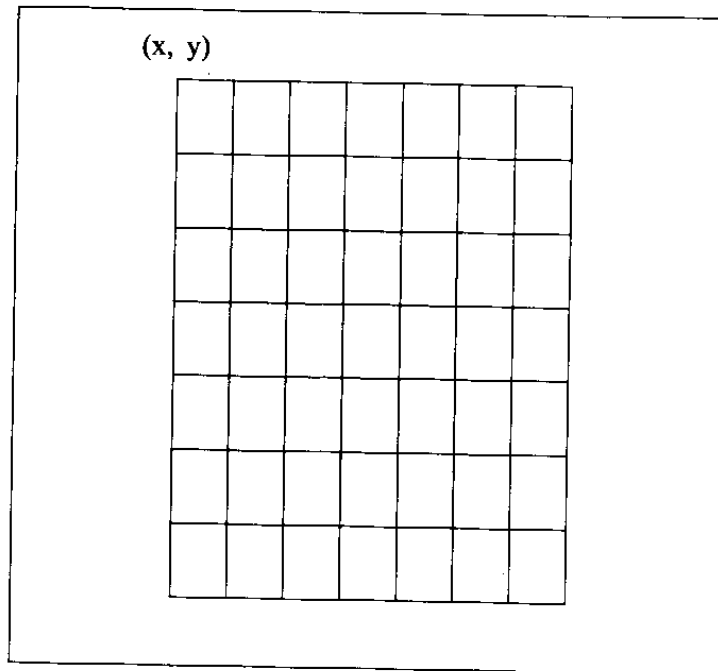
delp(x, y)
int x, y;
{
    int i;
    y--;
    for(i=0; i<7; i++) {
        linec(x-1, y, x+5, y);    y++;
    }
}

```

The Martian appears on the display when you RUN this program. The Martian moves left when you press **[4]**, right when you press **[6]**, up when you press **[8]**, and down when you press **[2]**.

4-6 More Martian animation

This program presents a variation on the Martian theme, with a figure that is a little more refined. With this program, however, the Martian moves left and right in a frame. The following shows the basic Martian figure.



The following shows the variable functions used in the program.

<code>init_disp (x, y);</code>	Displays Martian figure.
<code>disp_leg (x, y);</code>	Displays Martian's feet.
<code>cls_leg (x, y);</code>	Erases Martian's feet.
<code>incx (x, y);</code>	Moves Martian in direction x.

The following shows the complete program.

```
main()
{
    int x,y,i;
    clrscr();
    line(60,1,160,1);           /* Displays frame */
    line(60,30,160,30);
    line(60,1,60,30);
    line(160,1,160,30);
    x=70;y=6;
```

```

    init_disp(x,y);          /* Martian figure display */
    disp_leg(x,y);          /* Feet display */
    for(i=0; i<40; i++, x++) {
        cls_leg(x,y);      /* Erases feet */
        disp_leg(x+1,y);   /* Feet display */
        incx(x,y);         /* Moves Martian+1 */
    }
}

init_disp(x,y)
int x,y;
{
    line(x+3,y,x+3,y); y++;
    line(x+2,y,x+4,y); y++;
    line(x+1,y,x+5,y); y++;
    line(x,y,x+6,y); y++;
    line(x+1,y,x+5,y); y++;
    line(x+3,y,x+2,y);
    line(x+4,y,x+4,y);
}

disp_leg(x,y)
int x,y;
{
    line(x+1,y+6,x+1,y+6);
    line(x+4,y+6,x+4,y+6);
}

cls_leg(x,y)
int x,y;
{
    linec(x,y+6,x+7,y+6);
}

incx(x,y)
int x,y;
{

```



```
line(x+4,y,x+4,y);
linec(x+3,y,x+3,y);
line(x+5,y+1,x+5,y+1);
linec(x+2,y+1,x+2,y+1);
linec(x,y+3,x,y+3);
linec(x+1,y+2,x+1,y+2);
linec(x+1,y+4,x+1,y+4);
line(x+6,y+2,x+6,y+4);
line(x+7,y+3,x+7,y+3);
line(x+5,y+5,x+5,y+5);
linec(x+4,y+5,x+4,y+5);
line(x+3,y+5,x+3,y+5);
linec(x+2,y+5,x+2,y+5);
```

```
}
```

When you RUN the program, the Martian moves left and right within the frame.

4-7 Pseudo-random number generator

This program defines “rand()” function to generate pseudo-random numbers within the range of 0 through 1 to create a bar graph.

At the beginning of this program the three global variables “__seed1”, “__seed2”, “__seed3” are initialized with any value within the range of 1 through 30,000. The following calculation is used each time to generate the random numbers:

```
r = 918999161 * __seed1 + 917846887 * __seed2 + 917362583 * __seed3
__seed1 = 16555425264690 * __seed1 % 27817185604309;
r = 16555425264690 * r % 27817185604309;
return (r/27817185604309 );
```

The cycle of the random numbers generated by the above is approximately 6.95×10^{12} . This method is taken from page 188 of “Applied Statistics”, by B.A. Wichmann and I.D. Hill.

The “main()” program uses the “rand()” function to generate random numbers. It then uses the numbers to create a bar graph, with bars for the rand 0~0.1, 0.1~0.2.... 0.9~1.0. In the program, the “rand()” function is called from “main()”, and the generated values are displayed in a bar graph. 1,000 values are generated by the for loop. To interrupt execution part way, press the **BRK** key.

```
int x[12];
main()
{
    int i, j;
    double rand();

    clrscr();
    for(i=0; i<1000; i++) {
        j=10*rand();
        /* Assigns random values from j=0 through 9 */
        x[j]++; /* Counts number of random values */
        line(x[j], j*3, x[j], j*3+2);
        /* Displays bar graph */
    }
}
```

```
/* Store following in file "rand.c" */
int  _seed1=1000 ,_seed2=12000 ,_seed3=28000 ;
double rand()
{
    double r;
    _seed1=( _seed1%177 ) *171 - ( _seed1 /177 ) *2 ;
    if ( _seed1<0 )_seed1+=30269 ;
    _seed2=( _seed2%176 ) *172 - ( _seed2 /176 ) *35 ;
    if ( _seed2<0 )_seed2+=30307 ;
    _seed3=( _seed3%178 ) *170 - ( _seed3 /178 ) *63 ;
    if ( _seed3<0 )_seed3+=30323 ;
    r=_seed1/30269 .0+_seed2/30307 .0+_seed3/
                                     30323 .0 ;

    while (r>1 .0) r-=1 .0 ;
    return r ;
}
```

The portion of the program following the line: “/* Store following in file “rand.c” */” will be used in the “**Approximation of pi**” program in the following section. Use the following procedure to store this part of the program in a separate file named “rand.c”.

1. Enter the MENU mode.
2. Press the function key under [name] and give a name to the file that contains the pseudo-random number generator. Here, we will use the filename “random.c”.
3. Press the **F10** key and press the function key under [save].
4. The computer will ask you if you wish to save the file “random.c”. Change the name of the file to “rand.c” and press the **EXE** key. This creates a duplicate of the original file under the new name.
5. Return to the MENU mode and enter the editor for the file “rand.c”.
6. Delete all of the lines in the program above the line: “/*Store following in file “rand.c” */”.
7. Exit the file “rand.c” to save it.

4-8 Approximation of pi

This program also uses the “rand()” function to approximate the value of pi.

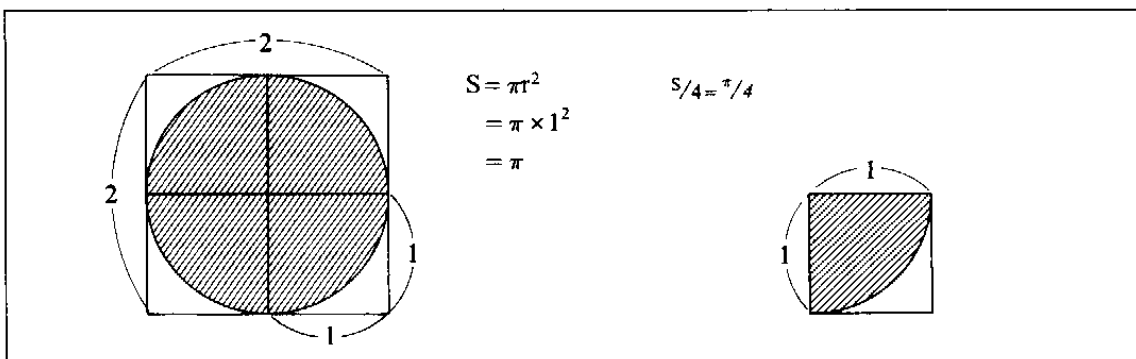
To generate random values, this program uses the section of the program in 4-7 following the remarks /* Store following in file “rand. c ” */. Be sure that you follow the instructions in the previous program to create such a file before attempting to use the program described here. The first line of the program described here is “#include”rand. c””, which tells the program to include the “rand. c” file in its execution. This program also employs “srand(seed)” to initialize the random number stream. This tells the program to change the conditions under which random values are generated each time the program is executed.

Otherwise, the same values would be generated each time the program is executed. Initial values “__seed1”, “__seed2”, and “__seed3” causes random values to be generated in different sequences. The function “srand(seed)” uses seed as the seed for a new sequence of random values. The seed is initialized with any value within the range of 0.0 through 1.0.

The program uses the Monte Carlo Method to approximate the value of pi. This method uses randomly generated values to determine the area of a figure.

A 1×1 square is inlaid with (x, y) points determined by the generated values. If a sufficient number of values are inlaid, it is possible to calculate the approximation of pi. The elliptical equation is $x^2 + y^2 = r^2$, so points within the range of $x^2 + y^2 < 1.0$ fall within the ellipse. These are counted and divided by the total number of points to calculate pi.

For example, if the points in the quarter circle with radius 1 are counted and divided by the total number of random values generated, $1/4$ of the area of the circle can be determined, which is $\pi/4$.



The complete program is listed below. It represents the inlay of random points graphically, and shows the approximation of pi according to the current number of random values generated in the upper left of the display.

When the program is executed, the message “Input seed (0.0 < seed < 1.0)” is displayed to prompt for entry of a value. The starting point of the random number generation is determined by this value. Next a frame and ellipse are displayed, random values are generated, and the approximation of pi is displayed along with points. The for loop in this program is an endless loop. It will continue looping until the batteries go dead, or until you interrupt execution by pressing the **BRK** key.

```

#include "rand.c"
main()
{
    double x,y,count,inc,rand(),seed,srand();
    int xx,yy;
    printf("Input seed(0.0<seed<1.0)");
    scanf("%f",&seed);
    srand(seed);
    clrscr();
    count=inc=0.0;
    line(99,0,130,0); /* Frame display */
    line(99,31,130,31);
    line(99,0,99,31);
    line(130,0,130,31);
    for(x=0.0;x<30.0;x+=0.5){ /* Ellipse display */
        yy=sqrt(900-x*x)+0.5;
        xx=x+100;
        line(xx,yy,xx,yy);
    }
    for( ; ; ){
        x=rand(); y=rand(); /* Assignment of x, y values to x, y */
        xx=100+x*30; yy=1+y*30;
        count++;
        if((x*x+y*y)<1.0)inc++;
        /* Determination of point falls within ellipse */
        gotoxy(0,0); /* Moves cursor to (0,0) */
        printf("%lf\n",inc*4.0/count);
        /* Approximation of pi display */
        line(xx,yy,xx,yy); /* Display of point in frame */
    }
}
double srand(seed); /* 0.0<seed<1.0 */
double seed;
{
    _seed1=seed * 30000.0;
    if(_seed1<30000)_seed1-=30000;
    while(_seed1<0)_seed1+=30000;
    _seed2=rand()*30000;
    if(_seed2==0)_seed2=1;
    _seed3=rand()*30000;
    if(_seed3==0)_seed3=1;
}

```

4-9 Mean and variance

The program presented here reads values from a file and then calculates the mean and variance of the values. The file that the program reads from contains multiple values separated by spaces or carrier returns. The file to contain the data is created using the [data] function, and it is named using the [name] function.

The following are the mean and variance formulas , in which “ x_i ” represents the i th numeric data item, while “ n ” represents the number of data items.

$$\begin{aligned} \text{Mean} &\rightarrow m = (x_1 + x_2 + \dots + x_n) / n \\ \text{Variance} &\rightarrow v = \{ (x_1 - m)^2 + \dots + (x_n - m)^2 \} / n \end{aligned}$$

The following shows the complete program.

```
main()
{
    FILE *infile;
    int i, n;
    char fname[32];
    double sum, mean, v, x;

    printf("Input FILE name ?");
    scanf("%s", fname);          /* Filename input */

    if((infile=fopen(fname, "r"))==NULL)
        exit();                 /* Open file */
    sum=0.0; n=0;
    while(fscanf(infile, "%lf", &x) != EOF)
    {
        sum+=x;
        n++;
    }
    mean=sum/n;                 /* Mean calculation */
    fclose(infile);

    if((infile=fopen(fname, "r"))==NULL)
        exit();
    for(v=0.0, i=0; i<n; i++)
    {
        fscanf(infile, "%lf", &x);
        v+=(x-mean)*(x-mean);
    }
}
```

```

v /= n;                                     /* Variance calculation */
fclose(infile);
printf("Data number=%d\nMean Value=%lf\n
Dif. Value=%lf\n", n, mean, v);
while(getch() != '\n');
}

```

The following shows a sample execution of the program. First comes creation of the data file.

```

Input: [data ]
       12.4 [EXE]
       39.1 [EXE]
       65.8 [EXE]
       88.1 [EXE]
       [MENU]
       [ETC]
       [name ]
       data [EXE]

```

The following shows actual program execution.

Input	Display
RUN [EXE]	Input FILE name ? __
data [EXE]	Data number = 4 Mean Value = 51.350000 Dif. Value = 806.632500 —

4-10 Solution of simultaneous linear equations

The program presented here provides solutions for simultaneous linear equations, using the simplest of methods available, Gauss' Method of Elimination.

This program solves the following nth degree simultaneous linear equations when coefficient $a[i][j]$ is given, for n unknowns $x_0, x_1, x_2, \dots, x_{n-1}$.

$$a[0][0] \cdot x_0 + a[0][1] \cdot x_1 + \dots + a[0][n-1] \cdot x_{n-1} = a[0][n]$$

$$a[1][0] \cdot x_0 + a[1][1] \cdot x_1 + \dots + a[1][n-1] \cdot x_{n-1} = a[1][n]$$

.....

$$a[n-1][0] \cdot x_0 + a[n-1][1] \cdot x_1 + \dots + a[n-1][n-1] \cdot x_{n-1} = a[n-1][n]$$

Execution of the function "gauss(n);" performs the following substitutions for solutions $x_0, x_1, x_2, \dots, x_{n-1}$.

$$a[0][n] \quad \leftarrow \quad x_0$$

$$a[1][n] \quad \leftarrow \quad x_1$$

$$a[2][n] \quad \leftarrow \quad x_2$$

.....

$$a[n-1][n] \quad \leftarrow \quad x_{n-1}$$

When you RUN this program, the computer first asks you for the number of unknowns "n". Then $n \times (n+1)$ coefficients are entered by the operator and the solutions are calculated.

