TEXAS INSTRUMENTS COMPACT COMPUTER 40 USER'S GUIDE

ADDENDUM

The Wafertape™ Digital Tape Drive is not available.

ADDENDUM

Texas Instruments **Compact Computer 40 User's Guide**

Caution: *Read the information on static electricity on page 1-4 before working with your computer.*

The following notes provide additional information about using and programming your Texas Instruments Compact Computer 40.

Page 1-7

The optional AC adapter referred to is the Texas Instruments model AC9201; use *only* the AC9201 with your CC-40.

Page 5-34

Add the following sentence to the end of the third paragraph.

I/O *error-type* 255 is returned as 0.

Page 5-43

The following paragraph provides more information on GETMEM.

The memory reserved by GETMEM can be released during program execution by a call to RELMEM. Any of the following actions cause the reserved memory to be released.

- Editing the program or subprogram.
- Entering a NEW, OLD, RENUMBER, RUN, SAVE, or VERIFY command.
- Listing the program to a peripheral device.
- Calling the ADDMEM or CLEANUP subprogram.
- Turning the system off or pressing the reset key.

Page 5-48

The example at line 290 should be as shown below.

```
290 IF A$="Y" THEN COUNT=COUNT+1:DISPLAY
    AT(4),"Enter value: ";:GOTO 400
```

Page 5-65

The following paragraph provides more information on *status-variable.*

When CALL KEY is executed, the keyboard is scanned for input. *Status-variable* is used to store a value that represents the status of the scan. A value of 0 means no key was pressed. A value of 1 means a different key was pressed since the last time the keyboard was scanned for input (e.g., since CALL KEY, KEY$, INPUT, or ACCEPT was last executed). A value of −1 means the same key was pressed.

(continued)

The following program segment provides more information on the use of the CALL KEY subprogram. This segment prompts twice for a key to be pressed. To determine that the responses are distinct, the status variable is compared to 1 (S<>1) in lines 520 and 560.

```
500 PRINT "MORE ENTRIES? (Y OR N)"
510 CALL KEY(K,S)
520 IF S<>1 THEN 510
530 IF K=ASC("Y") OR K=ASC("y") THEN 400
540 PRINT "END SESSION? (Y OR N)"
550 CALL KEY(K,S)
560 IF S<>1 THEN 550
570 IF K=ASC("Y") OR K=ASC("y") THEN STOP
```

Page 5-133

The example for line 330 should be as follows.

```
330 SUB PAYCHECK(DATE,Q,SSN,PAYRATE,TABLE(,))
```

Marks the beginning of a subprogram. The variables DATE, Q, SSN, PAYRATE, and the array TABLE with two dimensions may be used and/or have their values changed in the subprogram and their corresponding arguments in the calling statement changed. However, if the corresponding argument of DATE, Q, SSN, or PAYRATE is enclosed in parentheses in the CALL statement, the value of that argument cannot be changed. The corresponding array argument of TABLE *must* be passed by reference in the CALL statement and therefore any of its values can be changed in the subprogram.

Page I-4 (Appendix)

In the fifth line from the top, the address is the second breakpoint set, as shown below.

where *nnnn* is the address of the second breakpoint set.

Pages I-3,4

The following problem can occur when a breakpoint is set in assembly language software by either the breakpoint command or the single step command. If the register file stack is at register 9 and a breakpoint is executed, the program counter is destroyed. The most significant byte of the program counter is changed to the least significant byte plus one.

For example, if the breakpoint is set at address 1235_{16}, the breakpoint message is `3635 st 09:`, where `st` is the appropriate status register value. The PC command can be used to change the program counter back to the correct address and program execution can continue.

To avoid this problem, use one of the following methods to be sure that the stack pointer does not point to register 9 when a breakpoint is executed.

- Begin the register file stack at register A_{16} instead of at register 1.
- If the position of the stack cannot be altered, add PUSH and POP instructions to the code to ensure that the stack does not use register 9 within the section of code being debugged. The added instructions can be removed after the code is debugged.
- Write the assembly language software such that the stack pointer stays on even byte boundaries.

Page I-5

The last paragraph should begin as shown below.

> Then B can be typed to continue program execution...

Page K-6

The fifth error message paragraph should be changed as shown below.

- Invalid character in statement. For example “%”,“?”,“[”,“]”,“{”, etc., are valid only within quoted strings or in an IMAGE or REM statement.

TEXAS INSTRUMENTS COMPACT COMPUTER 40 USER'S GUIDE

This book was developed and written by:

Nancy Bain Barnett
John T. Baker
Robert E. Whitsitt, II

With contributions by:

Tom M. Ferrio
Bud Gerwig
Craig Benson
David G. Thomas

Artwork and layout were coordinated and executed by:

Gaither and Davy, Inc.

ISBN 0-89512-057-7
Library of Congress Catalog Card Number 82-74472

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER I
GETTING ACQUAINTED

Introduction

The Texas Instruments Compact Computer Model CC-40 is an affordable, textbook-size computer that puts the computing power of large systems right at your fingertips. With its advanced calculation capability, enhanced BASIC language, and flexible peripheral expansion, the CC-40 lets you perform data processing anywhere you need to.

In addition to its computer capabilities, the CC-40 provides calculator features from simple addition and subtraction to complex problem solving. Plug-in cartridges offer built-in problem solutions for business, engineering, scientific, and mathematical needs. A wide range of accessories, including a printer and a high-speed *Wafertape*™ digital tape drive, increases the capabilities and versatility of the CC-40.

The growing importance of computers in our lives is making it increasingly necessary that everyone become familiar and comfortable with computers. With the CC-40 you will discover how easy it is to use a computer to solve problems.

CHAPTER I
GETTING ACQUAINTED

Features

- Textbook-size computer with a 2.5MHz TMS70C20 CMOS 8-bit microprocessor and 34K (over 34,000) bytes of Read-Only Memory (ROM).
- Memory expandable with installation of cartridges.
- Preprogrammed *Solid State Software*™ cartridges.
- International-language prompting in selected cartridges (available separately).
- *Constant Memory*™ feature that retains information stored in memory when the computer is turned off.
- Long battery life with inexpensive AA alkaline batteries.
- Optional AC adapter to extend battery life.
- Standard typewriter keyboard layout plus quick-entry numeric key pad and special keys.
- Keys that repeat when held down for approximately half a second.
- 31-character 5-by-8 dot matrix Liquid Crystal Display (LCD) that scrolls to show an 80-character line.
- Adjustable display contrast.
- ASCII character set, including both upper- and lower-case alphabetic characters. Special display characters include Greek letters, Japanese characters, plus characters you can define.
- Peripheral port for printers, mass storage devices, and other accessories.
- Enhanced BASIC language for easy programming and quick problem solving.
- Descriptive error messages.
- Fold-out tilt-stand for easy desk-top use.
- Audible tone for use in prompting.
- Assembly language programming capability with the optional Editor/Assembler cartridge.

CHAPTER I
GETTING ACQUAINTED

Using this Manual

This manual explains how to use all the features of the CC-40. This chapter explains how to set up and care for the CC-40 and provides an introduction to the keyboard. Reading it will provide you with the fundamentals necessary to operate the CC-40.

The second chapter discusses how to use the CC-40 as a calculator. You'll find that the CC-40 can be used as a calculator for your quick calculation needs in addition to its use as a computer.

The third chapter discusses using optional cartridges and peripheral devices. *Solid State Software*™ cartridges offer a wide variety of professionally written programs and *Memory Expansion* cartridges are used to increase the memory capacity of the CC-40. Peripheral devices can greatly increase the flexibility and versatility of the CC-40.

The fourth chapter provides an overview of the CC-40's BASIC programming capability. If you are unfamiliar with BASIC, the book *Learn BASIC: A Guide to Programming the Texas Instruments Compact Computer* is available from local dealers. Even if you already know how to program, reading chapter 4 carefully will help you to use the features of the CC-40 which are not available on other computers.

The final chapter is an alphabetical reference section for BASIC programming with detailed explanations of each command, statement, and function.

The appendices contain information such as mathematical functions and error messages. Technical data is also provided for experienced computer users.

Care of the CC-40

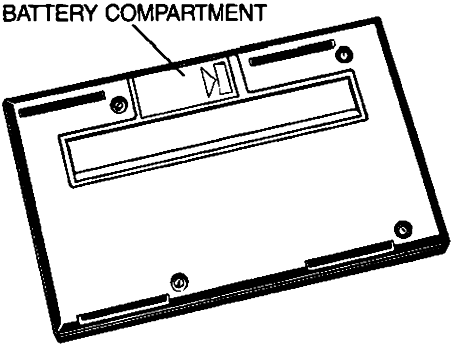
With proper care your CC-40 will give you years of trouble-free operation. Treat your CC-40 and cartridges with the same care you give other precision electronic products.

- Avoid exposing your CC-40 to moisture, extreme temperatures, or dust.
- Use a damp lint-free cloth to clean your CC-40. Do not use solvents.
- Do not place objects other than cartridges in the cartridge port.

CAUTION: Electronic equipment can be damaged by static electricity discharges. To remove static charges, touch a metal object (such as a doorknob, a desk lamp, etc.) before working with your computer, connecting peripheral devices, handling a cartridge, inserting a cartridge, etc. Failure to do so may cause damage to the cartridge or the computer.

Setting Up the CC-40

The battery compartment is located on the back of the CC-40 as shown in the illustration.

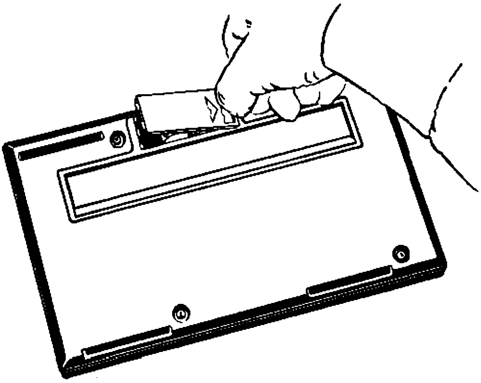


Battery Replacement

Your CC-40 requires four AA alkaline batteries. Replace the batteries when the low battery indicator is displayed or the display remains blank and cannot be made visible with the contrast control. Several hours of battery use remain when the low battery indicator first appears.

The battery replacement procedure is as follows.

1. Insert your thumbnail in the recess in front of the arrow. Press in the direction indicated by the arrow until the opposite end of the cover is released. Then remove the cover.

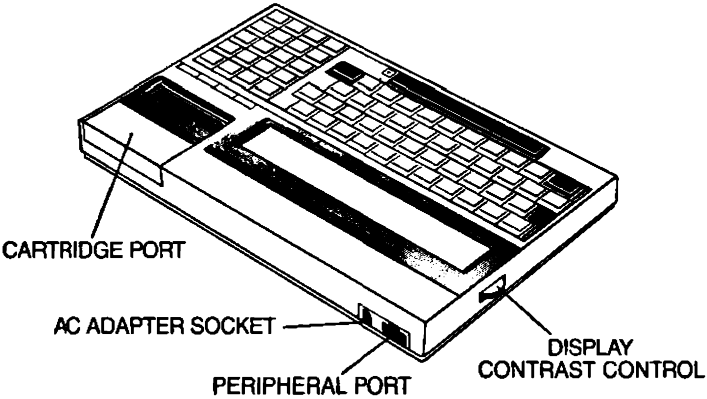


2. Remove the four batteries from the battery compartment.
3. Place four AA alkaline batteries in the battery compartment, as indicated in the bottom of the compartment.
4. Slide the arrow end of the cover under the unnotched side of the compartment. Press the loose end of the cover down until it snaps into place.

Peripheral Port

The peripheral port is located next to the AC adapter socket on the back of the CC-40. The larger opening is the peripheral port. It is used to connect peripheral devices to the CC-40.

Caution: The electronic components of the peripheral port can be damaged by discharges of static electricity. To avoid damage, do not touch the connector contacts or expose them to static electricity.



AC Adapter Socket

The AC adapter socket is next to the peripheral port. The AC adapter allows the computer to operate from standard AC power instead of batteries.

Display Contrast Control

The display contrast control is located on the left side of the CC-40. It allows you to adjust the display contrast for your viewing angle.