```
#define MAXA 10
double a[MAXA][MAXA];
main()
{
    int n, i, j;

    printf("N=?");
    scanf("%d", &n);          /* Entry of number of unknowns */
    for(i=0; i<n; i++){
        for(j=0; j<n+1; j++){
            printf("a[%d][%d]=?", i, j);
            scanf("%lf", &a[i][j]);
                               /* Entry of number of coefficients */
        }
    }
    gauss(n);                 /* Gauss' elimination */
    for(i=0; i<n; i++)
        printf("%8.2f", a[i][n]);
}
```



```

gauss(n)
int n;
{
    int i, j, k;

    for(k=0; k<n; k++){
        for(j=k+1; j<n+1; j++){
            a[k][j]/=a[k][k];
            for(i=0; i<n; i++){
                if(i!=k)
                    for(j=k+1; j<n+1; j++){
                        a[i][j]-=a[i][k]*a[k][j];
                    }
            }
        }
    }
}

```

The following shows a sample execution of this program.

Input	Display
RUN <input type="button" value="EXE"/>	N = ?_
2 <input type="button" value="EXE"/>	a[0][0] = ?_
1 <input type="button" value="EXE"/>	a[0][1] = ?_
1 <input type="button" value="EXE"/>	a[0][2] = ?_
2 <input type="button" value="EXE"/>	a[1][0] = ?_
1 <input type="button" value="EXE"/>	a[1][1] = ?_
-1 <input type="button" value="EXE"/>	a[1][2] = ?_
1 <input type="button" value="EXE"/>	1.50 0.50
	Ready
	—

Chapter 5

C Interpreter

This chapter contains everything you need to know about the PB-2000C's C interpreter. It contains important information on C in general, as well as operations that are characteristic to the PB-2000C.

5-1 Comments

Comments are non-executable lines of text that can be included anywhere within a program as notes or memos. Comments help to make your programs easier to understand. With C, comments are included between the characters “/*” and “*/” as shown in the following example.

```
/* Program to output "HELLO!" */
main ( )
{
    printf ("HELLO!");
}
```

5-2 Reserved words

Reserved words are words that C sets aside for special purposes. Whenever the computer comes across such words, it performs certain functions, so these words cannot be used for variable names or function names. If you try to use a reserved word for a variable or function name, an error will occur. Beside the reserved words shown here, the *standard function names* listed in Section 5-13 of this manual also cannot be used.

In the following list of reserved words, those marked with an asterisk also cannot be used in the interpreter of the PB-2000C.

auto	default*	float	register	struct	volatile*
break	do	for	return	switch*	while
case*	double	goto	short	typedef*	
char	else	if	signed*	union	
const*	enum*	int	sizeof	unsigned	
continue	extern	long	static	void	

5-3 Data types and lengths

The following table shows the data types and their respective lengths for the PB-2000C interpreter. These are almost identical to the data types used for variables and constants on personal computer compilers.

Type declaration	PB-2000C interpreter
char	8 bits signed
short	16 bits
int	16 bits
long	32 bits
float	32 bits
double	64 bits

5-4 Assigning variable names and function names

All variables must be declared before they can be used within a program. The following rules apply for variables, constants defined using the #define statement, and function names.

- The first character of a variable name must be an alphabetic character (A ~ Z, a ~ z), or an underline.
- Second and subsequent characters in a variable name may be an alphabetic character, underline, or number. The computer differentiates between upper case and lower case letters when matching variable names.
- Reserved words cannot be used as variable names, though a part of a variable name may be a reserved word.
- The length of a variable name is unlimited, but only the first eight characters are used to discriminate one name from another. This means that "abc012345" would be considered identical to "abc012344", since the initial eight characters of either variable name are the same.

Examples of legal variable names

var	XX
My_Name	count
judgement	q1_2_5
V100	int_32

Examples of illegal variable names

123
3A
#count

} First character not a letter or underline.

switch
gets

} Reserved words.

C-1 Hyphen is illegal character.

5-5 Data expressions

Character constants

Characters for the constant type char should be enclosed within single quotation marks as shown below:

'C' '8' '\n'

Just as many larger computers, the PB-2000C represents characters using ASCII codes. The '\n' in above examples is called an *escape sequence*. Escape sequences are used to express characters that cannot be represented using letters or numbers. The following is a table of all ASCII codes available.

Character code table

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0 (ROLL DOWN)	16 SP	32 0	48 @	64 P	80 '	96 p	112 Ç	128 É	144 á	160 ⋮	176 L	192 ⋮	208 α	224 ≡	240
1	1 (ROLL UP)	17 (DEL)	33 !	49 1	65 A	81 Q	97 a	113 q	129 ü	145 æ	161 í	177 ⋮	193 ⋮	209 β	225 ±	241
2	2 (LINE TOP)	18 (INS)	34 "	50 2	66 B	82 R	98 b	114 r	130 é	146 Æ	162 ó	178 ⋮	194 ⋮	210 π	226 Γ	242 ≥
3	3	19	35 #	51 3	67 C	83 S	99 c	115 s	131 â	147 ô	163 ú	179 	195 	211 π	227 ≤	243
4	4	20	36 \$	52 4	68 D	84 T	100 d	116 t	132 ä	148 ö	164 ñ	180 	196 	212 Σ	228 ∫	244
5	5 (LINE DEL)	21	37 %	53 5	69 E	85 U	101 e	117 u	133 à	149 ò	165 Ñ	181 =	197 +	213 F	229 σ	245 J
6	6 (LINE END)	22	38 &	54 6	70 F	86 V	102 f	118 v	134 á	150 û	166 a	182 	198 	214 π	230 μ	246 ÷
7	7 (BEL)	23	39 '	55 7	71 G	87 W	103 g	119 w	135 ç	151 ù	167 ó	183 π	199 	215 	231 τ	247 ≈
8	8 (BS)	24 (LINE C)	40 (56 8	72 H	88 X	104 h	120 x	136 ê	152 ÿ	168 ¿	184 =	200 ⋮	216 ≠	232 Φ	248 □
9	9 (TAB)	25	41)	57 9	73 I	89 Y	105 i	121 y	137 ë	153 Ö	169 ¬	185 =	201 	217 J	233 Θ	249 .
A	10	26	42 *	58 :	74 J	90 Z	106 j	122 z	138 è	154 Û	170 ¬	186 	202 ⋮	218 r	234 Ω	250 -
B	11 (HOME)	27	43 +	59 ;	75 K	91 [107 k	123 {	139 ï	155 ç	171 ½	187 π	203 ⋮	219 ■	235 δ	251 √
C	12 (CLS)	28 (→)	44 ,	60 <	76 L	92 \<	108 l	124 ì	140 î	156 £	172 ¼	188 ⋮	204 	220 ■	236 ∞	252 n
D	13 (CR LF)	29 (←)	45 -	61 =	77 M	93]	109 m	125 }	141 ì	157 ¥	173 	189 ⋮	205 =	221 	237 φ	253 ²
E	14	30 (↑)	46 .	62 >	78 N	94 ^	110 n	126 ~	142 À	158 Pt	174 «	190 =	206 ⋮	222 	238 ε	254 ■
F	15 (↓)	31 /	47 ?	63 O	79 _	95 o	111 DEL	127 À	143 f	159 »	175 ¬	191 ⋮	207 ⋮	223 ■	239 ∩	255 SP

Character codes within C programs are handled as shown below.

Character	A	Z
Code (decimal)	65	90
Code (hexadecimal)	41	5A
C program notation	0x41	0x5A

Even for char type data, internal operations are performed using binary notation. For example, the characters '3' and 'A' would be represented internally as:

```
'3' ..... 00110011
'A' ..... 01000001
```

In addition, the computer uses *control characters*, such as: NL, HT, BS, CR, which are expressed in C as escape sequences.

Code	Name	Meaning
<code>\a</code>	Bell (BEL)	Sound buzzer.
<code>\n</code>	Newline (NL)	Carrier return + linefeed
<code>\t</code>	Horizontal tab (HT)	Horizontal tab
<code>\b</code>	Backspace (BS)	Backspace (one character)
<code>\r</code>	Carrier return (CR)	Carrier return
<code>\f</code>	Form feed	Change page
<code>\\</code>	Character “\”
<code>\'</code>	Character “'”
<code>\"</code>	Character “”
<code>\0</code>	Null	Equivalent to 0.
<code>\nnn</code>	Octal notation	Character code for octal value nnn.

The following shows some example control characters (1-byte constants).

```
'\n'   '\b'   '\\\ '   '\033'
```

Integer constants

Constants of the int type can be broadly classified as decimal, octal, and hexadecimal. The letter ‘L’ or ‘l’ must be affixed for long type integers.

Decimals

Example:	1121	-128	68000	123456789L (for long)
Note:	Leading zeros cannot be used to differentiate between decimal and octal notation.			

Octal

Example:	033	0777	012	01234567L (for long)
Note:	Leading zero required for octal notation.			

Hexadecimal

Example:	0x1B	0xFFFF	0x09	0xffffffffL (for long)
Note:	0x or 0X required for hexadecimal notation.			

The following show the ranges of each type of notation. Negative integers are accomplished using unary minus operators with unsigned constants.

Decimal constants

int	0	~	32767
long	32768	~	2147483647

Octal constants

int	00	~	077777
unsigned int	0100000	~	0177777
long	0200000	~	01777777777

Hexadecimal constants

int	0x0000	~	0x7fff
unsigned int	0x8000	~	0xffff
long	0x10000	~	0x7fffffff

The ranges defined above also apply to entries for the scanf, fscanf, and sscanf functions.

Floating-point constants and double precision floating-point constants

Decimal values can be defined as float type or double type constants using the format shown below. Exponents are indicated by the letter "E" or "e".

Example: 3.1416 -1.4142 1.0e-4 1.23456E5

The following shows the ranges of float and double.

float	0, ±1e-63	~	±9.99999e+63
double	0, ±1e-99	~	±9.999999999e+99

String constants

String constants are contained in double quotation marks. The structure of character strings is basically the same as that for the compiler, with a 0 (null) being affixed at the end. The following shows an example of a character string:

"How do you do?"

5-6 Operators

C employs many of the operators not available with BASIC, FORTRAN and Pascal. Operators are represented by such symbols as "+", "-", "x", and "/", and they are used to alter values assigned to variables. Basically, the PB-2000C interpreter supports the same operators as the standard C compiler.

Precedence

When a single expression contains a number of operators, precedence determines which operator is executed first. Note the following:

$a + b * c$

In this case, the operation $b * c$ is performed first, and then the result of this is added to "a". This means that the precedence of multiplication is higher than that of addition. Note the following example:

$(a + b) * c$

In this case, the $(a + b)$ operation is performed first, because it is enclosed within parentheses.

The following table shows all of the operators used by C and their functions, explained in their order of precedence.

Primary operators	
(~), func (~)	Parenthetical, function argument operations.
x[~], y[~][~]	Specify array elements.
st. memb	Specify structure members.
pst -> memb	Specifies structure members using pointer.
Unary operators	
*px	Specifies contents indicated by pointer.
&x	Address of variable x.
+x, -x	Positive/negative value x.
++x, --x	+1/-1 before using variable x.
x++, x--	+1/-1 after using variable x.
~x	NOT performed on each x bit (inversion).
!x	Logical NOT (if x < > 0, 0 returned; if x=0, 1 returned).
(type) x	Forced conversion/specification of x (cast operator).
sizeof (x)	Variable x byte length value, sizeof(type) illegal.
Binomial operators	
x*y, x/y, x%y	Multiplication, division, modulus (remainder of x divided by y).
x+y, x-y	Addition, subtraction.
x<<y, x>>y	x left bit shift/right bit shift y times.
x<y, x<=y, x>y, x>=y	Relational operators (true=1, false=0).
x==y	Equality (unequal=0, equal=1).
x!=y	Inequality (unequal=1, equal=0).
x&y	Bit AND of x and y.
x^y	Bit XOR of x and y.
x y	Bit OR of x and y.
x&&y	Logical AND of x and y (1 if neither x and y are 0).
x y	Logical OR of x and y (1 if neither x or y are 0).
Conditional operators (trinomial operators)	
x?y : z	y if x is true (other than 0), z if x is false (0).
Assignment operators	
x=y	Assigns y to variable x.
x*=y ↔ x=x*y	Multiplies x by y and assigns result to x.
x/=y ↔ x=x/y	Divides x by y and assigns result to x.
x%=y, x+=y, x-=y, x<<=y, x>>=y, x&=y, x =y, x^=y	

The following table shows the precedence and associativity of the above operators.

Precedence and Associativity of Operators

Precedence	Operators		Associativity
High		(), [], - >, .	→ Left to right
	Unary	!, -, ++, --, ~, (Type), *, &, sizeof	← Right to left
	Multiplication/ division	*, /, %	→ Left to right
	Addition/ subtraction	+, -	→ Left to right
	Shift	<<, >>	→ Left to right
	Relational	>, <, >=, <=	→ Left to right
	Equality	=, !=	→ Left to right
	Bit AND	&	→ Left to right
	Bit XOR	^	→ Left to right
	Bit OR		→ Left to right
	Logical AND	&&	→ Left to right
	Logical OR		→ Left to right
	Conditional	? :	← Right to left
	Assignment	=, +=, -=, *=, /=, other	← Right to left
Low	Order	'	→ Left to right

Important

Data types are ranked as follows, with rank increasing from left to right:

char < int < long < float < double

In expressions with mixed data types, C will first convert all data to the same type as that of the highest ranked data included in the expression, and then perform the calculation. However, the PB-2000C cannot internally convert between unsigned type and long type data. Therefore, to use unsigned and long type data in the same calculation, you should include the following cast operators:

```

unsigned int a;

long b, c;
.....

c=(long) a + b;
.....

```

5-7 Control structures

C program statements that are made up of multiple execution units are executed in a specific sequence. A *control structure* makes it possible to change the sequence of the statement execution, repeat execution of statements, and cause jumps according to specific conditions.

Statements

The smallest statement unit is a simple expression that contains data, plus an operator or function call. Such statements end with a semicolon, and are generally used for assignment of a constant to a variable or to call a function.

```
x=sin(a+b)-c;  
printf("x=%10.4%lf\n", x);  
a=b+(c=getchar( ));
```

Compound statements

When multiple statements are enclosed between braces, the computer treats the group as a single statement. The only difference is that a semicolon is generally not included following the closed brace at the end of the compound statement.

```
{  
    Statement 1  
    Statement 2  
    .....  
    Statement n  
}
```

Control structures for jumps and repeats

The following are the rules that apply to jumps and repeats in C.

Conditional jumps (1)	
if (condition) statement	Statement is executed if condition is met.
if (condition) statement 1	Statement 1 is executed if condition is met.
else statement 2	Statement 2 is executed if condition not met.
Repeat control	
while (condition) statement	Statement is repeatedly executed as long as condition is met.
for (expression 1; expression 2; expression 3) statement	First, expression 1 executed on first pass only (initialization). Then, expression 2 is evaluated, and if true, statement is executed. Next, expression 3 is executed, expression 2 is evaluated, and the process repeats until expression 2 is false.
do statement while (condition);	First, statement is executed, and then condition is evaluated. If true, statement is executed again.
Endless loops	
The three types of endless loops are shown below. Endless loops can be terminated by the break statement, return statement, or exit(). The <code>ctrl</code> key can also be used to manually exit a loop.	
<pre>for (; ;) { ... repeat ... } while (1) { ... repeat ... } do { ... repeat ... } while (1);</pre>	
Conditional jumps (2)	
if (expression 1) statement 1	If expression 1 is true, statement 1 is executed, if expression 2 is true, statement 2 is executed, etc.
else if (expression 2) statement 2	
else if (expression 3) statement 3	
.....	
else if (expression n) statement n	If all expressions are false, statement n + 1 is executed.
else statement n + 1	
This else ~ if statement is functionally equivalent to the standard switch ~ case statement.	

Break statement and continue statement	
<pre>while (1) { if (expression) break; }</pre>	Execution of break statement exits while, for, and do ~ while loops. In example, if expression is true, loop is exited.
<pre>while (expression 1) { if (expression 2) continue; }</pre>	Execution of continue statement returns execution to beginning of while, for, and do ~ while loop. In example, if expression 2 is true, continue statement returns execution to statement 1 for evaluation.
Unconditional jump	
<pre>goto label; label:</pre>	Causes unconditional jump to location of label. Though available in C, unconditional jumps are rarely used.

5-8 Storage classes

Storage classes are used to specify the memory storage area for declared variables, as well as to specify the scope (range that program can read from/write to).

The following table shows the storage classes, as well as the variables that are available for each class.

Storage class	Variables
auto	Used for short-term storage within a program.
static	Reserves area throughout execution of program. Values accessed and acted upon throughout entire program.
register (same as auto)	High frequency access. Variables that are effective for increasing speed of execution by allocating values to registers.
extern	File-external or function-external global variables. With PB-2000C, file-external can be another file also loaded in memory.

Important

- With the PB-2000C's interpreter, register variables and auto variables are identical.
- The auto variables should be declared at the beginning of a function, as in Program A, below. Declarations within the execution of a function, as in Program B and Program C, result in an error.

```
func(a)
double a;
{
    int i, j, x;
    float fx, fy;
    long lx, ly;
    .....
}
```

Program A — OK

```
func(a)
double a;
{
    int i, j, x;
    printf("%0lf", a);
    float fx, fy;
    .....
}
```

Program B — Error

```
func(a)
double a;
{
    int i, j;
    float fx, fy;
    for(i=0; i<10; i++ ) {
        int x;
        .....
    }
}
```

Program C — Error

- File-internal gloval variables cannot be declared as in program A, below. The declaration should be made as shown in B or C. In these two programs, note that the current scopes are different.

```
static int x;
main( )
{
    .....
}
```

A-Error

```
int x;
main( )
{
    .....
}
```

B

or

```
main( )
{
    static int x;
    .....
}
```

C

- Even if you load multiple files that declare the same global variables, an error will not be generated. The variables will be treated as global variables, common to all of the files. The three following programs: "main.c", "sub1.c", and "sub2.c" all produce results of 100 and 200.

```
int x;
main( )
{
    x=100;
    sub1( );
    sub2( );
}
```

"main.c"

```
int x;
sub1( )
{
    printf("In sub1 ");
    printf("x= %d \n", x);
    x + =100;
}
```

"sub1.c"


```
extern int x;
sub2( )
{
    printf("In sub2 ");
    printf("x = %d \n",x);
}
```

“sub2.c”

This means that variable x is applied commonly in all three files. Of course, even if the following change is made in the declaration of “main.c” or “sub1.c”, 100 and 200 will still be output:

```
int x;          →      extern int x;
```

5-9 Arrays and pointers

Arrays

The PB-2000C's interpreter allows use of arrays up to n dimension in size.

```
int    m[2] [12] [31];
```

Initializing arrays

With the PB-2000C, you can initialize simple variables, but you cannot initialize arrays. Use the following assignment procedure in place of array initialization.

```
char *month[12] = {"Jan",
"Feb", ..., "Dec"};          →      char *month[12];
                                month[0] = "Jan", month[1] = "Feb",
                                ....., Month[11] = "Dec";
```

Pointers

The pointer is a 16-bit variable that holds a variable address.

For example:

```
x = *px;
```

Here, the contents at the address pointed to by `px` are assigned to variable `x`.

Note the following:

```
px = &x;
y = *px;
```

Here, the address of `x` is assigned to `px`, and then the contents at the address pointed to by `px` are assigned to variable `y`.

Consequently:

```
y = x;
```

You can also use a pointer within a character string to isolate single characters from the string.

```
main ( )
{
    char *p;
    p = "Casio";
    printf("%c %s\n", *p, p);
}
```

Executing this program produces the following result:

```
C Casio
```

This is because the pointer is pointing at the letter "C".

If you want to display the "s" of "Casio" together with the "C", you would rewrite the original program to:

```
main ( )
{
    char *p;
    p = "Casio";
    printf("%c %c\n", *p, *(p+2));
}
```

You can also use the pointer to show each element in an array.

```
main ( )
{
    int i, a[5], *pa;
    pa=a;
    for(i=0;i<=4;i+ +)
        *pa + + =i;
    for(i=0;i<=4;i+ +)
        printf("%d", a[i]);
    printf("\n");
}
```

Here, each element of array a are expressed as pa. The pointer moves to each successive element as follows:

pa	a[0]
pa + 1	a[1]
pa + 2	a[2]
pa + 3	a[3]
pa + 4	a[4]

Remember that the pointer contains the address of the variable only. This means that certain cases require that you first reserve a storage area.

```
main ( )
{
    char *pc, c[80];
    pc = "abc";
    pc = c;
    strcpy(pc, "abcdefgh");
    printf("%s\n", pc);
}
```

In this example, array c is declared in order to reserve a storage area for pointer pc and 80 characters.

5-10 Functions

In C, a single unit of work performance is called a *function*. C has been designed for the application of functions for simple, yet effective programs, so it is no wonder that the typical C program is made up of a number of component functions. The activity of “writing a C program” is virtually another way of saying “writing a C function”. C also features a number of built-in functions that are often referred to as *standard functions* or *library functions*. For details on the standard functions, see section 5-13, as well as the function reference in Part 2 of this manual.

Here, we will cover the fundamentals of functions, and point out a few important precautions.

main function

The main function is the very first to be executed. With the PB-2000C, the main function never contains an argument. This means that the command input lines cannot be surveyed within a program, as shown below, so the scanf function must be used for input from the keyboard.

```
main(argc, argv)
int  argc;
char *argv[ ];
{
    .....
}
```

Function type declaration

A function type must be declared except for functions that return the int type. The following sample program uses the function dsquare to square double type values from 0 through 10.

```
main()
{
    double d, dsquare();

    for(d=1.0; d<=10.0; d+=1.0)
        printf("(%.1f)^2=%.1f\n", d,
                dsquare(d));
}

double dsquare(x)
double x;
{
    return (x*x);
}
```

As we see above, declarations for the returned values that are not int type must be performed in both the function being called and the function that eventually uses the value. The following example shows the use of the extern declaration.

```
extern double dsquare();
main()
{
    double d;

    for (d=1.0; d<=10.0; d+=1.0)
        printf("(%.1f)^2=%.1f\n", d,
                dsquare(d));
}

double dsquare(x)
double x;
{
    return (x*x);
}
```

Note the other important precautions:

- The PB-2000C also allows recursive recall.
- The void function can also be declared.
- Function pointer declaration as shown below cannot be performed.

```
main()
{
    int pr(), (*pr_p)();
    pr_p = pr;
    pr();
    (*pr_p)();
}
pr()
{ printf("In pr_func\n"); }
```

- Standard functions can be used without type declaration. Standard C compilers require such header files as "stdio.h", "math.h", or "ctype.h" before standard functions can be used, but the PB-2000C does not require such header files.

5-11 Structures and unions

C lets you collect various data items, even those of different types, and group them together under a newly defined data structure.

```
struct identifier {structure declaration}
```

For example:

```
main( )
{
    struct dat {
        int month;
        int day;
    };

    struct dat x;
    struct dat y;
    x.month=12;
    x.day=16;
    y.month=10;
    y.day=21;
    printf("%0dm %0dd\n", x.month, x.day);
    printf("%0dm %0dd\n", y.month, y.day);
}
```

Here, a data structure is declared under the name (commonly called a *tag* in this case) "dat", and then structure dat is assigned to variables x and y. The data assigned to the items (commonly called *members*) inside of the x, y structures are expressed using the following member operators:

```
x.month
x.day
y.month
y.day
```

Another object that can contain data of different types is the union.

```
union identifier {structure declaration}
```

The union lets you store data of different types in the same location.

```
main( )
{
    union dat {
        int i;
        char c;
        double d;
    } x;
    x.i=123;
    printf("%d", x.i);
    x.c='A';
    printf("%c", x.c);
    x.d=123456e-12;
    printf("%e\n", x.d);
}
```

The difference between struct and union is that there is only one x variable in the case of union.

Note the following precautions when using structures and unions with the PB-2000C.

- You cannot initialize structures and unions. Initial values must be assigned.
- Enumeration type and bit fields cannot be used with structures and unions.
- Structures and unions cannot be nested.
- Constant expressions cannot be used after the declaration.

5-12 Preprocessor

C language features a preprocessor to process execution functions before they are compiled. With the PB-2000C interpreter, you can use `#include` to link one file with another, and `#define` to assign a symbolic name to a particular string of characters.

The following is the format for `#include`.

```
#include "filename"
```

The PB-2000C lets you include the filename as "filename", or `<filename>`. Note also that declaration of the following header files, required by standard C compilers, are not necessary with the PB-2000C:

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

The following is the format for `#define`:

```
#define identifier character string constant definition or replacement text
                                     (replacement text cannot be omitted)
```

Important

- The PB-2000C does not allow macro definitions including arguments with `#define` as shown below.

```
#define isupper(x) (((x) >= 'A' && (x) <= 'Z') ? 1 : 0)
```

Use the appropriate functions instead.

```
isupper(x)
unsigned int x;
{
    return(x >= 'A' && x <= 'Z') ? 1 : 0;
}
```

- The PB-2000C stores the following data automatically:

```
#define NULL 0
#define EOF -1
#define FILE char
char *stdin;
char *stdout;
char *stderr;
int errno;
```

However, extern declaration is required to use `stdin`, `stdout`, `stderr` and `errno`.

5-13 Standard functions

This section contains outlines of the most commonly used standard functions. For a complete list, see the function reference in Part 2 of this manual.

File input/output functions

Single-character input/output

<pre>error=putchar(c); error=putc(c, fp); char c; FILE *fp; int error;</pre>	<p>Writes character <i>c</i> to standard output file. Writes character <i>c</i> to file pointed to by <i>fp</i>.</p> <ul style="list-style-type: none"> • When <code>error == EOF</code>, error or EOF.
<pre>c=getchar(); c=getc(fp); char c; FILE *fp;</pre>	<p>Reads single character from standard input file. Reads single character from file pointed to by <i>fp</i>.</p> <ul style="list-style-type: none"> • Function value <i>c</i> is character code for entered character. • When <code>c == EOF</code>, error or EOF.

Character string input/output

<pre>r=puts(s); r=fputs(s, fp); int r; char *s; FILE *fp;</pre>	<p>Writes character string <i>s</i> to standard output file. Writes character string <i>s</i> to file pointed to by <i>fp</i>.</p> <ul style="list-style-type: none"> • When function value <code>r == EOF</code>, error or EOF.
<pre>p=gets(s); p=fgets(s, c, fp); char *s, *p; int c; FILE *fp;</pre>	<p>Reads one-line character string from standard input file. Reads one-line character string from file pointed to by <i>fp</i>. Character string is at location pointed to by pointer in argument. <i>c</i> is number of characters.</p> <ul style="list-style-type: none"> • When <code>p == NULL</code> NULL pointer, error or EOF.

Expression input/output

<pre>printf(s, arg1, arg2, ...); fprintf(fp, s, arg1, arg2, ...); sprintf(ss, s, arg1, arg2, ...); char *s; ← expression FILE *fp; char ss[]; ← character array to be written</pre>	<p>Performs output of expression to standard output. Performs output of expression to file pointed to by fp. Performs output of expression to character array ss.</p>
<pre>scanf(s, arg1, arg2, ...); fscanf(fp, s, arg1, arg2, ...); sscanf(ss, s, arg1, arg2, ...); char *s; ← expression FILE *fp; char *ss; ← character string to be read</pre>	<p>Performs input of expression from standard input. Performs input of expression from file pointed to by fp. Performs input of expression from character string ss.</p> <ul style="list-style-type: none"> • When function value == EOF, error or EOF.

Opening and closing files (S)

<pre>fp = fopen(filename, mode); char *filename, *mode; FILE *fp;</pre>	<p>Opens character string file.</p> <ul style="list-style-type: none"> • fp is file pointer.
<pre>error = fclose(fp); FILE *fp; int error;</pre>	<p>Closes character string file.</p> <ul style="list-style-type: none"> • When error == -1, error.

String functions

<pre>strcat(str1, str2); char *str1, *str2;</pre>	Appends string str2 to end of string str1.
<pre>strcpy(tstr, fstr); char *tstr, *fstr;</pre>	Copies string fstr to string array tstr.
<pre>a = strcmp(str1, str2); char *str1, *str2; int a;</pre>	Compares string str1 with string str2. If identical, returns 0. If different, returns other than 0.
<pre>l = strlen(str); char *str; int l;</pre>	Returns length (number of characters) of string str.

Numeric functions

<code>y = sqrt(x)</code>	Calculates square root of x.
<code>y = sin(x)</code>	Calculates sine of x.
<code>y = cos(x)</code>	Calculates cosine of x.
<code>y = tan(x)</code>	Calculates tangent of x.
<code>y = atan(x)</code>	Calculates arctangent of x.
<code>y = exp(x)</code>	Calculates exponent of x.
<code>y = log(x)</code>	Calculates natural logarithm of x.
<code>z = pow(x, y)</code>	Calculates x to the power of y.
<code>angle(n)</code>	Specifies the unit of angular measurement. n=0 : DEG, 1 : RAD, 2 : GRAD
<code>m = abs(n)</code>	Assigns the real value of integer type value n to integer type value m.

Memory functions

```
char *malloc(memsize);
unsigned memsize;
```

The malloc function makes it possible to reserve array memory area within a program. The number of bytes specified by memsize is reserved. Once memory is reserved, the pointer for the starting address of the memory area is returned. If memory could not be reserved, a null pointer (zero) is returned.