Cartridge Port

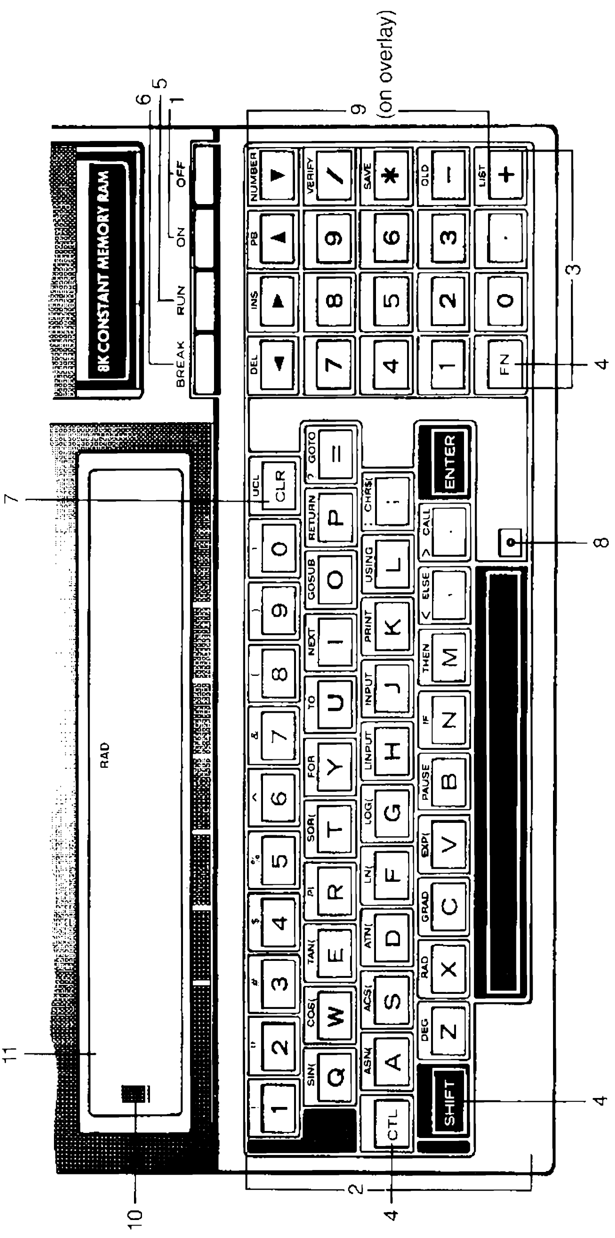
The cartridge port, which accepts *Solid State Software* and *Memory Expansion* cartridges, is in the upper right corner of the CC-40. See chapter 3 for information on installing and using cartridges.

Keyboard Tour

The CC-40 keyboard resembles a typewriter keyboard. The CC-40, however, has additional keys to perform calculations, edit program lines, and quickly access BASIC commands, statements, and functions. The edit keys and all of the keys that enter characters in the display repeat as long as they are held down.

In this manual a key with a label that contains more than one character is represented by brackets [ ] with the symbol of the key printed inside the brackets such as **[ON]**, **[ENTER]**, **[SHIFT]**, and **[FN]**. A key whose label is a single character is in bold face, such as **A**, **8**, and **→**.

The next two pages show a diagram of the keyboard with its parts labeled. Following that are explanations of each part along with examples to help you become familiar with the features of the CC-40. Place the overlay provided with your CC-40 over the keys. The text and the examples that follow refer to this overlay. Try the examples to see what happens. You can't damage the CC-40 by pressing its keys.


1
2
3
4
4
4
4
5
6
7
8
9
(on overlay)
10
11
8K CONSTANT MEMORY RAM
BREAK
RUN
ON
OFF
RAD
CLR
UCL
ENTER
SHIFT
CTL
FN
DEL
INS
PB
NUMBER
VERIFY
SAVE
OLD
LIST
GOTO
RETURN
GOSUB
NEXT
TO
FOR
SQR(
PI
TAN(
COS(
SIN(
ASN(
ACS(
ATN(
LN(
LOG(
LINPUT
INPUT
PRINT
USING
CHR$(
CALL
ELSE
THEN
IF
PAUSE
EXP(
GRAD
RAD
DEG

1. **[ON]** and **[OFF]** Keys
2. Typewriter Keys
    - Alphanumeric Keys
    - Space Bar
    - **[SHIFT]** Key
    - **[UCL]** (Upper Case Lock) Key
    - **[ENTER]** Key
3. Numeric Key Pad
    - Numeric Keys
    - Arithmetic Operator Keys
    - Edit Keys
4. Shift, Function, and Control Keys
    - **[SHIFT]** Key
    - **[FN]** (Function) Key
    - **[CTL]** (Control) Key
5. **[RUN]** Key
6. **[BREAK]** Key
7. **[CLR]** (Clear) Key
8. Reset Key
9. BASIC Keyword Keys (on overlay)
10. Cursor
11. Display

1. [ON] *and* [OFF] *Keys*

The [ON] key turns the CC-40 on. When the computer is turned on, memory content is checked for loss of information. If no loss of memory is detected, the flashing cursor is displayed in column one. This cursor informs you that you may enter information from the keyboard.

If there is a memory loss, the memory is cleared and the message `System initialized` is displayed. To proceed, press the [CLR] key. The display is cleared and the flashing cursor appears in column one.

If programs and data *may* have been lost from memory, the CC-40 warns you with the message `Memory contents may be lost.` (Pressing the reset key usually causes this message to be displayed.) You may then choose either to use what is in memory or to clear memory. To clear the warning message from the display, press [CLR]. Then if you wish to clear the contents of memory, type **NEW ALL** or **NEW** and press [ENTER].

The [OFF] key turns the CC-40 off. To conserve battery life, the *Automatic Power Down*™ (APD) feature turns the computer off automatically after approximately ten minutes of waiting for input. The effect is the same as pressing the [OFF] key. However, the CC-40 cannot be turned off by either the APD feature or the [OFF] key when a program is running.

Note: The APD feature can be disabled by typing **CALL POKE(2096,1)** and pressing [ENTER]. The APD feature is restored when the system is initialized or you enter **NEW ALL** or **CALL POKE(2096,0).**

When the CC-40 is turned off, the *Constant Memory*™ feature retains programs, user-assigned strings, and assembly language subprograms stored in memory. It does not retain the values of variables or the contents of the display.

2. Typewriter Keys

The keyboard is used to enter information such as data, BASIC program lines, messages, and formulas into your computer. The information typed appears in the display, character by character, just as you type it. The computer does not use the data in the display until you press **[ENTER]**, so the display contents can be edited, as described later, before you press **[ENTER]**.

These keys are arranged like the keys of a standard typewriter. The number and punctuation keys are across the top row.

Alphanumeric Keys

In computer usage, the letter, digit, sign, and symbol keys are called alphanumeric keys. Alphanumeric keys are used the same as standard typewriter keys. For example, type the word **hello** by pressing the appropriate keys and `hello` appears in the display. To clear the display, press the **[CLR]** key. Enter the number **456398** by pressing the top row of numeric keys. To clear the display, press **[CLR]**.

Space Bar

The space bar places a space in the display. If the space bar is pressed while the cursor is positioned over another character, that character is replaced with a blank.

[SHIFT] Key

You can type capital letters and the special characters above the numeric and punctuation keys by pressing **[SHIFT]** and the desired key. Note that there is only one **[SHIFT]** key.

When **[SHIFT]** is pressed, the `SHIFT` indicator appears in the display to indicate that the shift state is in effect so that the next key you press is shifted. The shift state is in effect until another key is pressed. The shift state can be cancelled by pressing **[SHIFT]** a second time. The **[SHIFT]** key may be held down simultaneously with the desired keys.

For example, type the following using the **[SHIFT]** key as needed.

How are you?

To clear the display, press **[CLR]**.

[UCL] (Upper Case Lock) Key

There is no shift lock key on the CC-40. However, the Upper Case Lock [UCL] key is similar to a shift lock key. Upper Case Lock causes all alphabetic keys to be interpreted as upper case and the [SHIFT] key to be ignored if it is pressed before an alphabetic key. Upper Case Lock does not affect the operation of punctuation or number keys.

To activate the Upper Case Lock, press the [SHIFT] key and then the [CLR] key. The UCL indicator appears in the display to indicate that the Upper Case Lock state is in effect. Press [SHIFT] [UCL] again to cancel the Upper Case Lock state.

Type the following after pressing [SHIFT] [UCL]. You will have to use the [SHIFT] key to type the !, ", and #.

```
UCL TYPES CAPS, NOT !, ", OR #
```

To clear the display, press [CLR].

[ENTER] Key

The [ENTER] key is located at the right of the keyboard. [ENTER] is used to indicate that you have finished typing the data on the current line and are ready for it to be processed.

3. Numeric Key Pad

The right side of the keyboard consists of digit keys arranged in calculator format, edit keys for correcting entries, arithmetic operator keys, and the [FN] key.

Numeric Keys

The numeric keys are arranged in calculator format to provide for rapid entry of numbers. This set of numeric keys and the set on the top row of the typewriter keyboard can be used interchangeably. The special characters above the numeric keys of the typewriter keyboard are also available with the keys on the numeric key pad.

Type the following using either set of numeric keys.

```
2301.5713
```

To clear the display, press [CLR].

Arithmetic Operator Keys

The arithmetic operator keys are used to perform simple arithmetic. To add, use the + key. To subtract, use the − key. To multiply,-use the * key. To divide, use the / key. You can raise a number to a power or calculate a root by pressing the **[SHIFT]** and ^ keys. These keys are described in chapter 2.

For example, to find $2+2^5$, type the following.

2+2^5

To display the answer, 34, press **[ENTER]**. To clear the display, press **[CLR]**.

Edit Keys

The edit keys, →, ←, **[SHIFT] [INS]**, **[SHIFT] [DEL]**, and **[SHIFT] [PB]**, are used to change the characters in the display. The cursor does not have to be repositioned to the end of the line before pressing **[ENTER]**.`

→(right arrow) moves the cursor one position to the right. The cursor does not erase or change characters as it passes over them. When the cursor reaches the right side of the display, holding → down scrolls the *display* to the left until the end of the 80-character line is reached.

←(left arrow) moves the cursor one position to the left. The cursor does not erase or change characters as it passes over them. If the cursor is at the beginning of a line, holding ← down does not move it.

For example, type the following.

Information is 555-1212

Press and hold → until all the displayed text has scrolled to the left. Notice the left arrow indicator in the display is turned on to inform you that text has scrolled to the left. Next hold down ← until the text is again displayed. The left arrow indicator is turned off.

To change the text in the display to `Information:   555-1212`, press ← or → until the cursor is positioned in the column after `Information`. Press **[SHIFT]** : and the space bar twice. To clear the display, press **[CLR]**.

[SHIFT] [INS] (insert) is used to insert characters. Each subsequent key that you type is inserted at the current cursor position, shifting all characters following the inserted character(s) one position to the right. Insertion continues with each character typed until **[ENTER]**, **[CLR]**, **[BREAK]**,←, or → is pressed. If insertions cause the line to exceed its 80-character limit, the characters at the end are lost.

[SHIFT] [DEL] (delete) deletes the character that the cursor is on and shifts all characters to the right of the cursor one position to the left.

For example, type the following (do not press **[ENTER]**).

2 + 2 is not equal to 5.

To change this entry to But 2 + 2 is equal to 4., first position the cursor over the first 2. Press **[SHIFT] [INS]** and then type **But** followed by a space. Move the cursor to the n in not. Press and hold **[SHIFT]** as you press **[DEL]** four times to delete not and the space. Move the cursor over the 5 and type **4**. To clear the display, press **[CLR]**.

[SHIFT] [PB] is used to exchange the last series of characters entered in the display with the current display contents.

For example, type the following.

5+255-375*5

Press **[ENTER]** and the answer, -1615, appears in the display. Press **[SHIFT] [PB]** to recall the series of characters that you entered. To alter the calculation by changing the 375 to 365, position the cursor over the 7 and press **6**. Press **[ENTER]** to display the answer, -1565. To clear the display, press **[CLR]**.

Note: The characters recalled by **[SHIFT] [PB]** are normally the last characters you typed. However, if a PRINT or DISPLAY statement has placed characters in the display (see chapters 4 and 5), then those characters are recalled.

4. Shift, Function, and Control Keys

The **[SHIFT]**, **[FN]**, and **[CTL]** keys can be pressed and released before or at the same time as the keys they modify.

[SHIFT] Key

The **[SHIFT]** key is used to allow the entry of "shifted" characters in the display from both the typewriter keyboard and the numeric key pad. When **[SHIFT]** is pressed, the SHIFT display indicator appears in the display, the same as with the **[FN]** and **[CTL]** keys. The shift state can be cancelled by pressing **[SHIFT]** a second time. If **[SHIFT]** is held down while several other keys are pressed, the SHIFT indicator is turned off after the first key. However, the shift state remains in effect until **[SHIFT]** is released.

[FN] (Function) Key

The **[FN]** (Function) key is used to enter certain BASIC keywords into the display. These keywords appear on the separate keyboard overlay above the alphabetic and punctuation characters. When **[FN]** is pressed, the FN display indicator appears in the display, the same as with the **[SHIFT]** and **[CTL]** keys. The function state can be cancelled by pressing **[FN]** a second time. If **[FN]** is held down while several other keys are pressed, the FN indicator is turned off after the first key. However, the function state remains in effect until **[FN]** is released.

To access one of the BASIC keywords printed on the overlay, press **[FN]** and then the key corresponding to the desired keyword. For example, to find the square root of 9, press the following keys.

[FN] T (which is usually denoted **[FN] [SQR( ]**.)

SQR(appears in the display. Type **9** and **)**. To have the CC-40 process the data you have typed, press **[ENTER]**. The answer, 3, is displayed. To clear the display, press **[CLR]**.

[FN] can also be used to assign a series of characters to the numeric keys. This feature is discussed later in this chapter.

[CTL] (Control) Key

The [CTL] (Control) key allows the use of special control functions. When [CTL] is pressed, the CTL display indicator appears in the display, the same as with the [FN] and [SHIFT] keys. The control state can be cancelled by pressing [CTL] a second time. If [CTL] is held down while several other keys are pressed, the CTL indicator is turned off after the first key. However, the control state remains in effect until [CTL] is released.