```
char *calloc(count, size);
unsigned count, size;
```

The calloc function is similar to malloc in that it is used within a program to reserve memory area for an array. The memory area reserved is determined by (size x count), where size is the length of each element (in bytes), and count is the number of elements making up the array (or structure). Once memory is reserved, the pointer for the starting address of the memory area is returned. If memory could not be reserved, a null pointer (zero) is returned.

Standard functions for display and keyboard control

beep(n); unsigned n;	Sounds buzzer. 0=low pitch, 1=high pitch.
clrscr();	Clears display and moves cursor to upper left of screen.
gotoxy(x, y); int x, y;	Moves cursor to coordinate (x, y).
line(x1, y1, x2, y2); int x1, y1, x2, y2;	Draws line from graphic coordinates (x1, y1) to (x2, y2).
linec(x1, y1, x2, y2); int x1, y1, x2, y2;	Erases line from graphic coordinates (x1, y1) to (x2, y2).
c=getch(); int c;	Gets single character from keyboard. Entered character shown on display.

Other standard functions

<code>exit(); /* no argument */</code>	Closes file and performs normal termination of execution.
<code>abort();</code>	Display "abort" message and performs abnormal termination of execution.
<code>breakpt();</code>	Terminates execution and enters debug mode.
<code>err = remove(fname);</code> <code>char *fname;</code> <code>int err</code>	Deletes files named fname. Normal if err=0; Error if err = -1.
<code>err = rename(old, new);</code> <code>char old, new;</code> <code>int err;</code>	Changes "old" filename to "new" filename. Normal if err=0; Error if err = -1.

Part 2

Reference

Chapter 6	Command Reference	122
Chapter 7	Standard Function Reference	126
Chapter 8	Error Message Tables	168

Chapter 6 Command Reference

LOAD

Format:

LOAD "filename"

Parameters:

filename specifies the name of the file to be loaded. See Note on filenames on page 125.

Purpose:

This command is used to load a C source program from the file identified by the specified filename. The syntax of the program is analyzed and the program is stored in the p-code area and symbol area of the computer memory. If an error is detected during the syntax analysis, an error message is displayed and execution is terminated.

The LOAD operation is impossible if a file with the same filename as that specified in the LOAD command is already loaded.

The identifier of files being loaded must be "C".

RUN

Format:

RUN [<standard input device name>] [>standard output device name]

Parameters:

standard input device name/standard output device name — standard input device/standard output device specified by redirect file. When omitted, the standard device corresponds to the filename type (see page 125).

Purpose:

This command executes the program stored in the p-code area as a main() function. Standard input devices are the computer, floppy disk, RS-232C, and RAM files. Standard output devices are the computer, printer, floppy disk, RS-232C, and RAM files. However, the identifier of the file must be "S" in the case of floppy disk or RAM file.

Example:

```
RUN >"PRN"  
RUN <"INPUT.DAT">"OUTPUT.DAT"
```

NEW

Format:

NEW

Purpose:

This command deletes the program stored in the p-code area and symbol area.

FLIST

Format:

FLIST

Purpose:

This command sequentially displays the names of the currently loaded files. Each display of a filename is followed by a question mark for which one of the two following commands should be entered:

- C or EXE — display next file
- A — terminate FLIST display

The message “No file” is displayed if FLIST is executed and there are no files currently loaded in memory.

EDIT

Format:

EDIT ["filename"]

Parameters:

filename ... specifies the name of the file to be edited (see page 125).

Purpose:

This command enters the C editor.

When a filename is specified in the EDIT command, that file is opened for editing. If the specified file is a RAM file, it must have the identifier “C”.

When the filename is omitted, the latest loaded RAM file or a RAM file which has generated an error is opened for editing.

An unnamed file is opened for editing in the following cases when the latest loaded file or the file which has generated an error is not a RAM file:

- Immediately after entering the editor by pressing the function key under [c] on the function key menu.
- Immediately following execution of the NEW command.
- Immediately after operation of the NEWALL button.
- Immediately after changing the C area memory status after pressing the function key under [memory] on the function key menu.

When a file is opened for editing because of an error generated following execution of the LOAD or RUN command, the computer automatically displays the beginning of the line in which the error was generated.

TRON

Format:

TRON

Purpose:

This command enters the TRACE mode.

In the TRACE mode, programs are executed line-by-line and the currently executed line is shown on the display.

The trace mode is automatically cancelled when you enter the interpreter.

Display format

[filename (line number)] source line?

filename name of the currently loaded file or the include file name.

line number 1 ~ 65535

source line source program logical line (this is displayed only when the loaded or include file is in computer memory. Not displayed when file is disk file or when file is not in memory.)

One of the following two commands can be entered in response to the “?” prompt:

C or **EXE** — display next line

BRK — enter break mode (see breakpt).

TROFF

Format:

TROFF

Purpose:

This command cancels the trace mode.

Note on filenames

A *filename* is a character string enclosed within double quotation marks. If you wish to include double quotation marks within a filename, the quotation marks that are part of the string must be enclosed within another set of double quotation marks that define the string. When the filename is at the end of a command line, the closing double quotation marks may be omitted.

Filename	Logical Device Name
CON	Console
con	Console
PRN	Printer
prn	Printer
0 : filename	Disk
COM0 : file parameters	RS-232C
com0 :	RS-232C
filename	RAM file

Chapter 7

Standard Function Reference

The *standard functions* are those that are already defined within the C interpreter, so they do not require definition when called.

The following is a list of the types of standard functions built into the PB-2000C's interpreter.

- Input/output functions
- Process functions
- Memory functions
- String functions
- Numeric functions
- Other standard functions

This standard function reference explains each function in terms of format, purpose, returned value and example.

The following is an example of the notation used for a standard function.

Return value type	Function	Argument	
int	getc	(stream)	;
	FILE *stream		;
			/* open stream */
	Type of argument that should be declared		Comment

Return value type

Declares the type of value returned by the function (char, short, int, long, float, double).

Function

Actual function. Parentheses are used to enclose arguments, and semicolons are used between statements.

Argument

Multiple arguments are separated by commas.

Type of argument that should be declared

Declares the type for the arguments to be used by the function. This specification is not necessary when arguments are not used.

fopen

Input/Output function

Format:

```
FILE *fopen(name, access);
char *name;          /* Character string representing filename */
char *access;        /* Character string representing access mode */
```

Purpose:

This function opens the file specified by "name".

Returned value:

If this function is executed normally, the pointer (stream) is returned to the FILE structure. In the case of an error, a null is returned.

Example:

```
/* Output of "file 0. dat" contents */
main()
{
    int    c;
    FILE  *stream;
    if((stream=fopen("file0.dat","r"))==
    NULL) exit();
    while((c=getc(stream))!=EOF)
        putchar(c);
    fclose(stream);          /* File close */
}
```

The file can be a console, printer, FDD, RS-232C or RAM file. Printer files can be opened for output only, and an attempt to open the file for input results in an error. Disk files and RAM files must have an "S" identifier.

Any attempt to open a file that is already opened results in an error, regardless of the access mode. If the access mode is changed, you must close the file and then reopen it.

RS-232C files can be opened for input/output.

Attempts to use other file types result in an error.

name

A character string that specifies the filename, consisting of a device name to which a filename or RS-232C set value is appended.

Device name

Device Name	File Destination
CON	Console
PRN	Printer
0 :	FDD
COM0 :	RS-232C
None	RAM

The device name can be either uppercase or lower-case characters. A device name made up of mixed uppercase/lower-case characters, however, is treated as a RAM file.

Filename

Filenames are used for FDD and RAM specifications. See the PB-2000C Owner's Manual for details on the RS-232C set values.

name examples

1. Console

```
stream=fopen("con","w"); /* Opens screen as stream */
```

2. Printer

```
stream=fopen("PRN","w"); /* Opens printer as stream */
```

3. FDD

```
stream=fopen("0 :XXXXXXXX.XXX","w");
/* XX~XX.XXX=character string */
/* Opens XX~XX.XXX file on FDD as stream */
```

4. RS-232C

```
stream=fopen("COM0 : XX~XX","w");
/* XX~XX=up to 17 characters long */
/* Opens RS-232C as stream */
```

5. RAM

```
stream=fopen("XXXXXXXX.XXX","w");
/* XX~XX.XXX=character string */
/* Opens S-characteristic RAM file as stream */
```

access



Character string that specifies the access mode.

Text mode

"r", "rt"Opens existing file for reading.

"w", "wt"Creates new text file for writing, discarding any previous contents.

"a", "at"Opens file for update, appending new data at the end.

During read operations, carrier return/linefeeds (0x0D, 0x0A) or carrier returns (0x0D) are converted to linefeed (0x0A), while   (0x1A) is converted to EOF.

During write operations, linefeeds (0x0A) are converted to carrier return/linefeed (0x0D, 0x0A).

Update mode

The following specifications can be made when the file destination is RS-232C.

"r+", "rt+", "r+t" "w+", "wt+", "w+t" — open for both reading and writing.

Binary mode

"rb" "r" binary

"wb" "w" binary

"ab" "a" binary

"rb+", "r+b" "r+" binary

"wb+", "w+b" "w+" binary

In order to save memory, file pointer management is fixed. In the following example program, the out1 and out2 values are identical.

```
main()
{
    FILE *out1;
    FILE *out2;
    out1=fopen("test1","w");
    fclose(out1);
    out2=fopen("test2","w");
    if(out1==out2) printf("SAME\n");
    else          printf("DIFFERENT\n");
    fclose(out2);
}
```

fclose

Input/Output function

Format:

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

Purpose:

This function closes an open stream.

Returned value:

If this function is executed normally, 0 is returned to the FILE structure. In the case of an error, EOF is returned.

Example:

```
main()
{
    FILE *stream;
    int status;
    stream=fopen("file0","r");
    :
    status=fclose(stream);
    if(status==EOF)printf("\nError!\n");
}
```

getchar

Input/Output function

Format:

```
int getchar();
```

Purpose:

This function reads one character from standard input stdin, and is identical to `getc(stdin)`. The character input is not returned until the `EXE` key is pressed. Entry of code 0x1A (`SHIFT` + `TAB`) is regarded as EOF.

Returned value:

This function returns the character which is read. EOF indicates an error or file end, determined using `feof` or `ferror`.

Example:

```
int c;
:
c = getchar();
```

getc

Input/Output function

Format:

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

Purpose:

This function reads one character from the current position of the input stream. The stream file pointer then advances to indicate the next sequential character.

For console input, the input character is not returned until the **EXE** key is pressed. For console or RS-232C input, entry of code 0x1A is regarded as EOF.

Returned value:

This function returns the character which is read. EOF indicates an error or file end, determined using feof or ferror.

Example:

The following program converts the contents of the file test.dat from uppercase to lowercase characters.

```
main()
{
    FILE *stream;
    int c;
    stream=fopen("test.dat","r");
    while((c=getc(stream))!=EOF)
        putchar(('A'<=c && c<='Z')?c+0x20:c);
    fclose(stream);
}
```

fgetc

Input/Output function

Format:

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

Purpose:

This function reads one character from the current position of the input stream. The stream file pointer then advances to indicate the next sequential character.

For console input, the input character is not returned until the **EXE** key is pressed. For console or RS-232C input, entry of code 0x1A is regarded as EOF.

Returned value:

This function returns the character which is read. EOF indicates an error or file end, determined using feof or ferror.

Example:

```
FILE *stream;
int c;
:
c = fgetc(stream);
```

putchar

Input/Output function

Format:

```
int putchar(c);
int c; /* Write string */
```

Purpose:

This function writes one character to standard output stdout, and is identical to putc(c, stdout).

Returned value:

This function returns the value which is written. EOF indicates an error, but EOF is also a proper integer, so ferror is used to detect error conditions.

Example:

The following program outputs a 1-line entry.

```
main()
{
    char buffer[64];
    int i,c;
    gets(buffer);
    for (i=0; buffer[i]!='\0'; i++) {
        c = putchar(buffer[i]);
        if (c==EOF)break;
    }
}
```

putc

Input/Output function

Format:

```
int  putc(c, stream);
int  c;                               /* Write string */
FILE *stream;                          /* Open stream */
```

Purpose:

This function writes one character to the current position of the output stream. The stream file pointer then advances to indicate the next sequential character.

Returned value:

This function returns the value which is written. EOF indicates an error, but EOF is also a proper integer, so ferror is used to detect error conditions.

Example:

The following program outputs a 1-line entry to "file 0".

```
main()
{
    FILE *stream;
    char  buffer[64];
    int   i,c;
    stream=fopen("file0","a");
    gets(buffer);
    for(i=0; buffer[i]!='\0';i++) {
        c = putc(buffer[i], stream);
        if(c==EOF)break;
    }
    fclose(stream);
}
```

fputc

Input/Output function

Format:

```
int fputc(c, stream);
int c; /* Write string */
FILE *stream; /* Open stream */
```

Purpose:

This function writes one character to the current position of the output stream. The stream file pointer then advances to indicate the next sequential character.

Returned value:

This function returns the value which is written. EOF indicates an error, but EOF is also a proper integer, so `ferror` is used to detect error conditions.

Example:

```
FILE *stream;
char buffer[32];
int c;
:
c = fputc(buffer[0], stream);
```

gets

Input/Output function

Format:

```
char *gets(string);
char *string; /* Data storage pointer */
```

Purpose:

This function reads one line from standard input `stdin` and stores it in "string".

Characters are read up to carrier return/linefeed encountered.

The carrier return/linefeed character is replaced with a null before being stored in the string.

Returned value:

This function changes the data storage pointer. Return of a null indicates an error or EOF, determined using `feof` or `ferror`.

Example:

```
char string[30], *result;
:
result = gets(string);
```

fgets

Input/Output function

Format:

```
char *fgets(string, count, stream);
char *string;           /* Data storage pointer */
int count;              /* Number of characters read */
FILE *stream;          /* Open stream */
```

Purpose:

This function reads a character string from the input stream and stores it in "string". Characters are read from the current location of the stream up to the first carrier return/linefeed character encountered, the end of the file, or until the number of characters read equals "count" — 1. A null character is appended to the end of the character string that is returned. Carrier return/linefeed characters are also read into the returned string.

Returned value:

This function returns the data storage pointer. Return of a null indicates an error or EOF, determined using feof or ferror.

Example:

```
FILE *stream;
char string[30], *result;
:
result = fgets (string, 30, stream);
```

puts

Input/Output function

Format:

```
int puts(string);
char *string;           /* Write data pointer */
```

Purpose:

This function writes a string to standard output stdout. The null character indicating the end of the character string is written as a carrier return/linefeed character.

Returned value:

If this function is executed normally, a linefeed character is returned. In the case of an error, EOF is returned.

Example:

```
int result;
:
result = puts ("string");
```

fputs

Input/Output function

Format:

```
int fputs(string, stream);
char *string;           /* Write data pointer */
FILE *stream;          /* Open stream */
```

Purpose:

This function writes a string starting from the current position of the output stream. The null character indicating the end of the string is not written.

Returned value:

This function returns the last written character. If the string is blank, 0 is returned, and EOF is returned when an error occurs.

Example:

```
FILE *stream;
int result;
:
result = fputs("string", stream);
```

fread

Input/Output function

Format:

```
int fread(buffer, size, number, stream);
char *buffer;           /* Data storage pointer */
int size;              /* Number of data bytes */
int number;            /* Number of data items */
FILE *stream;          /* Open stream */
```

Purpose:

This function reads from the data stream the number data items specified by "number", each data item being of the length specified by "size", and stores the items in the buffer. The stream file pointer advances the actual number of bytes read.

Returned value:

This function returns the number of data items read. A number is less than the value specified by "number" indicates an error or that EOF was reached.

Example:

```
FILE *stream;
long  buffer[30];
int   count;
    ⋮
stream=fopen("file0","r");
count=fread((char *)buffer,sizeof(long),
           30,stream);
```

fwrite

Input/Output function

Format:

```
int  fwrite(buffer, size, number, stream);
char *buffer;           /* Write data pointer */
int  size;              /* Number of data bytes */
int  number;           /* Number of data items */
FILE *stream;          /* Open stream */
```

Purpose:

This function writes from the buffer to the output stream, the number data items specified by "number", each data item being of the length specified by "size". The stream file pointer advances the actual number of bytes written.

Returned value:

This function returns the number of data items written. A number is less than the value specified by "number" indicates an error.

Example:

```
FILE *stream;
long  buffer[30];
int   count ;
    ⋮
stream=fopen("file0","w");
count=fwrite((char *)buffer,4,30,stream);
```

printf, fprintf, sprintf

Input/Output function

Format:

```
int printf(format [,argument...]);
int fprintf(stream, format [,argument...]);
int sprintf(buffer, format [,argument...]);
char *format;           /* Output conversion specification */
FILE *stream;          /* Open stream */
char *buffer;          /* Output buffer */
argument               /* Argument */
```

Purpose:

This function converts the “argument” in accordance with the “format” specification, and outputs it to stdout in the case of printf, to the file specified by the stream in the case of fprintf, or to the buffer in the case of sprintf. In the case of sprintf, the null character at the end of the character string is also output.

“format” is a character string greater than 0, and can consist of normal characters, escape sequences, and conversion specifications. Normal characters and escape sequences are output in the sequence that they appear, while conversion specifications are executed by sequentially extracting the arguments, performing the conversion and then outputting the data. When there are more arguments than conversion specifications, the extra arguments are simply disregarded. In the case that there are fewer arguments, results are uncertain.

Returned value:

This function returns the number of output characters. Note that the null at the end of the character string is not included in the count in the case of sprintf. EOF is returned when an error occurs.

Example 1 — printf output

```
int count = 234;
printf( "%d %06d %X %x %o\n", count, count, count,
        count, count );
```

Output result

```
234 000234 EA ea 352
```

Example 2 — printf output

```
int count = 234;
printf( "| %d | %6d | %-6d | \n", count, count, count );
```

Output result

```
| 234 |      234 | 234 |
```

The conversion specification is a character string that begins with the character “%”. However, when the character string following “%” does not have the meaning of a conversion specification, that character string (up to the next “%”) is output as it is. For example, “%” is output in the case of “%%”.

% [flag] [field width] [. precision] [l] conversion characters

flag

Include the flag to cause the conversion result to be entered into the field flush left. The default option is flush right.

field width

The minimum field width is specified by an unsigned decimal integer. When the conversion result is smaller than the specified field width, the value is displayed flush left in the field. If “-” is specified for the flag, the value is flush right.

Leading or following columns in the field are filled with spaces, unless the first digit of the field width specification value is 0. In that case, leading and following columns are filled with zeros.

When the conversion result is longer the field width, or when a field width is not specified, the conversion result is output as it is.

Precision

Precision is specified by an unsigned decimal integer following the field width. The field width and precision specifications are separated by a period. If only a period is included without a decimal value, a precision of 0 is assumed.

The following shows the meaning for each precision specification, in accordance with the conversion character.

- d, o, u, x, X.....Specifies the least number of digits output. If the number of digits in the argument are less than the precision, leading 0’s are added to fill empty digits to the left of the output value.
Default: 1
- e, E, f.....Specifies the number of digits output following the decimal points. No decimal places are output for precision 0.
Default: 6
- g, G.....Specifies the maximum number of significant digits output.
Default: all significant digits
- s.....Specifies the maximum number of characters output.
Default: up to null character output
- Other Precision disregarded

l

This value has one of the following meanings in accordance with the conversion characters.

- d, o, x, X.....Specifies long argument.
- Other Disregarded.

Conversion Characters

- d.....int; signed decimal notation.
- o.....int; unsigned octal notation.
- u.....int; unsigned decimal notation.
- x, X.....int; unsigned hexadecimal notation.
Variable for "x" is abcdef, and variable for "X" is ABCDEF
- f.....double; decimal notation in the form [-]ddd.ddd, in which ddd is a 1-digit or longer decimal value. The value is rounded off to the number of decimal places specified by the precision.
- e, E.....double; decimal notation in the form [-]d.ddde±dd for e, and [-]d.dddE±dd for E, in which d is a 1-digit decimal value, ddd is one or more digits, and dd is two digits. The value is rounded off to the number of decimal places specified by the precision.
- g, G.....double; converts f or e (E in the case of G), and the number of significant digits is determined by the precision specification. e (or E) is used if the exponent of the value is less than -4, or greater than or equal to the precision. Trailing zeros are cut off the result, and a decimal is included only when a fractional part is present.
- c.....int; converted to unsigned char.
- s.....char *; characters from the string are output until the null at the end of the character is reached (null not included in output) or until the number of characters specified by the precision have been output.

scanf, fscanf, sscanf

Input/Output function

Format:

```
int scanf(format [,argument...]);
int fscanf(stream, format [,argument...]);
int sscanf(buffer, format [,argument...]);
char *format; /* Input conversion specification */
FILE *stream; /* Open stream */
char *buffer; /* Input buffer */
argument /* Argument */
```

Purpose:

The function converts input data in accordance with the “format” specification, and assigns it to “argument”. Input is performed from stdin in the case of scanf, from the file specified by the stream in the case of fscanf, or from the buffer in the case of sscanf.

“argument” is a pointer that points to the variable type that corresponds to the “format” specification. In the case of sscanf, a null character is used as the end of the buffer, which is equivalent to the end-of-file of scanf and fscanf.

“format” is a character string greater than “0”, made up of spaces, normal characters or conversion specifications. When conversion specifications are present, converted values are sequentially assigned to the following arguments. When there are more values than conversion specifications, the excess arguments are disregarded. In the case that there are fewer arguments, results are uncertain.

Returned value:

This function returns the number of arguments that have been assigned values. EOF is returned when the end-of-file is reached, regardless of the conversions performed up to that point.

Example of input using scanf

```
int i;
float f;
double d;
scanf("%d%f%lf", &i, &f, &d);
```

If “123-1.23e10 203” is entered, i=123, f= -1.23e10, and d=203.0.

format

1. Space characters (spaces/carrier returns)

Input is read until a character that is not a space (this character is not read) is encountered or until there are no more characters. Execution of the function is terminated if a character that is not a space is encountered.

2. Normal characters (other than spaces and %)

The next character is read, and execution of the function is terminated if it is not a normal character, and the input character is not read.

3. Conversion specifications

Conversion specifications are executed according to the following sequence.

- Any spaces entered are skipped, except in the case of conversion character c.
- The input field is read. The input field is defined as being up to the first space, the first characters which cannot be converted by the conversion specification, or by the specified field width. The characters following the input field are considered to be yet unread. Execution of the function is terminated if the length of the input field is 0.
- The input field is converted to the type determined by the conversion character.
- Unless assignment is disabled by an asterisk, conversion results are assigned to arguments following "format" not yet assigned conversion results.

The conversion specification is a character string beginning with "%", in the format shown below. However, when the character string following "%" does not have the meaning of a conversion specification, that character string (up to the next "%") is input as it is. For example, "%%" is input in the case of "%%".

% [*] [field width] [l] conversion characters

*** (assignment disabled)**

The input field can be read to, but assignment of the conversion result to an argument cannot be performed.

field width

The maximum field width is specified by an unsigned decimal integer.

l

This value has one of the following meanings in accordance with the conversion characters.

- d, o, x.....Specifies a long argument.
- e, f, g.....Specifies a double argument.
- Other Disregarded.

Conversion characters

- d.....Decimal integer (can also be unsigned). The corresponding argument points to integer.
- o.....Octal integer (can also be unsigned). The corresponding argument points to integer.
- u.....Unsigned decimal integer. The corresponding argument points to integer.
- x.....Hexadecimal integer (can also be unsigned). The corresponding argument points to integer.
- e, f, g.....Floating-point number (sign is optional). The corresponding argument points to float. Results are uncertain when the input for the character string making up the exponent is greater than two digits.
- s.....String of non-white characters. The corresponding argument points to a character array capable large enough to hold the sequence, including the null character at the end. A null is automatically added to the end of the string.
- c.....String of characters made up of the number of characters specified by the field width. A size of 1 is assumed when there is no field width specification. The corresponding argument points to a character array capable large enough to hold the sequence. Spaces are not skipped, they are read into the string.

ungetc

Input/Output function

Format:

```
int ungetc(c, stream);
int c;                /* Return character */
FILE *stream;        /* Open stream */
```

Purpose:

This function returns one character to the open stream. Before the character is returned, at least one character must be read from the stream. An EOF as the character is disregarded. The number of characters that can be returned is equal to the number of characters read from the string. However, updating of the buffer makes the number of characters read 0.

Returned value:

This function returns character c. A return value of EOF generates an error.

Example:

```
FILE *stream;
int c;
:
c = getc(stream);
:
c = ungetc(c, stream);
```

fflush

Input/Output function

Format:

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

Purpose:

This function writes the buffer contents to the file when an output stream is specified. When an input stream is specified, the buffer contents are cleared. This function does not close the stream.

The buffer is automatically flushed when the buffer becomes full, when the stream is closed, or after normal program execution is attained without closing the stream.

Returned value:

If this function is executed normally, 0 is returned. In the case of an error, EOF is returned.

Example:

```
FILE *stream;
int c, status;
:
c = putc(c, stream);
status = fflush(stream);
```

feof

Input/Output function

Format:

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

Purpose:

This function checks whether or not the stream has reached EOF.

Returned value:

This function returns a value other than 0 when EOF has been reached. When EOF has not yet been reached, a 0 is returned.

Example:

```
FILE *stream;
int status;

status = feof(stream);
```

ferror

Input/Output function

Format:

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

Purpose:

This function checks for a stream error condition.

Returned value:

This function returns a value other than 0 when a stream error condition is detected. When an error condition is not detected, a 0 is returned.

Example:

```
FILE *stream;
int err;
:
err = ferror(stream);
```

clearerr

Input/Output function

Format:

```
void clearerr(stream);
FILE *stream;          /* Open stream */
```

Purpose:

This function clears a stream EOF and an error condition.

Example:

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

remove

Input/Output function

Format:

```
int remove(name);
char *name;          /* Character string representing filename */
```

Purpose:

This function deletes the file specified by the filename, and can be used for FDD and RAM files. An error occurs if an attempt is made to delete an FDD file while any other FDD file is open.

Returned value:

If this function is executed normally, 0 is returned. In the case of an error, -1 is returned. The following shows the errno values and their meanings.

- 2Specified file cannot be found.
- 13.....Specified file is open. File specified is other than FED file or RAM file. Attempt made to delete FDD file while an FDD file is open.

Example:

```
int status;
status = remove("file0");
```

rename

Input/Output function

Format:

```
int rename(oldname, newname);
char *oldname;          /* Character string representing old filename */
char *newname;         /* Character string representing new filename */
```

Purpose:

This function changes the oldname filename with the newname, and can be used with FDD and RAM files. Note that the oldname and newname must be for the same device. An error occurs if an attempt is made to change the name of an FDD file while any other FDD file is open.

Returned value:

If this function is executed normally, 0 is returned. In the case of an error, a value other than 0 is returned. The following shows the errno values and their meanings.

- 2.....file specified by oldname cannot be found.
- 13.....file specified by oldname is open. File specified is other than FDD file or RAM file. Attempt made to change the name of FDD file while an FDD file is open.
- 17.....filename specified by newname already exists.
- 18.....oldname device different from newname device.

Example:

```
int status;
status = rename("file0", "file1");
```

Input/Output Function Errors

When an error is generated by an input/output function call, variable errno is assigned a value which identifies the error.

The following table shows the meaning of each error. It does not include errors that can be generated for specific functions.

See the information on each function for information on function-specific errors. Note that the error value assigned to errno changes when the next function is called, so you should check the value of errno immediately after generation of an error to find out the specific reason.

Value	Meaning
2	<ol style="list-style-type: none">1. Improper filename specification.2. Specified file cannot be found.
5	<ol style="list-style-type: none">1. Error in I/O device operation.2. Problem with drive device.
9	<ol style="list-style-type: none">1. Incorrect stream pointer.2. Open file not referenced.3. Attempt to write to file or device opened for reading.4. Attempt to read from file or device opened for writing.
12	Overflow in work area (I/O buffer) or file area (RAM file).
13	<ol style="list-style-type: none">1. Attempt to open file that is already opened.2. Attempt to open file with other than S identifier.3. Attempt made to open other than RS-232C file for both read and write.4. Attempt to open printer for reading.
19	I/O impossible.
24	Too many open files.
28	No more area on disk available for writing.
35	Attempt made to write to disk file which is write protected.
36	Disk needs formatting.
37	<ol style="list-style-type: none">1. Overflow of RS-232C input buffer.2. Framing error detected at RS-232C port.3. Parity error or overflow error detected at RS-232C port.

exit

Process function

Format:

```
void exit( );
```

Purpose:

This function causes a normal termination of a program by flushing all stream buffers, and closing all open files before termination.

Example:

```
main( )
{
    FILE  *stream;
    :
    if( (stream=fopen("data", "r"))==NULL) {
        printf("couldn't open data file\n");
        exit( );
    }
    :
}
```

abort

Process function

Format:

```
void abort( );
```

Purpose:

This function displays the message "abort" for the standard error output stderr, and causes abnormal program termination. The abort function does not flush the stream buffer.

Example:

```
main( )
{
    FILE  *stream;
    :
    if( (stream=fopen("data", "r"))==NULL) {
        printf("couldn't open data file\n");
        abort( );
    }
    :
}
```

breakpt

Process function

Format:

```
void breakpt( );
```

Purpose:

This function terminates program execution and enters the BREAK mode.

Example:

```

:
:
breakpt( );
:
:

```

About the BREAK mode

The computer enters the BREAK mode when the **BRK** key is pressed in the TRACE mode, or when the breakpt function is executed. When the computer enters the BREAK mode, the prompt "Break ?" appears on the display.

At this time, one of the following commands can be entered in reply.

Command	Meaning
A or BRK	Interrupt execution.
C or EXE	Resume execution.
T	Enter TRACE mode and resume execution.
N	Cancel TRACE mode and resume execution.
D	Display values assigned to variables (if BREAK mode was entered following RUN execution).

Operation	Display
	Break ? _
d	> _
a EXE	> a int 12
	> _
BRK	Break ? _

malloc

Memory function

Format:

```
char *malloc(size);
      unsigned size;          /* Number of bytes reserved */
```

Purpose:

This function reserves a memory area of size "size".
The reserved memory area is cancelled when execution of the program is terminated.

Returned value:

This function returns the pointer of the reserved memory area. A null is returned if reservation of memory is unsuccessful.

Example:

```
main()
{
    double *array;
    :
    if((array=(double*)malloc(256*sizeof
(double))!=NULL){
        printf("Could not reserve\n");
        exit();
    }
    :
}
```

Important

malloc and calloc store data in the reserved memory area into the symbol area. Therefore the symbol area, free area must be at least 8 bytes. If the symbol area is smaller, execute the following allocchk command before malloc and calloc.

```
allocchk()          /* Symbol area check */
{
    int *bgn_p, *end_p;
    bgn_p=0x167E; end_p=0x0A8C;
    if(*end_p- *bgn_p<8) abort();
}
```

Example:

```
:
allocchk();
mem=malloc(size);
:
```

calloc

Memory function

Format:

```
char *calloc(n,size);
      unsigned n;           /* Number of elements */
      unsigned size;       /* Number of bytes per element */
```

Purpose:

This function reserves an array of “n” number of elements, each element being of “size” size. Specifying 0 in this function results in initialization. The reserved memory area is cancelled when execution of the program is terminated.

See malloc.

Returned value:

This function returns the pointer of the reserved memory area. A null is returned if reservation of memory is unsuccessful.

Example:

```
main()
{
    int *iarry;
    :
    if((iarry=(int*)calloc(1000,2))==NULL){
        printf("couldn't use IARRY\n");
        exit();
    }
    for(i=0; i<1000; i++) iarry[i]=0;
    :
}
```

free

Memory function

Format:

```
int free(ptr);
char *ptr;           /* Pointer of freed memory area */
```

Purpose:

This function frees memory area. Argument ptr must be the pointer of the memory area reserved by calloc or malloc.

Memory area which is first reserved by calloc or malloc cannot be freed.

Returned value:

This function returns 0 when the memory is freed. A -1 is returned if argument ptr is not the pointer of the memory area reserved by calloc or malloc.

Example:

```
char *array;
:
array = malloc(256);
:
free(array);
```

strlen

String function

Format:

```
int strlen(string);
char *string;           /* Character string */
```

Purpose:

This function returns the number of bytes in "string", up to the null at the end.

Returned value:

This function returns the length of "string", not including the null at the end. No value is returned when an error occurs.

Example:

```
int length;
:
length = strlen("abc");           /* length=3 */
```

strcpy

String function

Format:

```
char *strcpy(dest, source);  
char *dest, *source;    /* Character string */
```

Purpose:

This function copies from the beginning of “source” up to the null at the end (including the null) to the point starting from “dest”. Overflow check is not performed for the copy operation.

Returned value:

This function returns the “dest” pointer.

Example:

```
char *result, string[64];  
:  
:  
result = strcpy(string, "abc");  
/* string="abc" */
```

strcat

String function

Format:

```
char *strcat(dest,source);  
char *dest,*source;    /* Character string */
```

Purpose:

This function appends from the beginning of “source” up to the null at the end (excluding the null) to the point starting from the first null in “dest”, and inserts a null at the newly created string. Overflow check is not performed for the copy operation.

Returned value:

This function returns the “dest” pointer.

Example:

```

/* Copies file XXXX.XXX to XXXX.bak */
main()
{
    char filename[32], backup[32], *s, c;
    FILE *fp1, *fp2;
    scanf("%s", filename);
    strcpy(backup, filename);
    s = backup;
    while(*s != '\0' && *s != '.') s++;
    *s = '\0';
    strcat(backup, ".bak");
    /* Append character strings */
    if((fp1=fopen(filename, "r"))==NULL) exit();
    fp2=fopen(backup, "w");
    while((c=getc(fp1))!=EOF) putc(c, fp2);
    fclose(fp2); fclose(fp1);
}

```

strcmp

String function

Format:

```

int strcmp(string1, string2);
char *string1, *string2;    /* Character strings to be compared */

```

Purpose:

This function performs a character-by-character comparison (of ASCII codes) of "string1" and "string2" from the beginning of the strings up to the null. The null is also used in the comparison.

Returned value:

This function returns one of the following integer values.

string1 < string2 → value less than 0

string1 = string2 → 0

string1 > string2 → greater than 0

Example:

```

int result;
char string 1[6]="abcde";
char string 2[6]="abcba";
main()
{
    :
    result=strcmp(string1, string2);
    /* result=1 */
}

```


strchr

String function

Format:

```
char *strchr(string, chr);
char *string;           /* Character string */
int chr;                /* Characters to be searched for */
```

Purpose:

This function searches the characters of "string" to see if it contains an occurrences of "chr". This function can also be used to search for nulls.

Returned value:

This function returns the pointer of "chr". A null is returned if no occurrences of "chr" are found in "string".

Example:

The following program searches an input character string for any empty spaces and changes them to "_".

```
main()
{
    char instr[64], *ss;

    printf("Input string ? ");
    gets(instr);
    ss = instr;
    while((ss=strchr(ss, ' '))!=NULL)
        *ss = '_';
    puts(instr);
}
```

abs

Numeric function

Format:

```
int abs(n);
int n;                /* Integer */
```

Purpose:

This function returns the absolute value of an integer.

Returned value:

This function returns the absolute value of integer "n".

Example:

```
int x=-4,y;
y = abs(x);          /* Y=4 */
```

The following declaration would be used to return the absolute value of a floating point value.

```
extern double fabs();
:
double fabs(x)
double x;
{ return(x>0.0)?x:-x; }
```

The C language of the PB-2000C cannot use macros with arguments, so the following #define statement cannot be used, though it may be available on other computers.

```
#define fabs(x) ((x)>0) ? (x) : -(x)
```

sin, cos, tan

Numeric function

Format:

```
double sin(x);
double cos(x);
double tan(x);
double x;          /* Unit of angular measurement (DEG, RAD, GRAD) */
```

Purpose:

Each of these functions returns a value equivalent to its corresponding trigonometric function. The value 33 is assigned to `errno` when the calculation range of angle unit `x` is exceeded. The following shows the allowable calculation range for angle unit `x`.

```
|x| < 1440 (DEG)
|x| < 8π (RAD)
|x| < 1600 (GRAD)
```

Returned value:

Each of these functions returns a value equivalent to its corresponding trigonometric function. When an error occurs, 0 is returned.

Example:

The following program is used to input an angle and then produces its corresponding sin, cosine and tangent values.

```
main()
{
    double y;
    angle(0);
    for(;;){
        printf("Angle ?");
        scanf("%lf", &y);
        printf("%11.10g %11.10g %11.10g \n",
            sin(y), cos(y), tan(y));
    }
}
```

Execution example

Input	Display
RUN <code>EXE</code>	Angle ?_
30 <code>EXE</code>	0.5 0.8660254038 0.57735 02692 Angle ?_

asin, acos, atan

Numeric function

Format:

```
double asin(x);
double acos(x);
double atan(x);
      double x;
```

Purpose:

Each of these functions returns a value equivalent to its corresponding inverse trigonometric function. The value 33 is assigned to errno when the calculation range of x is exceeded. The following shows the allowable calculation range for x.

<Calculation range of x>
 $-1 \leq x \leq 1$ (for asin, acos)
 $|x| < 100^{100}$ (for atan)

<Range of returned value> (for RAD)
 $[-\pi/2, \pi/2]$ (for asin)
 $[0, \pi]$ (for acos)
 $[-\pi/2, \pi/2]$ (for atan)

Returned value:

Each of these functions returns a value equivalent to its corresponding inverse trigonometric function. When an error occurs, 0 is returned.

Example:

```
double y;
  :
angle(1);          /* RAD specification */
y = asin(1.0);    /* y=1.5707963268 */
y = acos(1.0);    /* y=0 */
y = atan(1.0);    /* y=0.7853981634 */
```

sinh, cosh, tanh

Numeric function

Format:

```
double sinh(x);
double cosh(x);
double tanh(x);
double x;
```

Purpose:

Each of these functions returns a value equivalent to its corresponding hyperbolic function.

$$\sinh x = (e^x - e^{-x})/2$$

$$\cosh x = (e^x + e^{-x})/2$$

$$\tanh x = (e^x - e^{-x})/(e^x + e^{-x})$$

The value 33 is assigned to `errno` when the calculation range of `x` is exceeded. The following shows the allowable calculation range for `x`.

$$|x| \leq 230.2585092 \text{ (for sinh, cosh)}$$

$$|x| < 10^{100}, -1 \leq |\tan(x)| < 1 \text{ (for tanh)}$$

Returned value:

Each of these functions returns a value equivalent to its corresponding hyperbolic function. When an error occurs, 0 is returned.

Example:

```
double y;
...
y = sinh(1.0);          /* y=1.1752011936 */
y = cosh(1.0);         /* y=1.5430806348 */
y = tanh(1.0);         /* y=0.761594156 */
```

asinh, acosh, atanh

Numeric function

Format:

```
double asinh(x);
double acosh(x);
double atanh(x);
double x;
```

Purpose:

Each of these functions returns a value equivalent to its corresponding inverse hyperbolic function.

$$\sinh^{-1}x = \log_e (x + \sqrt{x^2 + 1})$$

$$\cosh^{-1}x = \log_e (x + \sqrt{x^2 - 1})$$

$$\tanh^{-1}x = 1/2 \log_e (1 + x/1 - x)$$

The value 33 is assigned to `errno` when the calculation range of `x` is exceeded. The following shows the allowable calculation range for `x`.

$$|x| < 5E+99 \quad (\text{for asinh})$$

$$1 \leq x < 5E+99 \quad (\text{for acosh})$$

$$-1 < x < 1 \quad (\text{for atanh})$$

Returned value:

Each of these functions returns a value equivalent to its corresponding inverse hyperbolic function. When an error occurs, 0 is returned.

Example:

```
double y;
...
y = asinh(1.0);          /* y=0.881373587 */
y = acosh(2.0);         /* y=1.316957897 */
y = atanh(0.5);         /* y=0.5493061443 */
```

pow

Numeric function

Format:

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

Purpose:

This function raises x to the power of y . When $y=0$, 1 is returned. When $x=0$ and y is negative, when x is a negative value and y is not an integer, or when an overflow occurs in the result, the 33 is assigned to `errno`.

Returned value:

This function returns the result of raising x to the power of y . When an error occurs, 0 is returned.

Example:

```
double x=2.0, y=3.0, z;
    :
z = pow(x, y);           /* z=8.0 */
```

sqrt

Numeric function

Format:

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

Purpose:

This function returns the square root of x . When x is a negative value, 33 is assigned to `errno`. The following shows the allowable calculation range for x .

$$0 \leq x < 10^{100}$$

Returned value:

This function returns the square root of x . When an error occurs, 0 is returned.

Example