For example, press [CTL] and hold down 5 until the display fills with underlines. Press [CTL] → (tab) and the display moves to the next higher tab position. The left arrow display indicator is turned on to show that text has scrolled off the display. To clear the display, press the [CLR] key. A list of all of the character codes that can be accessed with the [CTL] key is in appendix D.

5. [RUN] *Key*

The [RUN] key followed by [ENTER] causes the computer to perform or execute the program that is stored in memory. [RUN] can also be followed by several options. It can be followed by a line number to start program execution at a specific line, a device and filename to load and execute a program from an external storage device, or a program name to run a program from a *Solid State Software* cartridge (see chapter 3).

6. [BREAK] *Key*

The [BREAK] key is used to stop a program that is executing.

7. [CLR] *(Clear) Key*

The [CLR] (Clear) key clears all the characters from the display when no program is running. When a program is waiting for input, the characters in the input field are cleared.

8. Reset Key

Reset is used to restart the computer when a problem occurs in which normal operation is interrupted. Pressing the reset key usually causes the message `Memory contents may be lost` to be displayed. *You* must determine if the situation which caused you to press reset was likely to have destroyed the memory contents. The reset key is flush with the case so that you will not press it accidentally.

9. BASIC Keyword Keys

The BASIC keyword keys provide a convenient entry method for BASIC commands, statements, and functions.

Some BASIC keywords are printed on the separate keyboard overlay. To access these keywords, press the **[FN]** key and the key below the desired keyword.

For example, one of the BASIC keywords is PRINT. To access this keyword, rather than typing each letter, you can press **[FN]** **[PRINT]**. The word `PRINT` appears in the display. Type the following using the BASIC keyword PRINT and the appropriate keys and then press **[ENTER]**.

```
PRINT "The answer is"
```

To clear the display, press **[CLR]**.

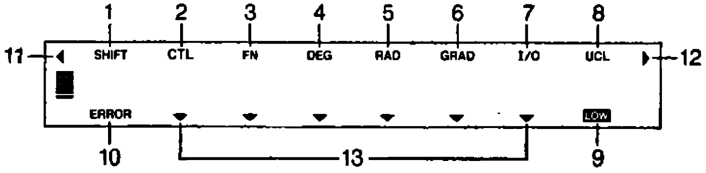
10. Cursor

There are two cursors. One is a flashing rectangle that indicates you can enter data. The other is an underline that indicates the system is waiting for you to acknowledge a pause by pressing **[ENTER]** or **[CLR]**.

11. Display

The display shows 31 characters of the 80-character line.

Eighteen display indicators are provided to indicate certain conditions. These indicators appear if certain controls are in effect, such as SHIFT, DEG, and FN. The display of the CC-40 is shown below with labels and descriptions.



	Indicator	Meaning
1.	SHIFT	The shift state is in effect.
2.	CTL	The control state is in effect.
3.	FN	The function state is in effect.
4.	DEG	The unit of angular measure is degrees.
5.	RAD	The unit of angular measure is radians.
6.	GRAD	The unit of angular measure is grads.
7.	I/O	Peripheral input or output is in progress.
8.	UCL	Upper Case Lock is in effect.
9.	LOW	Battery power is low.
10.	ERROR	An error has been detected.
11.	◀	Text has scrolled off the display to the left.
12.	▶	Text has scrolled off the display to the right.
13.	▼	An indicator that has been set. See chapter 4 and INDIC in chapter 5.

User-Assigned Strings

The user-assigned string feature of the CC-40 allows you to assign text to each of the number keys 0 through 9. Commonly used numbers, expressions, and text assigned to these keys can later be recalled from the keyboard using the [FN] key. The assigned text is retained even when the CC-40 is turned off.

To assign text to a number key use the following procedure.

1. Place up to 80 displayable characters in the display.
2. Hold down [SHIFT] and [FN] simultaneously until both the SHIFT and FN indicators appear in the display. Then release the keys.

3. Press the desired digit key. The characters are assigned to the indicated digit, the display is blanked, and the SHIFT and FN display indicators are turned off.

To display the assigned text, press **[FN]** and the digit key that was assigned. The text is placed in the display, starting at the current cursor position. You may repeat this key sequence as often as you wish to display the assigned text.

For example, assign the message "Frank– –555–1212" to digit key **6** using the procedure below.

Press	Display	Comments
1. **Frank--555-1212**	`Frank--555-1212`	
2. **[SHIFT] [FN] 6**	blank	Hold down both **[SHIFT]** and **[FN]** until the SHIFT and FN indicators appear in the display, then press 6.
3. **[FN] 6**	`Frank--555-1212`	Assigned characters are displayed.

You can delete a user-assigned string by performing the assignment when the display is blank. For example, if **[SHIFT] [FN] 6** is pressed when the display is blank, the string assigned to key **6** is deleted. All user-assigned strings are cleared when **NEW ALL** is entered or the system is initialized.

The following examples illustrate other kinds of text that can be assigned to the number keys.

Examples	Comment

`LIST "50.R=C"`
`RUN "1.MYFILE"`
	Frequently used commands and statements

`FV=PV*(1+i)^n`
`SQR(a^2+b^2)`
	Commonly used mathematical expressions

`100 INPUT "Variable= ";X`
`330 ACCEPT AT(15) BEEP,N`
	BASIC program lines

`179.234*3422.14-A`
`PV/(1+.10)^10`
	Repetitive calculations

`Ambulance 555-1212`
`Budget meeting at 4 pm`
	Memos

CHAPTER II
PERFORMING CALCULATIONS

Introduction

The calculating capabilities of the CC-40 are enhanced by a wide range of built-in mathematical functions. These functions range from simple arithmetic operators to logarithmic and trigonometric functions. You can use these functions in a BASIC program as well as directly from the keyboard. Repetitive calculations are easily performed on the CC-40.

Entering Numeric Data

The number keys, **0** through **9**, are located both on the right side of the console (called the numeric keypad) and across the top row of the keyboard and can be used interchangeably. The decimal point on the numeric keypad and the period on the keyboard can also be used interchangeably.

The value π to eleven decimal places, equal to 3.14159265359, can be included in a calculation by typing **PI**. (Note that the value of π is displayed as 3.141592654.) The value of π is substituted for PI when a calculation is performed.

Negative values are entered into the display by preceding the number with the minus (−) sign located on the numeric keypad.

If you need to correct an erroneous entry, use the ←, →, **[SHIFT] [INS]**, and **[SHIFT] [DEL]** keys. See chapter 1.

Numeric Variable Assignments

Numeric values can be used directly in a calculation or can be assigned to variables for use in later calculations. For example, entering **X=3** stores the value 3 in the variable X. Then X can be used in a calculation such as X^2+3X-2 (entered as **X∧2+3*X−2**), giving a result of 16.

The result of an equation or calculation can also be assigned to a variable. For example, Y=m*x+b assigns the result of the equation m*x+b to the variable Y. The previously defined variables m, x, and b supply the data necessary to evaluate the equation. See chapter 4 for details concerning variable assignments.

Simple Arithmetic Calculations

The +, −, *, /, and [SHIFT] ∧ keys are used to perform simple arithmetic calculations. The following examples show how to use these keys.

Example	Press	Display	Comment
2+3	**2+3 [ENTER]**	5	Addition
145−98	**145−98 [ENTER]**	47	Subtraction
3945 × −5	**3945* −5 [ENTER]**	-19725	Multiplication
7 ÷ π	**7/PI [ENTER]**	2.228169203	Division
$5^{-3.25}$	**5 ∧ −3.25 [ENTER]**	.0053499224	Exponentiation
$\sqrt[3]{8}$	**8∧(1/3) [ENTER]**	2	Root

A problem is evaluated when the [ENTER] key is pressed. You do not need to press the = key.

Note: A negative value cannot be raised to a non-integer power.

Order of Operations

The CC-40 uses standard algebraic hierarchy to evaluate mathematical problems. This evaluation method permits you to enter an equation or problem into the display in an order similar to the way it is written on paper—from left to right.

The CC-40 uses the following order of precedence when evaluating mathematical operations.

Operation	Example	Entry
1. Functions	$\sqrt{25}$	**SQR(25)**
2. Exponentiation	$8^{(1/3)}$	**8∧(1/3)**
3. Unary minus	−17	**−17**
4. Multiplication and division	6*4÷3	**6*4/3**
5. Addition and subtraction	3+7−2	**3+7−2**

When [ENTER] is pressed, the CC-40 employs these rules to evaluate the problem in the display. Operations with the same priority are evaluated from left to right.

For example, to evaluate $2+3*8/2^2-4$, the CC-40 first performs exponentiation, followed by multiplication and division, and then addition and subtraction. This problem is evaluated in the following manner.

Problem	2+3*8/2^2-4
Exponentiation first	2+3*8/4-4
Multiplication/division from left to right	2+24/4-4
	2+6-4
Addition/subtraction from left to right	8-4
	4

You can change the normal order of evaluation by enclosing part of a calculation in parentheses. Any expression in parentheses is evaluated first. For example, to calculate 2*(8+3)*2, the CC-40 first evaluates the contents of the parentheses. This problem is evaluated in the following manner.

Problem	2*(8+3)*2
Contents of parentheses evaluated	2*11*2
Multiplication/division from left to right	22*2
	44

Scientific Notation

Scientific notation allows you to represent very small and very large numbers in an easy-to-read format. Scientific notation is especially useful in scientific and engineering applications, where such numbers are often used.

These numbers are expressed in a format in which a number (the mantissa) is multiplied by 10 raised to a power (the exponent). For example, the sun is approximately 92,900,000 miles away from the earth. This number can be represented in scientific notation as 9.29×10^7 where 9.29 is the mantissa and 7 is the exponent.

Numbers in scientific notation are entered into the display of the CC-40 with the mantissa followed by E (or e) and the exponent. For example, 9.29×10^7 can be expressed as 9.29E+07 or, more conveniently, as 9.29e7. Like other numbers, values expressed in scientific notation must not contain spaces. The following table gives other examples of scientific notation.

Standard Notation	Scientific Notation
-512340000000	-5.1234E+11
.00000000000188	1.88E-12
-.000000000000123	-1.23E-13

Scientific notation allows you to enter and evaluate calculations on numbers with magnitudes as small as ±1E-128 or as large as ±9.9999999999999E+127. The computer automatically displays numbers in scientific notation when more significant digits can be displayed than in the usual ten-digit format.

Mathematical Functions

There are two ways to access the mathematical functions of the CC-40. You can either type the function name or press **[FN]** and the alphabetic key under the desired function shown on the overlay, as described in chapter 1. Note that some functions are not available using the **[FN]** key; you have to type them.

Several of the mathematical functions are described below. For more information on these and the other functions (listed in appendix B), see chapter 5.

When the **[FN]** key is used, the keystrokes shown in the examples describe the function accessed and not the key actually pressed. For example, pressing **[FN]** and then **R** to access PI is shown as **[FN] [PI]**.

Many mathematical functions require an argument. If an argument is required, it must be enclosed in a set of parentheses. For example, ABS(−4) calculates the absolute value of −4. Both the open and close parentheses are required. The message `Unmatched parenthesis` is displayed when a parenthesis is omitted.

If you use the **[FN]** key to place a function name, such as ABS, into the display, the computer provides the open parenthesis. Thus, pressing **[FN] [ABS(]** places `ABS(` in the display. You then provide the argument and closing parenthesis, and press **[ENTER]** to evaluate the expression.

Square Root

SQR is used to calculate the square root of a number.

Example	Press	Display
$\sqrt{5}$	**[FN] [SQR(]**	SQR(
	5)	SQR(5)
	[ENTER]	2.236067978
$\sqrt{25}$	**SQR(25) [ENTER]**	5

The first example illustrates using the **[FN] [SQR(]** sequence to calculate the square root. In the second example, the **[FN]** key is not used.

Logarithms and Antilogarithms

LOG is used to calculate the common logarithm (base 10) of X. Ten raised to a number is used to calculate the common antilogarithm.

Example	Press	Display
LOG(2.4)	**LOG(2.4) [ENTER]**	.3802112417
$10^{1.17}$	**10 ^ 1.17 [ENTER]**	14.79108388

LN is used to calculate the natural logarithm (base e) of X. **EXP** (the exponential function, e^x) is used to calculate the natural antilogarithm of X.

Example	Press	Display
LN(2.4)	**LN(2.4) [ENTER]**	.8754687374
$e^{.25}$	**EXP(.25) [ENTER]**	1.284025417

Trigonometric Functions

The CC-40 evaluates trigonometric functions in either radian, degree, or grad (RAD, DEG, or GRAD) angle units. The RAD angle setting is selected when **NEW ALL** is entered or the system is initialized. The angle setting remains in effect until you change it or the system is initialized.

When you use a trigonometric function, check the angle setting indicator in the display to make certain that the desired angle setting has been selected. If you wish to choose a different setting, you may either use the [FN] key or manually type the desired setting as RAD, DEG, or GRAD.

SIN, **COS**, and **TAN** are used to compute the sine, cosine, and tangent of an angle. The angles are measured in the units of the indicated angle setting. The following example finds the sine of 50 degrees.