```
double y;
    :
y = sqrt(2.0);           /* y=1.4142135624 */
```

exp

Numeric function

Format:

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

Purpose:

This function returns the exponent function of x (e^x). If x is outside of the range of $x \leq 230.2585092$, 33 is assigned to `errno`.

Returned value:

This function returns the exponent function of x (e^x). When an error occurs, 0 is returned.

Example:

```
double y;
    ⋮
y = exp(1.0);           /* y=2.7182818285 */
```

log, log10

Numeric function

Format:

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

Purpose:

The `log` function returns the natural logarithm of x ($\log_e x$), while `log10` returns the common logarithm of x ($\log_{10} x$). If $x \leq 0$, 33 is assigned to `errno`.

Returned value:

The `log` function returns the natural logarithm of x , while `log10` returns the common logarithm of x . When an error occurs, 0 is returned.

Example:

```
double y;
    ⋮
y = log(1000.0);       /* y=6.907755279 */
y = log10(1000.0);    /* y=3.0 */
```

angle

Numeric function

Format:

```
void angle(n);
    unsigned n;                /* Angle mode */
```

Purpose:

This function specifies the angle mode for trigonometric and inverse trigonometric functions.

0 : DEG (degrees)
1 : RAD (radians)
2 : GRAD (grads)

When the specification for *n* is not 0, 1, or 2, a value of 33 is assigned to *errno*.

Example:

```
double y;
    :
    angle(0);                /* DEG specification */
y = sin(90.0);              /* y=1.0 */
    angle(1);                /* RAD specification */
y = sin(1.570796327);      /* y=1.0 */
```

beep

Format:

```
void beep(n);
    unsigned n;                /* Buzzer mode */
```

Purpose:

This function is used to control the volume of the buzzer sound as follows:

0 : low volume beep
1 : high volume beep

When the specification for *n* is not 0 or 1, a value of 33 is assigned to *errno*.

Example:

```
beep(0);                    /* Specifies low volume beep */
```

clrscr

Format:

```
void clrscr( );
```

Purpose:

This function clears the display and moves the cursor to the home position (upper left corner).

Example:

```
clrscr( );
```

Execution result

The screen is cleared and the cursor is moved to the home position.

gotoxy

Format:

```
void gotoxy(x, y);  
int x;           /* x-coordinate */  
int y;           /* y-coordinate */
```

Purpose:

This function moves the cursor to the position on the virtual screen (32 columns × 8 lines) specified by x and y. The origin of the virtual screen is the upper left corner (0, 0), and the ranges for the x and y coordinates are as follows:

$$0 \leq x \leq 31$$
$$0 \leq y \leq 7$$

Specifying a coordinate outside of this range results in 33 being assigned to errno.

Example:

```
gotoxy(10, 2);  
/* Specifies cursor location (10,2) */
```

getch

Format:

```
int getch( );
```

Purpose:

This function returns the character code of a single character entered from the keyboard. This function cannot be used for input of **ANS**, **MEMO**, **IN**, **OUT**, **CALC**, **ETC**, function keys or **SHIFT** followed by a one-key command.

When the key buffer is empty, the computer stands by for further input. The cursor is shown on the display, but characters are not displayed as they are input.

Returned value:

This function returns the character code equivalent to a single character entered from the keyboard.

Example:

```
int x;  
:  
x = getch();
```

line, linec

Format:

```
void line(x1, x2, y1, y2);
void linec(x1, y1, x2, y2);
    int x1;           /* Start coordinate */
    int y1;           /* Start coordinate */
    int x2;           /* End coordinate */
    int y2;           /* End coordinate */
```

Purpose:

The line function draws a straight line from a start point (x1,y1) to an end point (x2,y2), specified using the coordinates on the virtual screen (192 × 64 dots). The origin of the virtual screen is the upper left corner (0, 0), and the ranges for the x and y coordinates are as follows:

$$0 \leq x1 \leq 191$$
$$0 \leq y1 \leq 63$$
$$0 \leq x2 \leq 191$$
$$0 \leq y2 \leq 63$$

Specifying a coordinate outside of this range results in 33 being assigned to errno. The linec function operates the same as the line function, except that it erases lines between the specified coordinates.

Example:

For a detailed example of this function, see Chapter 4 in Part 1 of this manual.