Example	Press	Display	Comments
	[DEG] [ENTER]	DEG indicator set	Set degree mode.
SIN(50°)	**SIN(50) [ENTER]**	.7660444431	Sine of 50°.

The formulas for secant, cosecant, cotangent, and hyperbolic functions are given in appendix E.

ASN, **ACS**, and **ATN** are used to compute the arcsine ($\sin^{-1}$), arccosine ($\cos^{-1}$), and arctangent ($\tan^{-1}$) of a value. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using these functions. The following example finds the arcsine of .5 with radians as the trigonometric unit.

Example	Press	Display	Comments
	[RAD] [ENTER]	RAD indicator set	Select radian mode.
ASN(.5)	**ASN(.5) [ENTER]**	.5235987756	Arcsine of .5.

The formulas for arcsecant, arccosecant, arccotangent, and inverse hyperbolic functions are given in appendix E.

Chain Calculations

The CC-40 provides you with the capability to chain keyboard calculations. Chain calculations are often used when the result of one calculation is needed in another calculation. A loss of accuracy occasionally results when you chain calculations. See appendix F for accuracy information.

When the result of a keyboard calculation is displayed, it is followed by the flashing cursor. You can press any of the arithmetic operators (+, −, *, /, or ∧), or the **[SHIFT] [INS]**,→, or ← keys to use the displayed result in another calculation. If an alphanumeric key is pressed, the display is cleared and that key entered into the display. If **[SHIFT] [INS]** is pressed, the cursor is automatically positioned to column 1, where characters can be inserted.

For examplé, the formula $h=\sqrt{a^2+b^2}$ is used to calculate the length of the hypotenuse of a right triangle. When a = 3 and b = 4, h can be calculated by entering **SQR(3 ∧ 2 + 4 ∧ 2)**. To illustrate how to chain calculations, this problem is divided into three steps. First, 3 ∧ 2 is calculated. Next, the result of 4 ∧ 2 is added to the displayed result of 3 ∧ 2. Then the square root of that sum is calculated.

Press	Display	Comment
3∧2	3∧2	
[ENTER]	9	Result of 3^2 followed by the flashing cursor.
+4∧2	9+4∧2	
[ENTER]	25	Result of $3^2 + 4^2$ followed by the flashing cursor.
[SHIFT] [INS]		Flashing cursor moves to column 1.
[SQR( ]	SQR(25	Insert SQR(.
→→	SQR(25	Move cursor past the 5.
)	SQR(25)	Enter close parenthesis.
[ENTER]	5	Result of SQR($3^2 + 4^2$) followed by the flashing cursor.

If no other calculations are to be performed, press **[CLR]** to clear the display.

Repetitive Calculations

The CC-40 provides two ways to perform repetitive calculations. The [SHIFT] [PB] key can be used to display the contents of the previous display, or the user-assigned keys can be used to store and recall frequently used expressions and data.

Playback

The playback feature can be used when you need to perform the same calculation repeatedly with different values. When a keyboard calculation is performed, the result is displayed and you can press [SHIFT] [PB] to recall the calculation that was last in the display. The calculation can be altered using the edit keys and a new result obtained.

For example, suppose you want to find the equivalent Celsius temperature for a Fahrenheit temperature. The formula for converting to Celsius is shown below.

C = (F − 32)*5/9

To convert 212° Fahrenheit to Celsius, enter the following.

(212 − 32)*5/9

The answer, 100, is displayed.

To change another Fahrenheit reading to Celsius, press [SHIFT] [PB] to display the last line that was entered, (212-32)*5/9. Use the edit keys to change 212 to 100 and press [ENTER]. The result of this calculation, 37.77777778, is displayed.

To change another Fahrenheit reading to Celsius, press [SHIFT] [PB] to display the last line that was entered, (100-32)*5/9, and use the edit keys to change the 100 to another temperature.

This type of repetitive calculation is appropriate when you do not have to enter values for variables. When you enter a value, that value becomes the contents of the last display, and will be displayed when [SHIFT] [PB] is pressed. To perform a repetitive calculation with variables, use the user-assigned keys.

User-Assigned Keys

User-assigned keys are used to reduce time spent typing commonly used expressions or strings. You can assign a string of up to 80 characters to each of the number keys **0** through **9** and recall those characters with the **[FN]** key (see chapter 1). This capability is especially useful for storing mathematical expressions.

For example, the following formula can be used to calculate the inverse secant.

SGN(X)*ACS(1/X)

You can calculate the inverse secant for different values of X by assigning this formula to a numeric key (0-9) and then entering different values for X. The following example illustrates assigning the formula to key 1 and finding the inverse secant of 5 and 10 in radians.

Press	Comments
SGN(X)*ACS(1/X)	Place string in the display.
[SHIFT] [FN] 1	Hold down both **[SHIFT]** and **[FN]** until the SHIFT and FN indicators appear in the display, then press **1**.
[RAD] [ENTER]	Select radian mode.
X=5 [ENTER]	Define X to be 5.
[FN] 1	Recall string.
[ENTER]	Displays 1.369438406.
X=10 [ENTER]	Define X to be 10.
[FN] 1	Recall string.
[ENTER]	Displays 1.470628906.

All user-assigned strings are cleared when **NEW ALL** is entered or the system is initialized.

CHAPTER III
USING OPTIONAL CARTRIDGES & PERIPHERAL DEVICES

Introduction

The memory capacity and problem-solving capabilities of the CC-40 can be greatly increased by installing optional *Solid State Softwåre*™ or *Memory Expansion* cartridges in its cartridge port. The peripheral devices available for use with the CC-40 also enhance its versatility and usefulness.

Solid State Software cartridges give you rapid access to professionally-written programs in areas such as mathematics, engineering, and finance. The programs in these cartridges take full advantage of the alphanumeric display capability of the CG-40 to provide descriptive prompting for data entries and labeling of results. Illustrative examples and detailed instructions for using each of the cartridge programs are provided in the owner's manual supplied with each cartridge.

Memory Expansion cartridges can increase the Random Access Memory (RAM) capacity of your CC-40. Like the *Solid State Software* cartridges, *Memory Expansion* cartridges are installed in the cartridge port of the CC-40.

Peripheral devices available for use with the CC-40 allow you to expand the capabilities of the computer. The peripheral expansion flexibility of the CC-40 enables you to expand your system to print information, store and retrieve programs and data, communicate with other computers, print graphs and charts, and view data on a CRT (television) display. The computer communicates with peripheral devices through the TI *HEX-BUS*™ Intelligent Peripheral Interface (the peripheral port). The *HEX-BUS* interface is a standardized interconnection system with a uniform set of cabling conventions, control signals, and message structures.

Caring for Cartridges and Peripherals

Even though the cartridges and peripherals are durable devices, you should handle them with care. Follow these precautions when handling cartridges or peripherals.

- **BE SURE THAT YOUR BODY IS FREE OF STATIC ELECTRICITY.** Prior to handling any cartridge or peripheral, touch some metal object to discharge any static electricity you may be carrying.
- Keep the cartridge port cover secure on the computer to keep the cartridge port dust-free.
- Keep the contact area of the cartridges clean. A buildup of debris or foreign particles on the contacts can impair their operation. Keep the cartridges stored either in the original container or in the computer's cartridge port.
- Use a cotton swab soaked in alcohol to clean the cartridge and cartridge port contacts when necessary. After the alcohol has dried, remove any remaining lint with a clean, soft-bristled brush.

  CAUTION: Do not use any other liquid substance to clean the contacts.
- Check the peripheral manuals for special maintenance instructions for the peripherals.

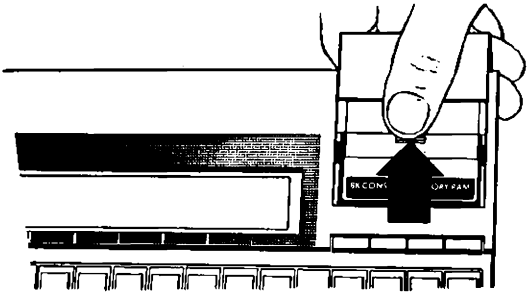
Installing or Replacing a Cartridge

Both *Solid State Software*™ and *Memory Expansion* cartridges are installed in the cartridge port of the CC-40. Turn the computer off when installing or replacing a cartridge. Installing a cartridge while the computer is on may result in memory loss.

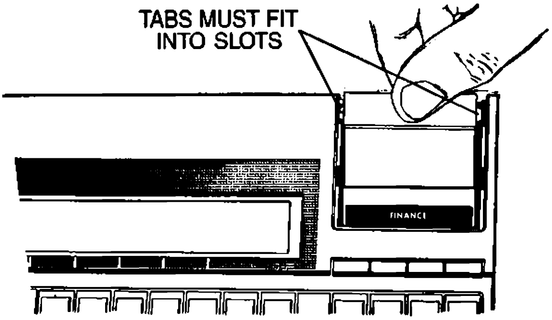
Use the following procedure when installing a cartridge.

1. Turn the computer off.

2. Slide the cartridge port cover back and remove it from the computer as shown in the diagram below.



3. If a cartridge is already installed, remove it by pushing the cartridge away from the keyboard until it is released. Lift the cartridge from the cartridge port.
4. Lay the cartridge to be installed in the cartridge port, with the cartridge name facing up and toward the keyboard.
5. Press firmly on the back of the cartridge and slide it toward the keyboard until the tabs on the cartridge enter the slots provided and the cartridge locks into place. Replace the cartridge port cover and check that its tabs also fit into the slots.



Using *Solid State Software* Programs and Subprograms

The programs in **some** *Solid State Software*™ cartridges start running as soon as you turn on the computer. Instead of a blank display with the cursor, you see a message. See the owner's manual that comes with the cartridge for instructions on how to proceed.

You must run the programs in other cartridges by typing **RUN**, quotation marks, the name of the program, and quotation marks, and then pressing **[ENTER]**. The owner's manual gives the name of the program that you need to run and describes how to use the program.

Most cartridges contain several programs and subprograms, each described in the owner's manual. You can run the programs from the keyboard or from a program. Subprograms are called from programs. Refer to chapter 4 for how to run programs and use subprograms in a program.

To run a program in a *Solid State Software* cartridge, type **RUN**, quotation marks, the program name, and quotation marks, and then press **[ENTER]**. For example, you can run the normal distribution program in the Statistics cartridge by typing **RUN "normal"** and then pressing **[ENTER]**. If the cartridge is installed, `NORMAL DISTRIBUTION` is displayed. If the cartridge is not installed, `Program not found` is displayed.

Using *Memory Expansion* Cartridges

Memory Expansion cartridges can be installed in the cartridge port of the CC-40 to expand the resident memory and allow you to write and use larger programs. Cartridge memory is appended to the resident memory when the ADDMEM subprogram is called (see chapter 5). This link between resident memory and cartridge memory is broken when **NEW ALL** is entered, the reset key is pressed, or the cartridge is removed.

Set-up Instructions for Peripherals

Setting up the CC-40 peripherals is a simple process. First you connect the device to the *HEX-BUS*™ Intelligent Peripheral Interface; then you check its operation. This section describes the steps involved in setting up peripherals.

Connecting Peripheral Devices

The devices in the TI *HEX-BUS* system have identical eight-pin recessed connectors for the cable through which they communicate. The computer has one such connector, while each peripheral device has two of them so that a series of devices may be attached to the computer.

Before connecting any peripherals, turn off the computer. Then wait for *all* peripheral activity to cease before you turn off the peripherals. You may link peripherals to the computer in any order. The first peripheral is plugged directly into the computer using a cable. Then another cable is plugged into the other connector on the peripheral. The other end of the cable is plugged into the next peripheral you are connecting. The plugs are keyed so that you can insert them only one way. The last peripheral has one connector free.

Peripherals are normally arranged in a stack next to the computer, using the cables supplied with the peripheral devices. Longer cables are available separately if you prefer to arrange the peripherals differently.

CAUTION

> To prevent damage, disconnect all devices before moving any part of the *HEX-BUS* system. Even though the computer and peripherals are light and portable and easily carried in a briefcase, the cables and connectors are subject to accidental strain if not detached. For shipment over long distances, repack the system securely, preferably in its original packing materials.

Checking the Operation of a Peripheral

The IO subprogram in the CC-40 can be used to determine if a peripheral is attached correctly to the computer. The IO subprogram is accessed by entering **CALL IO** with the device number of the peripheral you are testing. For example, **CALL IO(1,1)** is used to test peripheral device 1 and **CALL IO(7,1)** is used to test peripheral device 7. Refer to the peripheral manuals for the device numbers of your peripherals.

Use the following procedure to check if a peripheral is connected correctly.

1. Turn on the peripherals and the computer. All attached peripherals should be turned on for proper operation.
2. To test a peripheral device, enter **CALL IO** with the number of the device. For example, to test device 20 (the RS232 peripheral), type **CALL IO(20,1)** and press **[ENTER]**. The ERROR indicator is turned on and the following message is then displayed if the peripheral is attached correctly.

   ```
   I/O error 4 "20"
   ```

   Note: **CALL IO(20,1)** is an attempt to end access to the RS232 peripheral. Since it has just been turned on at this point, the device is not yet enabled for access. In returning error code 4 (device not open), the peripheral is operating as it should.
3. Press **[CLR]** to clear the error message and restore the cursor. The peripheral is ready for use.

If the correct error code is not displayed when the **CALL IO** instruction is entered, the device may not be connected properly. Check the cabling between the computer and the peripheral. Refer to the peripheral manual for more information.

If the I/O display indicator stays on while the rest of the display remains blank, the system has "locked up" at some point in the test sequence. The computer cannot respond to input from the keyboard while in this state. Turn the peripheral off momentarily to clear the condition. Then check the cabling connection and try the operational check once more. If this procedure does not correct the situation, you may have a hardware problem in the cable or the peripheral. Refer to *In Case of Difficulty* in the peripheral manual for information.

CHAPTER IV
BASIC PROGRAMMING

Introduction

The BASIC programming language was developed at Dartmouth College in the 1960s. The word *BASIC* is an acronym for *B*eginner's *A*ll-purpose *S*ymbolic *I*nstruction *C*ode. BASIC is the language most commonly used on personal and home computers and is increasingly accepted on larger machines. The Texas Instruments Compact Computer Model CC-40 uses an advanced version of BASIC called CC-40 BASIC. Programs written in other versions of BASIC, including TI BASIC and TI Extended BASIC (used on the Texas Instruments Home Computer), may have to be modified for use on the CC-40.

This chapter provides an introduction to BASIC that enables you to use the BASIC programming features immediately. It contains overviews of what a program is, how to write a program, the rules and syntax of CC-40 BASIC commands, statements, and functions, what input and output are, the various parts of a program, how to edit a program, how to save a program, and how to debug (or find the errors in) a program. Each of the elements that makes up the CC-40 BASIC language is mentioned, and some are explained in detail. For details on syntax and additional examples, see chapter 5, which is an alphabetical listing of all the CC-40 BASIC commands, statements, and functions.

If you have not used BASIC before, the book *Learn BASIC: A Guide to Programming the Texas Instruments Compact Computer* is available from local dealers. The only way to learn to program is to actually program. Try the examples in this book. Don't worry about making mistakes when you begin. You can always cancel any operation by pressing **[BREAK]** and **[CLR]**.

If you are already familiar with some version of BASIC, this chapter is a quick refresher. Be sure to review the topics that are marked with the TI logo (ti) for features of CC-40 BASIC that may differ from other versions of BASIC.

Getting Started

A program is a series of instructions that the computer can perform. You tell the computer what to do by typing instructions on the keyboard and then pressing **[ENTER]**. These instructions are performed only when you run or execute the program by typing **RUN** and pressing **[ENTER]**. When you run a program, the instructions are performed or executed one at a time.

The computer has its own set of words, called keywords, that it knows how to perform. There are not many such words, but taken together these words let you perform virtually any computational task.

As a simple example, you can write a program to multiply 25 times 7 and print the answer. A program to do this is shown in the following section.

Writing, Running, and Listing a Program

Turn on the CC-40. Either a message is displayed or there is a flashing cursor in column 1. To clear the message from the display, press the **[CLR]** key. Then in both cases, type **NEW** and press **[ENTER]**.

Type the following line exactly as shown, including spaces. As you type, the cursor moves to the right to show where the next character is placed. If you make a mistake typing, use the edit keys (described in chapter 1) to correct the line, or press the **[CLR]** key to start over.

```
100 PRINT 25*7
```

Press **[ENTER]** for the computer to store the instruction in its memory. Type the next line exactly as shown.

```
110 PAUSE
```

Press **[ENTER]** for this instruction to be stored.

The computer has now stored two instructions or lines in its memory. To have the computer perform these instructions, you must run the program by typing **RUN** and pressing **[ENTER]**.

The computer calculates 25 times 7 and prints the answer, 175, in the display. It then pauses so you can see the answer. The result is preceded by an underline cursor in column 1. The underline cursor indicates that the computer is waiting for the **[CLR]** or **[ENTER]** key to be pressed. To leave the pause, press **[CLR]**. The flashing cursor appears in column 1 to indicate that you can enter information from the keyboard.

Note: You can leave any BASIC program that is running by pressing **[BREAK]**.

To look at the program in the CC-40, type **LIST** and press **[ENTER]**. The CC-40 prints the first line of the program.

```
100 PRINT 25*7
```

Press **[ENTER]** to display the next line.

```
110 PAUSE
```

Press **[ENTER]** again and the flashing cursor appears on a blank line. The blank line means there are no more program lines stored in memory. You can now proceed to enter information.

If you are in the middle of a listing and do not want to see the rest of a program, press **[BREAK]** to end the listing.

BASIC Programming Procedures

The two-line program you just entered, executed, and listed shows many of the procedures to follow as you write BASIC programs. Remember—before you enter a program in the CC-40, type **NEW** and press **[ENTER]** to be sure that memory has been cleared.

The rules for entering program instructions are described briefly in the following sections.

Lines and Line Numbering

100 PRINT 25*7 is called a line of a program. This line instructs the computer to calculate 25 times 7 and print the answer. The computer recognizes * as multiplication and the word PRINT as a task to perform.

The 100 is the line number. Every line of a program must have a line number from 1 through 32766, followed by a space. The CC-40 executes program lines in numerical order, regardless of the order in which they are entered.

It is good programming practice to have unused line numbers between the lines of a program. You can then insert additional lines in the program. For example, suppose you want to perform one more calculation in the program, 25 times 6. Enter the following two lines.

```
105 PAUSE
108 PRINT 25*6
```

If you **LIST** the program, you will find that the two lines have been added to the program in memory. The computer sequences the lines by their line numbers. To run the program, be sure the display is clear and then type **RUN** and press **[ENTER]**. When the computer pauses while displaying the first answer, 175, press **[ENTER]** to display the next answer, 150. Then press **[ENTER]** again.

If you enter a line with the same line number as one already stored in memory, the new line replaces the old one. For example, type the following line and then press **[ENTER]**.

```
100 PRINT 25*25
```

When you list the program, you will find the last line entered as 100 is the one that is now stored in memory.

Note: When entering any information into the computer, always press **[ENTER]** after you have finished typing. In the rest of this chapter, any references to enter any information assume that the **[ENTER]** key is pressed after you have finished typing.

Keywords

Following the line numbers in the program just executed are English words that the CC-40 recognizes and knows how to perform or execute. All program lines contain characters that resemble algebraic formulas and/or English words. These English words correspond to single tasks and are called keywords. When entering keywords in the CC-40, you can type them using either upper- or lower-case characters. When the program is LISTed, these words are always displayed in upper-case letters. The different types of keywords are discussed in the following sections.

Statements

Statement keywords are elements of a program line that cause an action, such as PRINT and PAUSE. Statement keywords must be followed by a space in the program line. For example, the CC-40 recognizes the statement 10 PRINT 2 but not 10 PRINT2. The CC-40 performs statements in a program only when you execute the program.

Many statement keywords can also be executed immediately by entering them without a line number. The statement is executed as soon as the **[ENTER]** key is pressed. For example, enter the following in your CC-40.

```
PRINT 25*7
```

The answer 175 is displayed immediately.

A list of all the statement keywords, indicating which ones can be executed immediately as well as used in program lines, is given in appendix A. For simple calculations such as the one above, it is usually more practical to use the calculator features of your CC-40 (refer to chapter 2). However, when you want to perform the same series of calculations repeatedly, you can often save time and effort by writing these calculations as a program for the CC-40 to perform.

Functions

The function keywords perform specialized routines and return a value. Most functions require that a value (called an argument) be given to the function. There are function keywords for many mathematical functions such as square root, logarithm, and sine.

All of the functions available with the CC-40 are listed in appendix B and many are discussed in this chapter in sections that deal with similar instructions. All of the functions can be used in program lines and most of them can be executed immediately.

Commands

The command keywords, such as NEW and LIST, are always executed immediately. These keywords may not be used in a program line. The commands available on the CC-40 are listed in appendix A and are discussed throughout this chapter.

Using a Function in a Program

The program below calculates and displays the square roots of the first 25 whole numbers. To obtain these answers without a program would require making 25 separate calculations. Enter the program shown below in your CC-40. (Remember to type **NEW** and press **[ENTER]** before you enter the program, and to press **[ENTER]** after you have typed each line.)

```
100 FOR A=1 TO 25
110 PRINT A; SQR(A)
120 PAUSE 1
130 NEXT A
```

In line 100, the FOR statement sets up a loop, a group of statements that are repeated a specific number of times. The loop consists of the statement immediately following the FOR statement and all the statements down to a NEXT statement. In this case, the letter A is the counter that starts at 1 and goes TO 25 by ones.

Line 110 tells the computer to print the value of the counter A and the square root of the counter. The first time line 110 is executed, the value of the counter is 1.

Line 120 tells the computer to pause for 1 second after it prints an answer so that you have time to see it.

Line 130 is the last statement in the loop. The counter, A, is incremented by one and the CC-40 goes back and repeats lines 110, 120, and then 130, where the counter is again incremented. The loop is repeated until the counter is incremented past the number 25, which is the number following the word TO in line 100. Thus, this loop is executed 25 times.

Enter **RUN** to execute the program. The CC-40 displays the first twenty-five whole numbers and their square roots, pausing one second to display each.

Ending a Program

A program normally stops running after the last line in the program has been executed. However, if you wish, you can enter an END statement as the last statement in your program. The STOP statement can be entered anywhere in your program that you want the program to stop execution.

Kinds of Entries

Everything entered in the CC-40 is determined by the CC-40 to be one of two kinds of entries.

1. An entry that begins with a number from 1 through 32766 followed by a space and an alphabetic character (or an at sign, the underscore, or an exclamation point) is treated as a program line that is stored in memory.
2. Any other entry is assumed to be a command, a statement, or a calculation that is executed immediately. Calculations are discussed in chapter 2.

Program Lines

This section describes the requirements and restrictions of program lines including line numbering, line length, lines containing remarks, and multiple statement lines.

Line Numbering

Each line in a program must begin with a number followed by a space. Line numbers can be any integer from 1 through 32766. It is good practice to number lines in multiples of 10 in case you need to insert lines.

Automatic Line Numbering

You can have the CC-40 supply line numbers by entering the command **NUM** (for **NUMBER**). The CC-40 displays 100 followed by a space. The cursor is positioned where the first character of the line starts. After you type the statement and press **[ENTER]**, the CC-40 displays 110 followed by a space and waits for you to enter the statement for that line. When you have finished entering all of the program lines, press either **[ENTER]** or **[BREAK]** when the next line number appears.

You can also use NUM to tell the CC-40 where to start numbering and what increment you want. For example, entering **NUM 10,20** starts numbering the lines at 10 and increments each succeeding line number by 20.

Renumbering Program Lines

After editing a program, you may want to renumber the program lines. The CC-40 automatically renumbers the lines in a program when you use the RENUMBER (or REN) command.

Line Length

A line may be up to 80 characters long, including the line number. Additional characters typed at the end of the line replace the 80th character.

Lines Containing Remarks

You can include explanations and comments in a program by using remarks. A remark is not executed, but it is stored in memory. Enter remarks either by typing **REM** followed by a space and the explanatory remark or by typing an exclamation point and the explanatory remark as shown in lines 100 and 110 below.

```
100 REM THIS LINE IS A REMARK AND IS NOT EXECUTED
110 ! NEITHER IS THIS ONE
```

The exclamation point can also be used as a tail remark symbol by following the statements on a line with the exclamation mark (!) and the explanatory remark, as shown below.

```
120 FOR A=1 TO 25 ! SET UP LOOP
```

Multiple Statement Lines

Each program line may contain more than one statement by separating the statements with colons. For example, the program that calculated the square roots of the first 25 whole numbers could be written on one line as shown below.

```
100 FOR A=1 TO 25:PRINT A;SQR(A):PAUSE 1:NEXT A ! PRINT
    SQUARE ROOTS
```

The line begins with a line number followed by a space. The statement **FOR A=1 TO 25** is followed by a colon to signal the CC-40 that there is another statement, **PRINT A; SQR(A)**. There are four statements in the line. The tail remark symbol (!) tells the CC-40 that the rest of the program line is an explanatory remark which is not to be executed when the program is run.

Program Storage and Execution

You can save a program that you want to keep by using the SAVE command. To execute a program that has been stored, use the RUN statement or the OLD command and RUN.

Saving a Program

The SAVE command is used to copy a program in memory to an external storage device. To store a program on a new tape, you must first format the tape. If you format a tape that already has information on it, all the data is erased. The example below illustrates how to save a program on a new tape.

```
FORMAT 1
SAVE "1.MYPROG"
```

The tape on device 1 is formatted and the program in memory is written to the tape with the filename MYPROG. To save a program on a tape that contains other programs, be sure to give the program in memory a name that does not already exist for a program on the tape.

You can also protect a program when you save it by using PROTECTED in the SAVE command. If the SAVE command includes the option PROTECTED, the saved copy can not be listed, edited, or stored. For example, the following SAVE command places a protected copy of the program in memory on external device 1.

```
SAVE "1.MYPROG",PROTECTED
```

Note: Since a protected program can never be listed, edited, or stored, be sure to save an unprotected copy.