```
line(10, 10, 50, 50);
    /* Draws straight line from start
    coordinate(10,10) to end coordinate
    (50,50)*/
linec(10, 10, 50, 50);
    /* Erases straight line from start
    coordinate(10,10) to end coordinate
    (50,50) */
```

Chapter 8 Error Message Tables

1. Command Error Messages

Error Message	Meaning	Action
Illegal command	Illegal format in command.	Recheck command format.
Buffer overflow	Logical command line exceeds 255 characters.	Keep logical command lines 255 characters or less.
No file	Attempt to LOAD a file that does not exist (displayed upon execution of FLIST command only).	LOAD a file before executing FLIST.

2. Syntax Analysis Error Messages

Error Message	Meaning	Action
Syntax error	Number of elements in array outside of specified number, or attempt to use a value not defined by #define.	Check syntax.
Unknown character	Attempt to use illegal character as identifier, etc.	Check character.
Illegal char constant	Character constant not single character which can be displayed, or not escape sequence. Character constant not included within single quotation marks.	Check character string.
Illegal string	Character string not characters which can be displayed, or not escape sequence. Character string not included within double quotation marks. String exceeds 255 characters.	Check character string.
Illegal escape sequence	Characters following \ do not form a valid escape sequence.	Check escape sequence. Valid escape sequences are: \'·\"·\\·\·\ddd·\xdd· \a(07h)·\b(08h)·\f(0Ch)· \n(0Ah)·\r(0Dh)·\t(09h)

Error Message	Meaning	Action
Arithmetic overflow	Floating point constant outside of expressible range.	Check floating point constant.
Illegal preprocessor	Preprocessor syntax error.	Check preprocessor.
Too many include	Nested #include's exceed 10 levels.	Keep nesting 10 levels or less.
Illegal storage class	Specified storage class cannot be used.	Check storage class or check position of specification.
Illegal type	Attempt to use illegal type specifier. Void type is specified as type specifier by data definition.	Check type specifier.
Illegal struct/union	Struct or union nesting cannot be performed	Check structure declaration of struct or union.
Illegal pointer DCL	Pointer declaration has more than one * specification. Pointer of type specifier void cannot be declared.	Check pointer declaration.
Illegal function DCL	Attempt to specify struct/union for function return value. Attempt to declare function within function form declaration or struct/union structure declaration. Attempt to use name of standard function in function definition or declaration.	Check function definition or declaration.
Illegal array DCL	Number of element specification in array declaration not an integer or not a positive value. Array too large (exceeds 65535 bytes). Attempt to declare array as local variable without size reservation.	Check array declaration.
Illegal initialization	Attempt to initialize functions. Attempt to initialize extern declaration, parameter, or union. Attempt to initialize array or structure using other than external definition.	Check initialization.

Error Message	Meaning	Action
Illegal argument DCL	Attempt to declare function as parameter.	Check argument declaration for function definition.
'identifier' undefined	Attempt to use undefined identifier.	Check identifier. If not defined, add appropriate definition.
'identifier' redefined	Attempt to define identifier already defined.	Check identifier. If already defined, use another name.
'identifier' not a function	Attempt to use identifier as a function without declaring it as a function.	Check identifier.
'identifier' not a variable	Attempt to use identifier as a variable without declaring it as a variable.	Check identifier.
Illegal sizeof	sizeof operand not identifier.	Check sizeof operand.
Illegal cast	Attempt to cast void, struct, union type.	Check cast type name.
Illegal break	Attempt to use break statement in other than do, for, or while statement.	Use break statement inside of do, for, or while statements only.
Illegal continue	Attempt to use continue statement in other than do, for, or while statement.	Use continue statement inside of do, for, or while statements only.
Too complex expression	Expression too complex.	Break expression down.
Too many nest	Attempt to nest if, do, for or while statements deeper than 25 levels. System stack overflow caused by too much depth in multiple statements.	Eliminate some nesting.
Too many break/continues	Number of break statements and continue statements within do, for or while statements exceed a total of 25.	Ensure that total of break and continue statements do not exceed 25.
P-code overflow	Overflow of P-CODE area (program too large).	Increase size of code area (see page 26).
Symbol overflow	Overflow of SYMBOL table (too many identifiers and constants).	Increase size of symbol area (see page 26).

3. Program Execution Error Messages

Error Message	Meaning	Action
'main' not found	Computer unable to find main function.	Load file that includes main function.
Illegal 'main'	Attempt to specify argument for main function.	Do not specify argument for main function.
Too many initializers	Too many initialized elements.	Check initial values.
'identifier' undefined	Attempt to use undefined identifier.	Define identifier.
'identifier' not a function	Attempt to use identifier as a function without declaring it as a function.	Check identifier.
'operands' incompatible types	Attempt to use incompatible operand types in expressions or for return values. Display appears as follows when relevant operator does not exist: ' ? : '.	Check operand types.
'operand' Illegal operand	Operand type not correct. Display appears as follows when relevant operator does not exist: ' ? : '.	Check operands.
'operand' Illegal value	Operand on right side. Display appears as follows when relevant operator does not exist: ' ? : '.	Check operands.
Illegal index	Attempt to use negative value for array subscript, or declared number of elements exceeded.	Check subscript specification.
Illegal function call	Number of arguments for called function does not match number of arguments declared during definition of function. Argument type cannot be converted to parameter type.	Check number of arguments and argument types.
Argument list too big	Argument list exceeds 128 bytes.	Decrease number of arguments.
Illegal stream pointer	Attempt to use an illegal stream pointer. Open file not referenced. Attempt to write to file or device opened for reading. Attempt to read from file or device opened for writing.	Check stream pointer.

Error Message	Meaning	Action
Mathematical arithmetic error	Attempt to perform illegal mathematical operation, such as division by 0. Argument of standard function exceeds allowable range. Format error generated for float/double type data.	Check numeric values.
Arithmetic overflow	Calculation result exceeds allowable range.	Check calculation.
Illegal argument	Argument of standard function exceeds allowable range.	Check arguments.
Global overflow	Overflow of global area (total size of externally defined variables and stack variables within functions too large).	Increase size of stack area (see page 26).
Stack overflow	Overflow in stack area (argument list and stack used by internally defined variables too small due to return calls).	Increase size of stack area (see page 26).
Abort	Execution interrupted by abort standard function.	

4. errno Error

errno Value	Meaning
2	<ul style="list-style-type: none"> ① Improper filename specification. ② Specified file cannot be found.
5	<ul style="list-style-type: none"> ① Error in I/O device operation. ② Problem with drive device.
9	<ul style="list-style-type: none"> ① Incorrect stream pointer. ② Open file not referenced. ③ Attempt to write to file or device opened for reading. ④ Attempt to read from file or device opened for writing.
12	Overflow in work area (I/O buffer) or file area (RAM file).
13	<ul style="list-style-type: none"> ① Attempt to open file that is already opened. ② Attempt to open file with other than S identifier. ③ Attempt made to open other than RS-232C file for both read and write. ④ Attempt to open printer for reading. ⑤ File specified in remove or rename is other than disk file or file stored in memory. ⑥ Attempt to execute remove or rename while file is open.
17	Newname specification in rename function already used as filename.
18	Different devices for oldname and newname specifications in rename function.
19	I/O impossible.
24	Too many open files.
28	No more area on disk available for writing.
33	<ul style="list-style-type: none"> ① Attempt to perform illegal mathematical operation, such as division by 0. ② Argument of standard function exceeds allowable range. ③ Format error generated for float/double type data.
34	Calculation result exceeds allowable range.
35	Attempt made to write to disk file which is write protected.
36	Disk needs formatting.
37	<ul style="list-style-type: none"> ① Overflow of RS-232C input buffer. ② Framing error detected at RS-232C port. ③ Parity error or overflow error detected at RS-232C port.
38	Weak batteries.

5. General Error Messages

Error Message	Meaning	Action
Illegal file name	Filename incorrectly specified.	Check filename.
Illegal device	Attempt to specify device which cannot be used.	Check device name.
Different device	Old filename and new filename devices different in rename operation.	Check filename.
Illegal file discrimination	Attempt to specify file with illegal identifier.	Check filename.
Too many files	Too many files opened.	Close some files.
File not found	Specified file not found.	Check filename.
File already exists	Attempt to load file with name of file already loaded (rename operation only).	Check filename or erase currently loaded file using NEW command.
Memory overflow	Overflow in work area (I/O buffer) or file area (RAM buffer).	Increase work area or file area size (see page 26).
Open error	Attempt to open file that is already open.	Check filename.
I/O not ready	Input/Output not possible.	Check I/O connection. Ensure that power is switched ON.
Write protect error	Attempt to output to file which is write protected.	Cancel write protection.
FDD no space	Attempt to write to a disk that is already full.	Erase unnecessary files from disk or use another disk.
I/O read/write error	I/O device operation error.	Check I/O device.
FDD format error	Disk requires formatting.	Format disk.
FDD drive error	Abnormal disk drive operation.	Disk contents may be damaged.
RS-232C buffer overflow	Overflow of RS-232C input/output buffer.	Set RS-232C baud rate to slower speed, or change XON/XOFF specification.
RS-232C framing error	Framing error detected at RS-232C port.	Check RS-232C connection and data transfer procedure.

Error Message	Meaning	Action
RS-232C po error	Parity error or overflow error detected at RS-232C port.	Check RS-232C connection and data transfer procedure. Set baud rate to slower speed.
Low battery	Battery power low.	Replace batteries or change to AC adaptor.
C undefined error	Hardware problem or possible memory abnormality caused by C program execution.	Perform NEWALL operation.

Index

`%c` 46, 50
`%` construction 46
`%d` 44
`%f` 46, 51, 61
`%x` 46, 50
`&x` 51, 70
`\n` 33, 43, 96
2-dimensional array 49, 61

A

a (append) 71
abort 149
Abort 172
abs 157
absolute value 157
access 128
access mode 128
acos 159
acosh 161
addition/subtraction operators 102
address 51, 70, 110
ampersand 70
angle 51, 164
angle mode 164
angle unit 158
angular measurement 51, 158
`ANSI` key 17
Approximation of pi 87
argument 43, 45, 66, 126, 138, 141
Argument list too big 171
argument type declaration 66
arithmetic operators 5
Arithmetic overflow 169, 172
array 49, 108
array elements 101
array initialization 108
array memory area 119
ASCII codes 46, 96
asin 159

asinh 161
assignment operators 101
associativity of operators 102
asterisk 36, 142
atan 159
atanh 161
auto 65, 94, 105
auto variables 106
AUTO.EXE 36

B

Backspace 98
BAT 34
batch file 33
beep 164
BEL 98
Binary mode 129
binominal operators 101
bit AND 75, 101, 102
bit fields 114
bit logical operators 5
bit OR 101, 102
bit shift operators 5
bit XOR 101, 102
braces 52, 56, 103
brackets 49
break 65, 94, 104
BREAK mode 150
break statement 105
breakpt 150
`BRK` key 39
BS 98
BS error 27
Buffer overflow 168
buzzer sound 164

C

[c] 9, 11, 13, 15, 17, 21, 24
C area 26, 27

- C area capacity 26
 C area memory status display 27
 C editor 8, 123
 C file 12
 C function 9
 C identifier 12, 35
 C interpreter 8
 C language 4
 C program character notation 97
 C undefined error 175
CA key 19
 CAL mode 9, 19, 37
 calculation range 159
 calculator 9
 calloc 119, 125
 carrier return 33, 98, 129
 case 65, 94
 cast 75
 cast operator 101
 char 47, 65, 94, 95, 126
 character % 139
 character array 117
 character code 50, 97, 166
 character code (decimal) 97
 character code (hexadecimal) 97
 character code table 97
 character constant 96
 character format 50
 character operator stack 28
 character string 43, 62
 character string input/output 116
 clearerr 146
 clrscr 165
 code area 26, 27
 comma 45, 48, 126
 command 122
 Command Error Messages 168
 command line 13
 comments 45, 94, 126
 comment line 45
 common logarithm 163
 comparison 155
 compiler 6
 compiler language 6
 compiling 6, 8
 compound statements 103
 condition 104
 conditional jump 104
 conditional operators 101
 conjugate complex number 53
 Console 125, 127, 128
 const 65, 94
 constant 95
 constant expression 114
 constant type char 96
 continue 65, 94
 continue statement 105
 control characters 98
 control structures 4, 37, 103
 control structures for jump 104
 control structures for repeats 104
 conversion characters 139, 143
 conversion specifications 138, 142
 cos 158
 cosh 160
 cosine curve 79
 counter variable 59
 CR 98
 create a program 14
 [C/S] 12
 creating file 30
 cursor 22
 cursor home position 165
- D**
- [data] 11
 data bank function 9
 DATA EDIT mode 11
 data expression 96
 data length 95
 data storage pointer 134
 data structure 113
 data type 95
 debugging 37


-
- decimal constants 99
 - decimal integers 46, 143
 - decimal value 44
 - declared variables 105
 - decrement operator 57
 - decrementing 57
 - default 65, 94
 - #define 115
 - #define identifier string 28
 - #define statement 57, 95
 - #define token string 28
 - degree 51
 - dest pointer 154
 - device name 128
 - Different device 174
 - directory area 28
 - Disk 125, 127
 - [disk] 11
 - do 65, 94
 - “do~while” loop 58
 - double 47, 54, 65, 94, 95, 126
 - double argument 142
 - double precision floating-point constants 100
 - double precision value 67
 - double quotation marks 33, 100, 125
 - double-precision floating point 47, 54
 - draws a straight line 167

 - E**
 - EDIT 13, 14, 123
 - [edit] 11, 19
 - EDIT command 19
 - editor 8, 19
 - else 65, 94
 - else~if statement 104
 - end of file 71
 - endless loop 104
 - enum 65, 94
 - enumeration type 114
 - EOF 71, 116, 129
 - equality operator 101, 102
 - Eratosthenes method 74

 - errno 115
 - errno value 147, 148, 173
 - error messages 5, 168
 - escape sequence 96, 98, 138
 - ESC key 9
 - execute programs 15, 32
 - execution 6, 8
 - exit 149
 - exp 163
 - exponents 100, 143, 163
 - expression 59, 104
 - expression input/output 117
 - extension 35
 - extern 65, 94, 105
 - extern declaration 112, 115
 - external storage device 11



 - F**
 - factorial 68
 - false 52, 55
 - fclose 64, 71, 130
 - FDD drive error 174
 - FDD format error 174
 - FDD no space 174
 - feof 145
 - ferror 145
 - fflush 144
 - fgetc 131
 - fgets 135
 - field width 139, 142
 - File already exists 174
 - file area 26, 27
 - file input 71
 - file input/output functions 116
 - File not found 174
 - file output 71
 - FILE structure 127
 - Filename 122, 125, 128
 - flag 139
 - FLIST 13, 123
 - float 47, 65, 94, 95, 126
 - floating point 45, 46, 47, 51
-

- floating point calculation 6
 floating point value 79, 157
 floating-point constants 100
 floating-point number 143
 floppy disk 11, 122
 fopen 64, 71, 127
 for 65, 94
 for, while, do ~ while statement 104
 "for" loop 58
 forced conversion 75, 101
 Form feed 98
 format 138, 141, 142
 formula memory 9
 fprintf 138
 fputc 134
 fputs 136
 fractional part 140
 fread 136
 free 153
 free area 28
 fscanf 99, 141
 functions 43, 111, 126
 function argument operations 101
 function definition 65
 function key menu 9
 function key menu commands 10
 function keys 10
 function name 43, 65, 95
 function type 65
 fwrite 137
- G**
- Gauss' Method 91
 general error messages 174
 general memory status display 26
 getc 131
 getc (stdin) 130
 getch 166
 getchar 50, 57, 64, 130
 gets 64, 134
 global area 28
 Global overflow 172
- global variables 68, 69, 85, 105, 107
 goto 65, 94
 goto statement 105
 gotoxy 165
 grad 51
 Greatest Common Measure 58
- H**
- hexadecimal constants 99
 hexadecimal format 50
 hexadecimal integer 46, 143
 hexadecimal value 44, 46
 HT 98
 Horizontal tab 98
 hyperbolic function 160
- I**
- identifier 8, 35, 113
 identifier character strings, constants 28
 'identifier' not a function 170, 171
 'identifier' not a variable 170
 'identifier' redefined 170
 'identifier' undefined 170, 171
 if 65, 94
 if (condition) statement 52, 104
 if ~ else statement 52, 104
 "if" selection statement 52
 Illegal argument 172
 Illegal argument DCL 170
 Illegal array DCL 169
 Illegal break 170
 Illegal cast 170
 Illegal char constant 168
 Illegal command 168
 Illegal continue 170
 Illegal device 174
 Illegal escape sequence 168
 Illegal file discrimination 174
 Illegal file name 174
 Illegal function call 171
 Illegal function DCL 169


- Illegal index 171
- Illegal initialization 169
- Illegal 'main' 171
- Illegal pointer DCL 169
- Illegal preprocessor 169
- Illegal sizeof 170
- Illegal storage class 169
- Illegal stream pointer 171
- Illegal string 168
- Illegal struct/union 169
- Illegal type 169
- imaginary number 53, 55
- #include 29, 87, 115
- increment operator 57
- incremental/decremental operators 5
- incrementing 57
- inequality 101
- initial mode 9
- initialize 43
- input field 142
- Input/output function 126
- Input/Output Function Errors 147
-  key 22
- INSERT mode 22
- int 47, 65, 94, 95, 99, 126
- int type 99, 111
- integer 47
- integer constant 46, 99
- integer declaration 47
- integer notation 47
- interpreter 6, 8, 13
- inverse hyperbolic function 161
- inverse trigonometric function 159
- I/O buffer 28
- I/O not ready 174
- I/O read/write error 174

- K**
- [kill] 11

- L**
- l 139, 142

-  key 22
-  key 22
- library functions 111
- line 167
- linec 167
- linefeed 129
- linking 6, 8
- [llist] 12
- LOAD 13, 24, 122
- [load] 11
- load command 30
- LOAD file table 28
- load programs 32
- local variable 29, 68
- log 163
- log10 163
- logical AND 101, 102
- logical NOT 101
- logical operators 5
- logical OR 101, 102
- long 47, 54, 65, 94, 95, 99, 126
- long argument 140, 142
- long type integer 99
- loop counter value 61
- loops 55
- Low battery 175

- M**
- machine language 5, 6, 8
- macro 157
- macro definition 115
- main () 65
- main function 65, 111
- 'main' not found 171
- malloc 119, 151
- Martian animation 80, 82
- Mathematical arithmetic error 172
- MD-100 interface unit 11
- Mean and variance 89
- member operators 113
- members 113
- [memory] 12, 25

- memory address 5
 memory area 25, 27
 memory capacity 26
 Memory display 75
 memory functions 119, 126
 memory map 28
 Memory overflow 174
 memory status 25
 memory storage area 105
 memsize 119
 menu display 10
 key 10, 13
 MENU mode 19
 [merge] 12
 module 65
 modulus (remainder) 48, 101
 Monte Carlo method 87
 MS-DOS 4
 multiple execution unit 103
 multiple solution 53
 multiple statements 52, 103
 multiplication/division operator 102
- N**
- name 57, 127
 [name] 11, 16, 31
 natural logarithm 163
 negative integers 99
 nested loop 60
 NEW 13, 123
 NEWALL 27
 [newc] 11, 19
 newline 43, 63, 98
 [next] 24
 NL 98
 No file 123, 168
 normal characters 138, 142
 NOT 101
 Null 98, 116
 NULL pointer 116, 119
 number of elements 152
 numeric functions 118, 126
 numeric value 44
 numeric variable data area 28
- O**
- octal constants 99
 octal integer 143
 octal notation 98
 octal value 44
 one-key command 17
 Open error 174
 'operand' illegal operand 171
 'operand' illegal value 171
 'operands' incompatible types 171
 opening and closing files (S) 117
 operators 100
 order operator 102
 other standard functions 120, 126
 output characters 42
 overflow check 154
 overlay sheet 18
 OVERWRITE mode 22, 23
- P**
- P-code area 28
 P-code overflow 170
 parameters 125
 parentheses 45, 126
 period 139
 Perpetual calendar 76
 pointers 5, 51, 70, 109
 pointer (stream) 127
 pow 162
 power 162
 precedence 100
 precedence of operators 102
 precision 139
 preprocessor 115
 [preset] 12, 36
 preset file 12, 36
 primary operators 101
 prime numbers 74

-
- Printer 125, 127, 128
printf () 33, 45, 64, 138
process functions 126
program execution error messages 171
program module 65
programming language 4
Pseudo-random number generator 85
putc 133
putchar 64, 71, 132
puts 64, 135
- Q**
quadratic equation 53
- R**
r (read) 71
radian 51
RAM file 122, 125, 127, 128
random numbers 85
real number 53
RECALL function 17
recursive function call 68
recursive recall 112
register 65, 94, 105
register variables 106
relational operators 5, 54, 101
remove 146
rename 147
repeat control 104
replacement text 115
Reserved words 65, 94
return 65, 94
Return value type 126
RS-232C 122, 125, 127, 128
RS-232C buffer overflow 174
RS-232C framing error 174
RS-232C interface 11
RS-232C po error 175
RUN 6, 13, 15, 122
RUN command 29, 32, 38
RUN execution 150
- S**
"S" identifier 12, 37, 127
[save] 11
scanf () 51, 64, 99, 141
scientific functions 6
scope 105
[search] 23
search function 23
selection statement 52
semicolons 58, 103, 126
separate compilation 30
sequential data file 11
[set] 9, 12
shift operator 102
short 47, 65, 94, 95, 126
signed 65, 94
signed decimal notation 140
simultaneous linear equations 91
sin 158
sine curve 79
single character 46
single quotation marks 96
single-character input/output 116
single-precision floating point 47
sinh 160
sizeof 65, 94
Solution of simultaneous
linear equation 91
source 154
space 142
space characters 142
specify the memory status 26
sprintf 138
sqrt 162
square root 162
square value 67
sscanf 99, 141
Stack 28
stack area 26, 27
Stack overflow 172
standard error output stderr 149
standard functions 43, 64, 111, 116, 126
-

- standard function name 94
 standard functions for display 119
 standard functions for
 keyboard control 119
 standard input device name 122
 standard input stdin 130
 standard library function 33
 standard output device name 122
 standard output stdout 132
 statements 66, 103
 static 65, 94, 105
 stderr 115
 stdin 115, 130, 141
 stdout 115, 138
 storage area 110
 storage classes 105
 store a program in a file 16
 strcat 64, 154
 strchr 156
 strcmp 64, 155
 strcpy 64, 154
 string 46, 47
 string constants 100
 string functions 118, 126
 strlen 64, 153
 struct 65, 94, 113
 structure 6, 113
 structure declaration 113
 structure members 101
 switch 65, 94
 switch ~ case statement 104
 symbol area 26, 27
 Symbol overflow 170
 symbol table 28
 syntax analysis error messages 168
 Syntax error 168
- T**
- tag 113
 tan 158
 tanh 160
 Text mode 129
- Too complex expression 170
 Too many break/continues 170
 Too many files 174
 Too many include 169
 Too many initializers 171
 Too many nest 170
 trace function 37
 TRACE mode 13, 37, 38, 124, 150
 trace off 40
 trace on 38
 trigonometric function 158
 trinomial operators 101
 TROFF 13, 40, 125
 TRON 13, 124
 TRON command 38
 true 52, 55
 type declaration 95, 112
 type of value 70
 typedef 65, 94
- U**
- unary minus operators 99
 unary operators 101
 unconditional jump 105
 ungetc 143
 union 6, 65, 94, 113
 UNIX 4
 Unknown character 168
 unnamed C file 11
 unnamed file 16, 34, 43, 123
 unsigned 47, 65, 94
 unsigned char 140
 unsigned constants 99
 unsigned decimal integer 139, 143
 unsigned decimal notation 140
 unsigned hexadecimal notation 140
 unsigned int 99
 unsigned octal notation 140
 Update Mode 129
 user area 27, 28
 user area capacity 26

V

value 48, 51
variable 47
variable address 109
variable errno 147
variable name 49, 95
variable storage location 70
variable table 28
variable type 47
variance 89
void 65, 94
void function 112
volatile 65, 94

W

w (write) 71
while 62, 65, 94
while, for, do~while loops 104
while, if break 105
while, if continue 105
while loop 55
work area 26, 27
Write protect error 174