Executing a Stored Program

To execute a program stored on a peripheral device, the program must be loaded into memory by using the OLD command or the RUN statement. The OLD command is used when you want to edit the program or verify that it was loaded into memory correctly. The commands shown below load a program into memory and verify that it was loaded correctly.

```
OLD "1.MYPROG"
VERIFY "1.MYPROG"
```

To execute the program, enter **[RUN]**.

The RUN statement can be used to execute a program stored on a peripheral device. The statement below loads a program into memory from peripheral device 1 and then executes it.

```
RUN "1.MYPROG"
```

Editing Program Lines

After you enter a program, it is often useful to check the program lines for errors by using LIST or the edit keys. Many of the editing features are obtained using the [SHIFT], [CTL], and [FN] keys. By using these keys, you can display lines, delete lines and portions of lines, and move the cursor within a line.

Note: In CC-40 BASIC you cannot delete a line by entering its line number alone. You must use the DELETE keyword described later in this section.

The Right Arrow Key— →

The right arrow key moves the flashing cursor one position to the right. If you press and hold the → key, the cursor continues to move to the right to column 31 and then scrolls the display to the left until column 80 is reached or until the key is released.

The Left Arrow Key— ←

The left arrow key moves the flashing cursor one position to the left. If you press and hold the ← key, the cursor continues to move to the left until it reaches column 1 or until the key is released.

The Up Arrow Key— ↑

The up arrow key is used to display the next lower-numbered program line. If ↑ is pressed with the first line of the program in the display, the CC-40 displays the flashing cursor on a blank line. If you press ↑ again, the highest-numbered program line is displayed. You can also use the ↑ key to display a specific program line by typing the line number and pressing ↑.

The Down Arrow Key— ↓

The down arrow key is used to display the next higher-numbered program line. If ↓ is pressed with the last line of the program in the display, the CC-40 displays the flashing cursor on a blank line. If ↓ is pressed again, the lowest-numbered program line is displayed. You can also use the ↓ key to display a specific program line by typing the line number and pressing ↓.

Character Insert—[SHIFT] [INS]

[SHIFT] [INS] is used to insert characters in a line. The following keys can be used to end an insert.

→	leaves the edited line in the display and moves the cursor one position to the right.
←	leaves the edited line in the display and moves the cursor one position to the left.
↑	enters the edited line. If the line was a program line, the next lower-numbered program line is displayed.
↓	enters the edited line. If the line was a program line, the next higher-numbered program line is displayed.
[ENTER]	enters the edited line. If the line was a program line, the display is cleared. If LIST is in effect, the next higher-numbered program line is displayed.

Character Delete—[SHIFT] [DEL]

[SHIFT] [DEL] is used to remove the character at the position occupied by the flashing cursor. If you press [SHIFT] and then press and hold [DEL], the computer continues to delete characters, one at a time, until [DEL] is released. If you press [SHIFT] [DEL] when you are inserting characters, the insert is ended.

Playback—[SHIFT] [PB]

[SHIFT] [PB] causes the previous display contents to reappear. If you want to enter a line similar to the most recently entered line, press [SHIFT] [PB] and edit the line using the edit keys. The [PB] key can be used to avoid retyping a long line. If you press [SHIFT] [PB] when you are inserting characters, the insert is ended.

Tab—[CTL] →

[CTL] → shifts the display to the next higher-numbered tab position. Tab positions are set at 1, 25, and 50.

Back Tab—[CTL] ←

[CTL] ← shifts the display to the next lower-numbered tab position.

Home—[CTL] ↑

[CTL] ↑ moves the cursor to position 1 of the line.

Erase Field—[CTL] ↓

[CTL] ↓ clears the display from the current cursor position to the end of the line.

Line Delete—DELETE

The DELETE (or DEL) keyword is used to delete a group of program lines. DELETE can be accessed by pressing **[FN] [DEL]** or by typing DELETE or DEL. You can delete a single line or a group(s) of lines by entering DELETE (or DEL) followed by one or more of the line groups shown below.

Line-group	Effect
a single line number	Deletes that line.
line number –	Deletes that line and all following lines.
– line number	Deletes that line and all preceding lines.
line number – line number	Deletes that inclusive range of lines.

If more than one line number group is used, use commas to separate the groups. For example, **DEL 150,320-350,560-** deletes line 150, lines 320 through 350, and lines 560 through the end of the program.

Error Handling

As you begin to write BASIC programs, you may make mistakes as you enter instructions. The computer tells you through error messages what is wrong. Sometimes a line can not be stored in memory because you have made an error in typing it. Sometimes a program may not work the first time you attempt to execute it. By using the error messages that the computer displays, you can determine what corrections to make.

For example, the following program has two errors in it.

```
100 FOR A=1 TO25
110 PRINT A; SQR(A
120 PAUSE 1
130 NEXT A
```

When you enter lines 100, 120, and 130, they are stored in memory. However, line 110 causes an error when you try to enter it. The error indicator in the display is turned on and the error message `Unmatched parenthesis` is displayed. To correct this line, press **[SHIFT] [PB]** to display it, and then add a parenthesis after the last A.

If you try to run the program, the error message `Illegal syntax` is displayed. When an error message is displayed, press → to display the error code and the number of the erroneous line. In this case, the error code that is displayed is E1 and the number of the erroneous line is 100.

Press ↑ or ↓ to display the erroneous line. Between the word TO and the number 25 there must be a space. Use the edit keys to place a space there. You can then run the program.

Refer to appendix K for a list of the error codes and messages. You can handle errors which occur while a program is running by using the error processing statements available in CC-40 BASIC. Refer to Handling Errors in a BASIC Program in this chapter.

Constants and Variables

The data used by BASIC keywords may be either constants or variables. The rules and conventions used are described in the following sections.

Constants

A constant is a value that does not change throughout the entire execution of a program. There are two kinds of constants, numeric and string.

Numeric Constants

A numeric constant is either a positive or negative real number or zero. Positive numbers may optionally be written with a + sign. Negative numbers must be preceded by a minus sign. Commas and spaces are not allowed in numbers.

Constants may be entered with any number of digits, but they are rounded to 13 or 14 digits due to the internal storage method used by the CC-40. Only ten digits of a constant are displayed when a program is running, but all 13 or 14 digits are used in calculations and are displayed when a program is listed.

Numbers are normally stored and displayed in standard notation. Very large or small numbers are stored and displayed in scientific notation, which is described in chapter 2.

For example, if you enter a constant as **3E4**, it is retained in memory and displayed when the program is listed or run as 30000.

The following are examples of valid numeric constants.

5
25.7
3.598E4 (which is retained internally as 35980)
−1900

String Constants

A string constant is a series of characters usually enclosed in quotation marks. The quotation marks may be omitted when a string constant is used in a DATA or IMAGE statement. To include leading and trailing blanks in a string constant, you must use quotation marks. A quotation mark within a quoted string constant is represented by two quotation marks. To include a quotation mark at the beginning or end of a string constant, you use three quotation marks. The CC-40 does not change any lower-case alphabetic characters to upper-case characters in string constants.

The following are examples of valid string constants and the way they would appear if printed.

String Constant Example	Appears In Print
Hello“”Goodbye	Hello""Goodbye
“““”Hello“”Goodbye“”””	"Hello"Goodbye"
Hello Goodbye	Hello Goodbye
“ Hello Goodbye”	Hello Goodbye

Variables

A variable is a name given to a memory location in the CC-40. You can store a value in that location, and later change the value of the variable by storing a different value in the location.

A variable name can consist of up to 15 characters, the first of which must be a letter of the alphabet, an underline (_), or the at sign (@). The remaining 14 characters can be alphanumeric, the underline, or @. A program can include up to 95 variable names. The keywords that are reserved for use by CC-40 BASIC may not be used as variable names, but they may make up part of a variable name. See appendix C for a complete list of the words reserved for CC-40 BASIC.

There are two kinds of variables, numeric and string.

Numeric Variables

A numeric variable is a name given to a location that stores a numeric value. The following are valid numeric variable names.

X, A9, @ALPHA, BASE_PAY, _@TABLE4

String Variables

A string variable is a name given to a location that stores any combination of characters (letters, numbers, and other symbols). The string variable name must end with a $, which is counted as one of the 15 characters allowed. The following are examples of valid string variable names.

N$, YZ2$, NAME@$, Q_5O5$, ADDRESS$

Assigning Values to Variables

Before values are assigned, numeric variables are equal to zero and string variables are assumed to have no characters (or be null). You can set the value of a variable to be either a constant or the result of a calculation by using an assignment statement. You can also set the value of a variable with READ and DATA statements or by various input statements. Assigning values using READ and DATA statements and input statements is discussed later in this chapter.

In the example below, a 5 is put or stored in the location called K when line 210 is executed and the characters File- are stored in the location called K$ when line 220 is executed. When line 230 is executed, the value of K*10 (50) is stored in locations A, B, and C.

```
210 K=5
220 K$= "File-"
230 A,B,C=K*10
```

You can also use the optional keyword LET in an assignment statement. The following statement stores the result of 25.5 times 3 in the location called A when the line is executed.

```
250 LET A=25.5*3
```

Arrays

An array is a group of values given the same variable name. Each value is an element of the array. The elements are in an ordered sequence to provide easy access to any value in the array. When a variable name is chosen for an array, that name must always refer to the array. For example, if A is chosen as the name for an array, then A cannot appear as a simple variable elsewhere in the program.

To tell the computer which element you are using, you need a pointer. The pointer, called a subscript, is a value enclosed in parentheses immediately following the name of the array. In CC-40 BASIC an array begins with element 0.

The DIM Statement

To have the CC-40 reserve space for an array, specify the array in a DIM statement such as 130 DIM C(5) which reserves six locations, C(0) through C(5) for the array C. The CC-40 chooses a memory location, names it C, and reserves enough space for array C. You can use an array without including it in a DIM statement if you do not require more than 11 elements.

A One-Dimensional Array

Suppose the array C has had the following values assigned to the elements.

ARRAY C:	C(0)	C(1)	C(2)	C(3)	C(4)	C(5)
Value:	0	10	25	30	45	90

Then C(2) refers to the third element in array C, which has a value of 25 in this example. If M=4, then C(M) refers to the fifth element of array C, which has a value of 45.

An example of a string array is shown below.

ARRAY NM$:	NM$(0)	NM$(1)	NM$(2)	NM$(3)	NM$(4)
Value:	Bob	Tom	Bud	Nancy	John

A Two-Dimensional Array

You can extend an array to include information from a table which has rows and columns. A two-dimensional array has two subscripts that refer to the element's row and column. Suppose the array B is specified in the statement DIM B(2,2). Then the CC-40 reserves 9 locations for the 9 elements in array B. Suppose these locations have the values as shown below.

ARRAY B

B(0,0)	15	B(0,1)	18	B(0,2)	21
B(1,0)	24	B(1,1)	27	B(1,2)	30
B(2,0)	33	B(2,1)	36	B(2,2)	39

Then you refer to the element of array B that has a value of 30 as B(1,2). Refer to the element of array B that has a value of 18 as B(0,1). You can refer to any element of array B as B(R,C), where R is equal to the row and C is equal to the column of the element.

In CC-40 BASIC you can have an array with up to 3 dimensions.

Using Arrays

Enter the following program in the CC-40. (Remember to type **NEW** and press **[ENTER]** before you start.)

```
100 DIM A(5),B(5)
```

Line 100 reserves six locations for array A and six for array B.

```
110 A(1)=2:A(2)=4:A(3)=6:A(4)=8:A(5)=10 !A(0) is not used
```

Line 110 has several assignment statements, assigning values to A(1) through A(5).

```
120 B(1),B(2),B(3)=2:B(4)=3E-5:B(5)=10 !B(0) is not used
```

Line 120 has several statements to store values in array B.

```
130 FOR C=1 TO 5
```

Line 130 sets up a loop which is executed 5 times.

```
140 PRINT "A*B= ";A(C)*B(C):PAUSE 2.5
```

Line 140 prints the string constant `A*B=` in the display, followed by the product of the array elements that the counter C refers to. The PAUSE statement holds the answer in the display for 2.5 seconds so you can see the answer.

```
150 PRINT "A/B= ";A(C)/B(C):PAUSE 2.5
```

Line 150 prints the string constant A/B= followed by the quotient of the elements of the arrays and then pauses for you to read the answer.

```
160 NEXT C
```

Line 160 is the last line of the loop.

To list the program, type **LIST** and then press **[ENTER]** to see each succeeding line. To execute the program, enter **[RUN]**. The CC-40 displays the following products and quotients: 4 and 1; 8 and 2; 12 and 3; .00024 and 266666.6667; and 100 and 1.

READ and DATA Statements

When you have many values to assign to variables, you can easily assign them using READ and DATA statements. Each time a READ statement is executed, it reads data from a DATA statement. The values are written in a DATA statement(s) which may appear anywhere in your program.

A READ statement can read data into any number of variables. The data in the DATA statement is read from left to right. If necessary, a READ statement reads from more than one DATA statement. More than one READ statement can assign the values in a DATA statement; each READ statement assigns the first unread data value.

The previous example could use the READ and DATA statements instead of arrays and multiple assignment statements, as shown below.

```
100 DATA 2,2,4,2,6,2,8,3E-5,10,10
```

Line 100 lists the first value of A followed by the first value of B and then repeats values of A and B for all five pairs of numbers.

```
110 FOR C=1 TO 5
```

Line 110 sets up a loop to be executed five times.

```
120 READ A,B
```

Line 120 stores the first value from the DATA statement in A and the next value in B. Each time line 120 is executed, the next pair of values in the DATA statement is stored in A and B.

```
130 PRINT "A*B=";A*B:PAUSE 2.5
```

Line 130 prints A*B= followed by the product of A and B and then pauses 2.5 seconds for you to see the answer.

```
140 PRINT "A7B=";A/B:PAUSE 2.5
```

Line 140 prints A/B= followed by the quotient of A and B and pauses 2.5 seconds.

```
150 NEXT C
```

Line 150 ends the loop.

To run this program, enter [RUN] and you will get the same answers as in the previous example.

READ and DATA with the RESTORE Statement

You can also cause a READ statement to assign values from the first DATA statement again or from other DATA statements by using the RESTORE statement.

The following program reads the first five pairs of values from the first DATA statement and reads the next two pairs from the second DATA statement. After the products and quotients have been displayed, the RESTORE statement in line 160 causes the next READ statement executed to start assigning values from the first value in the first DATA statement.

All the values in the two DATA statements are assigned and the sums printed. The RESTORE statement in line 210 causes the next READ statement executed to start assigning values from the first value in the DATA statement in line 105.

```
100 DATA 2,2,4,2,6,2,8,3E-5,10,10
105 DATA 30,40,5,20
110 FOR C=1 TO 7
120 READ A,B
130 PRINT "A*B=";A*B:PAUSE 2.5
140 PRINT "A/B=";A/B:PAUSE 2.5
150 NEXT C
160 RESTORE
170 FOR C=1 TO 7
180 READ A,B
190 PRINT A+B:PAUSE 2.5
200 NEXT C
210 RESTORE 105
220 READ A,B,C,D
230 PRINT A;B;C;D:PAUSE 2.5
```

When you run this program, the following answers are displayed.

```
4
1
8
2
12
3
.00024
266666.6667
100
1
1200
.75
100
.25
4
6
8
8.00003
20
70
25
30  40  5  20
```

Expressions

In BASIC, calculations are performed by writing them as expressions. Expressions are constructed from constants, variables, and functions. There are four types of expressions: numeric, string, relational, and logical.

Numeric Expressions

A numeric expression is a series of one or more constants, variables, and/or functions connected by any of five arithmetic operators: +, −, *, /, ^. An operator must appear between each pair of numeric constants, variables, and/or functions. When a numeric expression is evaluated, the result is always a number. CC-40 BASIC uses standard algebraic hierarchy in evaluating numeric expressions in the order given below. Refer to chapter 2 for a discussion of this order.

1. Calculations within parentheses are evaluated first.
2. Exponentiation is performed next.
3. Negation is performed next.
4. Multiplication/division from left to right is performed.
5. Addition/subtraction from left to right is performed last.

String Expressions

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (&) which combines or links strings. If the string length exceeds 255 characters, characters on the right are lost and the warning `String truncation` is displayed. The following is an example of string concatenation.

```
AB$="THIS IS AN EXAMPLE "
BB$="OF STRING CONCATENATION"
STRING$=AB$ & BB$
PRINT STRING$
```

Enter the above four lines in the CC-40 and it displays

```
THIS IS AN EXAMPLE OF STRING CONCATENATION.
```

Relational Expressions

Relational expressions are constructed from variables, constants, and functions to compare two values, using the following six relational operators.

=	(equal to)
< >	(not equal to)
<	(less than)
< =	(less than or equal to)
>	(greater than)
> =	(greater than or equal to)

The result of the comparison is either true or false. A relational expression has a value of −1 if it is true. A relational expression has a value of 0 if it is false.

Relational expressions are most often used in the IF THEN ELSE statement (described later in this chapter), but may be used anywhere that a numeric expression is allowed. The values compared must be either both numeric or both string.

Numeric Comparisons

Relational expressions are evaluated from left to right after all arithmetic operations within the expression are completed. The following examples illustrate the use of relational expressions to compare numeric values.

```
150 IF X<Y THEN 200
```

If X is less than Y, control transfers to statement 200. If X is greater than or equal to Y, control continues with the statement after line 150.

```
200 A=2<5
210 PRINT A:PAUSE
```

Sets A equal to −1 since it is true that 2 is less than 5 and prints the value of A.

```
100 PRINT 2>5:PAUSE
```

Prints 0 since it is false that 2 is greater than 5.

String Comparisons

Comparisons of string values are performed by taking one character at a time from each string and comparing their ASCII codes. Leading and trailing blanks are significant. (See appendix D for a complete list of ASCII codes.) If the ASCII codes differ, the string with the lower code is less than the string with the higher code. If all the ASCII codes are the same, the strings are equal. For strings of unequal length, the comparison is performed for as many characters as there are in the shorter string. If all the ASCII codes are the same, the longer string is considered greater. The null string (“”) is less than every other string.

```
100 PRINT “THIS” = “THAT”:PAUSE
```

Prints 0 since it is not true that “THIS” is equal to "THAT”.

```
110 PRINT “ABC”<“ABCD”:PAUSE
```

Prints -1 since it is true that “ABC” is less than “ABCD”.

Logical Expressions

The logical operators (AND, OR, NOT, and XOR) are generally used with relational expressions. The logical operators can also be used to manipulate data on a bit basis. Refer to appendix H for a description of using the logical operators in this way.

The order of precedence for logical operators, from highest to lowest, is NOT, XOR, AND, and OR. The following examples illustrate the use of logical operators with relational expressions to form logical expressions. These logical expressions have a value of either true or false.

A logical expression with AND is true if the conditions on both its left and right sides are true.

100 IF 3<4 AND 5<6 THEN L=7

Sets L equal to 7 since 3 is less than 4 and 5 is less than 6.

110 IF 3<4 AND 5>6 THEN L=7

Does not set L equal to 7 because 3 is less than 4, but 5 is not greater than 6.

A logical expression with OR is true if either the condition on its left side is true, the condition on its right side is true, or both the conditions are true.

120 IF 3<4 OR 5>6 THEN L=7

Sets L equal to 7 because 3 is less than 4.

A logical expression with XOR (exclusive or) is true if either the condition on its left side is true, the condition on its right side is true, but not if both the conditions are true.

130 IF 3<4 XOR 5>6 THEN L=7

Sets L equal to 7 because 3 is less than 4 and 5 is not greater than 6.

140 IF 3<4 XOR 5<6 THEN L=7

Does not set L equal to 7 because 3 is less than 4 and 5 is less than 6.

A logical expression with NOT is true if the condition following it is not true.

150 IF NOT 3=4 THEN L=7

Sets L equal to 7 because 3 is not equal to 4.

Note: NOT 3=4 is equivalent to 3< >4.

160 IF NOT 3=4 AND (NOT 6=5 XOR 2=2) THEN 200

Does not pass control to line 200 because while it is true that 3 is not equal to 4, it is true that both 6 is not equal to 5 and 2 is equal to 2, so the condition in parentheses is not true.

Order of Execution of Expressions

The order of operations within arithmetic, relational, and logical expressions was given in the discussion for each type of expression. The order of precedence for evaluating expressions is given below.

- Functions are evaluated first.
- Arithmetic operations are performed next.
- String operations are performed next.
- Relational operations are performed next.
- Logical operations are performed last.

Input/Output Statements

Before data can be processed, it must be transferred into the computer. Data can be transferred into the computer by using assignment statements, READ and DATA statements, the KEY$ function, or some form of input statement. In CC-40 BASIC the input statements are INPUT, LINPUT, and ACCEPT.

After the data has been processed, you either want to view it or store it for future use. To display or store the processed data, use some form of output statement. The output statements are PRINT and DISPLAY.

Using input and output statements with the display is described below. Using input and output statements with external devices such as printers and tape drives is described in this chapter under "Using External Devices".

The PAUSE Statement

The CC-40 displays printed items so quickly that you can not see them. There are three ways to have items remain in the display long enough so that you can read them.

First, you may have the computer pause after each statement that displays items by putting the statement PAUSE ALL in the program before any output statement. During a pause, the underline cursor is displayed in column 1 waiting for you to acknowledge the pause by pressing **[ENTER]** or **[CLR]**. After either of the keys is pressed, the computer resumes execution of the program with the next statement.

Second, you may have the computer pause after a specific statement by following that statement with PAUSE and the number of seconds that the pause is to last. In this case the cursor is not displayed. After the number of seconds specified has passed, the program resumes execution. If you use PAUSE with no time parameter, the cursor is displayed in column 1 and the program resumes execution after **[ENTER]** or **[CLR]** is pressed.

Finally, to keep a prompt for an ACCEPT statement in the display, you can follow the PRINT or DISPLAY statement with a comma or a semicolon to create a pending print (described later under "Pending PRINT and DISPLAY Statements").

Input Statements

The input statements allow a program to get data from the keyboard. The INPUT, LINPUT, and ACCEPT statements store the value(s) entered from the keyboard into the variable(s) listed in the statement. Only one value can be entered at a time. The KEY$ function is used to halt program execution until a key is pressed.

Each statement is discussed briefly in the section below. Refer to chapter 5 for a detailed explanation of each input statement.

The INPUT Statement

You can put values typed on the keyboard into variables by using the INPUT statement. For example, enter the following program in the CC-40.

```
100 INPUT K
110 INPUT "ENTER DEGREES:  ";D
120 INPUT A, B$
130 PRINT K;D;A;B$:PAUSE
```

When the program is run, the INPUT statement in line 100 halts program execution, displays a ? in column 1, and waits for a value to be entered from the keyboard. When a value is entered, it is stored in variable K.

Line 110 displays `ENTER DEGREES:` and waits for you to enter a value to be stored in D.

Line 120 displays a ? in column 1 and waits for you to enter a value for A. When a value is entered, it is stored in A. A question mark is displayed again to prompt you for a value for B$.

The LINPUT Statement

The LINPUT statement assigns any series of characters entered from the keyboard to a string variable. Therefore, you can enter commas and leading and trailing spaces which are not allowed in the INPUT statement unless they are enclosed in quotes.

```
120 LINPUT "NAME: ";NEM$
```

displays `NAME:` and waits for a value to be entered that will be stored in NEM$.

The ACCEPT Statement

The ACCEPT statement gives more control over data that is input from the keyboard. The options available in ACCEPT allow you to sound a tone, erase all or part of the display, limit the number and type of characters, and specify the column where they can be entered. For example, when the following line is executed, the computer beeps, erases the display, positions the cursor at column 10, and waits for you to enter a value with up to 4 characters for DEG. As you type the value, each character is tested to see if it is numeric (0-9, +, −, ., E) before it can be entered from the keyboard.

```
100 ACCEPT AT(10) VALIDATE(NUMERIC) BEEP ERASE ALL SIZE(4),
    DEG
```

The KEY$ Function

The KEY$ function allows you to halt program execution until a key is pressed. KEY$ returns a one character string that corresponds to the key that was pressed. For example, when the following statement is executed, program execution halts until a key is pressed. The character corresponding to the key that was pressed is then stored in K$. Refer to appendix D for a list of keycodes.

```
150 K$=KEY$
```

Output Statements

The output statements allow you to write data in many different ways and on different media. Each statement is discussed briefly in the section below. Refer to chapter 5 for a detailed explanation of each output statement.

The PRINT and DISPLAY Statements

The PRINT statement allows you to print numbers and strings in the display. Negative values are preceded by a minus sign and nonnegative numbers are preceded by a space (instead of a plus sign). Numeric values are followed by a space.

When several values are to be displayed on a line, they are separated in the output statement with a semicolon or a comma. The semicolon causes the next value to be printed immediately after the preceding one. The comma causes the next value to be printed in the next field. The display is divided into fields 15 characters long. The fields start at columns 1, 16, 31, 46, 61, and 76.

The following statements print the output shown if X equals −7 and Y equals 13.

Statement	Output
100 PRINT X; Y:PAUSE	-7 13
110 PRINT X, Y:PAUSE	-7 13

You can also display string constants in an output statement. Unlike numeric values, string values have no leading signs and no trailing spaces. For example, the following statements print the output shown.

Statement	Output
180 X$="X IS ":X=10:Y$=" Y IS ": Y=20	
190 PRINT X$;X;Y$;Y:PAUSE	X IS 10 Y IS 20

The DISPLAY statement gives you more control than the PRINT statement over data that is displayed. The options available in DISPLAY allow you to sound a tone, erase the display, specify the size of items displayed, and specify the columns where values are displayed. For example, when line 120 is executed, the computer beeps, erases the display, and displays the value of A$ at column 3 followed by the value of B as shown below.

```
Statements                          Output
110 A$="The answer is ":B=15.55
120 DISPLAY AT(3) BEEP ERASE
    ALL,A$;B                        The answer is   15.55
130 PAUSE
```

USING with the PRINT and DISPLAY Statements

The PRINT and DISPLAY statements may optionally include a USING clause that allows you to display the numbers with a specific format. Pound signs (#) are used to show how many digits to use in printing the number. The format can be specified in the PRINT or DISPLAY statement itself or can be written in an IMAGE statement. For example, the following program uses the USING option in a PRINT statement.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE-SMILE)/GALL
140 PRINT USING "Miles per gallon=###.##";MPG
150 PAUSE
```

If you run this program and enter values of **5405.7**, **5807.9**, and **18.3**, then `Miles per gallon =   21.98` is displayed. Without the USING clause, the MPG would have been displayed as `21.97814208`.

You could use the IMAGE statement by adding line 132 and changing line 140 as shown below.

```
132 IMAGE Miles per gallon=###.##
140 PRINT USING 132;MPG
```

TAB with the PRINT and DISPLAY Statements

The TAB function is used with the PRINT and DISPLAY statements to format data, much as the TAB key on a typewriter does. TAB displays enough spaces to make the next value printed appear in a specific column.

The last example can be changed to use the TAB function to display values starting in a specific column. In the following program, lines 134 and 136 have been added to use the TAB function.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE-SMILE)/GALL
132 IMAGE Miles per gallon=###.##
134 PRINT "Miles traveled: ";TAB(20);EMILE-SMILE:PAUSE 2.5
136 PRINT "Gallons used: ";TAB(20);GALL:PAUSE 2.5
140 PRINT USING 132;MPG
150 PAUSE
```

If you enter the same values as before, **5405.7**, **5807.9**, and **18.3**, the display shows

```
Miles traveled:     402.2 (for 2.5 seconds)
Gallons used:        18.3 (for 2.5 seconds)
Miles per gallon =  21.98
```

Pending PRINT and DISPLAY Statements

The PRINT statements used in the examples have displayed exactly one line. Sometimes you may want to have several PRINT or DISPLAY statements display information on the same line. A pending print is created when a PRINT or a DISPLAY statement ends with a comma or semicolon. If a comma ends the statement, the computer spaces over to the next field; if a semicolon ends the statement, the computer does not space over. Then the next PRINT or DISPLAY statement prints on the same line at the current column position.

The program above can be changed to print the mileage and the gallons on the same line by changing lines 134 and 136 as shown below.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE-SMILE)/GALL
132 IMAGE Miles per gallon=###.##
134 PRINT "Miles=";EMILE-SMILE;
136 PRINT "Gallons=";GALL:PAUSE 2.5
140 PRINT USING 132;MPG
150 PAUSE
```

Control Statements

Most of the programs you have run on the CC-40 started executing the first statement and continued executing each sequential line to the last. The flow of the program or flow of control has gone from the first statement to the last.

Control statements are used to direct the flow of the program. Some statements form a loop and cause some lines to be repeated a specified number of times. You have already used two of these statements, the FOR TO STEP and NEXT statements. Some statements compare data and cause program execution to jump or branch to another line rather than go to the next program statement. This section describes the various control statements available in CC-40 BASIC.

The FOR TO STEP Statement

You have already used the FOR TO and NEXT statements in a program to create a loop. The FOR TO statement has another option that allows you to increment the counter other than by 1. For example, entering

```
100 FOR COUNT=2 TO 100 STEP 2
```

starts a loop where the counter begins at 2 and is incremented by 2 each time. The loop is repeated until the counter is greater than 100.

If the starting value of the counter is greater than the limit value, the loop is not executed. If the starting value and the limit value of the counter are the same, the loop is executed one time.

You can also use a negative value for STEP. The counter of the FOR statement is decreased each time the loop is executed. If the starting value of the counter is less than the limit value, the loop is not executed. If the starting value and the limit value of the counter are the same, the loop is executed one time.

Enter the following program. The CC-40 displays the steps it calculates in the loop.

```
100 FOR A=6 TO 4 STEP -.25
110 DISPLAY AT(10) BEEP,"A=  ";A:PAUSE 2.1
120 NEXT A
```

You should not transfer control into the middle of a loop from the outside. The counter or control variable is set up only when the FOR TO statement is executed. You may transfer control out of a loop with a GOTO, GOSUB, ON GOTO, or ON GOSUB statement and then transfer back in.

Nested Loops

A FOR TO NEXT loop can be contained within another loop. The loop that is inside is called a nested loop. A nested loop must always be entirely inside the outer loop. For example, in the program above, the value of the counter can be displayed in successive columns by adding statements 105 and 130 and by changing statements 110 and 120 as shown below.

```
100 FOR A=6 TO 4 STEP -.25
105 FOR B=1 TO 7 STEP 3
110 DISPLAY AT(B) BEEP,"A= ";A:PAUSE 2.1
120 NEXT B
130 NEXT A
```

The GOTO Statement

The GOTO statement tells the computer what line in a program to execute next. The following program uses a GOTO statement to read all the data in the DATA statement.

```
100 DATA 5,10,3.5,420,55.25
110 READ R
120 PRINT R, 2*PI*R:PAUSE 2
130 GOTO 110
```

Each time line 130 is executed, control is transferred back to line 110 which is executed again. When line 110 tries to read past the data in the DATA statement, an error occurs and the message DATA error is displayed. To determine when to stop reading data, a dummy value (a value you know marks the end of the data) can be inserted in the DATA statement and the IF THEN ELSE statement used to test it.

The IF THEN ELSE Statement

The IF THEN ELSE statement allows you to compare data in a program. The data compared can be constants, variables, and/or expressions. If the comparison or condition being tested is true, the statement(s) following the word THEN are executed. If the comparison is false, the statement(s) following the word ELSE are executed. If the comparison is false and there is no ELSE, the line following the IF statement is executed.

In the following program a check is made on the data read. If the dummy value (a value less than zero) has been read, an end of data message is printed. If the dummy value has not been read, the result of the calculation is printed. After **[ENTER]** is pressed, the next value in the DATA statement is read.

```
100 PAUSE ALL
110 DATA 5,10,3.5,420,55.25,-5
120 READ R
130 IF R<0 THEN 160
140 PRINT R, 2*PI*R
150 GOTO 120
160 PRINT "END OF DATA"
```

The following are examples of IF THEN ELSE statements.

400 IF D=999 THEN DISPLAY "ARE YOU FINISHED?" ELSE 150

The computer checks the value in location D to determine if it is 999. If D is 999, the computer displays `ARE YOU FINISHED?` and executes the next line. If D is not 999, the computer executes line 150.

510 IF L(C)<>12 THEN C=S+1 ELSE COUNT=COUNT+1:GOTO 140

The computer checks the value in L(C) and if it is not equal to 12, then C is set equal to S+1 and the next line is executed. If L(C) is equal to 12, then COUNT is set equal to COUNT plus 1 and line 140 is then executed.

The ON GOTO Statement

Another control statement is ON GOTO. The ON GOTO statement is used to transfer control to a program line based on whether the value of the variable following the word ON is 1, 2, 3, etc.

```
100 REM THIS PROGRAM IS A DEMONSTRATION
110 ! OF THE ON GOTO STATEMENT
120 PRINT “1 for LOG, 2 for LN, 3 for EXP”;
130 ACCEPT AT(31) SIZE(1) BEEP VALIDATE(“123”),CODE
140 DISPLAY ERASE ALL, “ENTER ARGUMENT:”;
150 ACCEPT BEEP, ARG
160 IF ARG<0 THEN 140
170 ON CODE GOTO 180, 200, 220
180 PRINT “LOG of ”;ARG;“is ”;LOG(ARG):PAUSE
190 GOTO 120
200 PRINT “LN of ”;ARG;“is ”;LN(ARG):PAUSE
210 GOTO 120
220 PRINT “EXP of ”;ARG;“is ”;EXP(ARG):PAUSE
230 GOTO 120
```

The program above accepts a 1, 2, or 3 for the variable CODE. The PRINT statement displays a prompt and the ACCEPT statement halts program execution until a value is entered for ARG. If the value of ARG is negative, the prompt is again displayed and the ACCEPT statement waits for another value for ARG. When a nonnegative value is entered for ARG, the program calculates the LOG, LN, or EXP of ARG depending upon the value entered for CODE.

Strings and String Manipulation

String constants and string variables have already been defined in this chapter. However, you may find that you need to be able to manipulate a string. This section describes strings and the functions you can use on the CC-40 to manipulate them.

Each character is stored in the CC-40 as a number from 0 through 255. The number is called the ASCII character code. For example, the string values “BASIC” and “Basic” are represented as shown below. The string BASIC is less than the string Basic because the ASCII code for A is less than the ASCII code for a.

B	A	S	I	C		B	a	s	i	c
66	65	83	73	67		66	97	115	105	99

Converting a Character to ASCII Code—ASC

You can convert the first character in a string to its ASCII character code by using ASC.

```
100 NUMB1=ASC("H")
110 NUMB2=ASC("hello")
120 NUMB3=ASC("%")
```

Assign NUMB1 the value 72, NUMB2 the value 104, and NUMB3 the value 37.

Converting a Number to its Corresponding Character—CHR$

You can convert a number (from 0 through 255) to the character that is designated by the number according to ASCII conventions.

```
100 A$=CHR$(42)
```

Assigns the character * to A$.

The following program accepts a value from the keyboard. If the value is a lower-case character, it is changed to upper-case. The value is then printed and control returns to statement 100. To terminate the program, press **[BREAK]**.

```
100 PRINT "Press a key: "
110 A$=KEY$
120 IF ASC(A$)<97 OR ASC(A$)>122 THEN 140
130 A$=CHR$(ASC(A$)-32)
140 PRINT "The character is now ";A$:PAUSE 2: GOTO 100
```

Finding the Length of a String—LEN

You can determine the length of a string by using the LEN function.

```
100 ALB$="LIST 10"
110 N=LEN(ALB$)
```

Define the string ALB$ and assign the number of characters in ALB$ to the variable N. In this case N has a value of 7.

Repeating a String—RPT$

You can repeat a string by using the RPT$ function.

```
130 PRINT RPT$("BASIC ",5):PAUSE
```

Displays the string BASIC BASIC BASIC BASIC BASIC .

Finding a String within a String—POS

You can determine the position of one string within another string by using the POS function.

```
140 BB=POS(ALB$,"ST",1)
```

Assigns the variable BB the position in string ALB$ where the string "ST" first occurred. From the example above, ALB$ is equal to "LIST 10" and the position where the string "ST" first occurs in ALB$ is 3.

Getting a Substring of a String—SEG$

You can get a substring of a string by using the SEG$ function.

```
190 CC$=SEG$(ALB$,4,3)
```

Assigns the variable CC$ three characters from the string ALB$ starting at the fourth character. If ALB$ is equal to "LIST 10", CC$ is set equal to T 1.

Converting a Number to a String—STR$

You can convert a number to a string by using the STR$ function.

```
150 BB=17
160 VALUE$=STR$(BB)
```

Converts the number in BB to a string that represents that number and assigns it to VALUE$. In this case VALUE$ contains the string "17".

Converting a String to a Number—VAL

You can convert a string to a number by using the VAL function.

```
170 NUMBRE=VAL(VALUE$)
```

Converts the string representing a number in VALUE$ (which is "17") to the number and assigns it to NUMBRE. In this case, NUMBRE has a value of 17.

Testing a String for a Numeric Constant—NUMERIC

You can test a string to determine if it is a valid representation of a numeric constant by using the NUMERIC function. NUMERIC returns a value of −1 (true) if the string is a valid representation of a numeric constant and a value of 0 (false) if it is not. NUMERIC can be used on a string to see if VAL will convert it to a numeric value.

The statements below are used to test if A$ is a valid representation of a numeric string before the VAL function is used to change the string to a number.

```
160 IF NUMERIC(A$) THEN A=VAL(A$) ELSE PRINT "NOT A
    NUMBER":PAUSE
```

Built-in BASIC Functions

All of the CC-40 BASIC functions may be used in a program line. The trigonometric and logarithmic/exponential functions have been discussed in chapter 2. Some CC-40 functions are described in this chapter in sections that deal with similar instructions. The functions available on the CC-40 to manipulate numbers and generate random numbers are described below.

Manipulating Numbers

The absolute value of an expression can be obtained by using ABS. In the example below, K is set equal to 20. ABS always returns a positive value or zero.

```
100 K=ABS(-4*5)
```

The sign of a number can be determined by using SGN. In the example below, K is set equal to 1 if C is positive, 0 if C is zero, and −1 if C is negative.

```
110 K=SGN(C)
```

The INT function is used to find the largest integer that is less than or equal to a number. In the example below, K is set equal to 23 and L is set equal to −5.

```
120 K=INT(23.99999)
130 L=INT(-4.1)
```

Generating Random Numbers

You can have the CC-40 generate random numbers for programs involving statistical analysis, games, and simulations. The CC-40 produces random numbers from 0 to 1 when RND is used. For example, the program below generates 10 random numbers.

```
100 FOR J=1 TO 10
110 PRINT RND:PAUSE 1.5
120 NEXT J
```

The same series of random numbers is generated each time you run a program unless a RANDOMIZE statement is executed before generating the random numbers.

The program below prompts for the number of random numbers to be generated. The random numbers are printed one at a time followed by their average. Press **[BREAK]** to stop the program. **Note**: The average of many random numbers is approximately .5.

```
100 INPUT "ENTER NUMBER OF VALUES: ";QUAN
110 AVER=0
120 FOR A=1 TO QUAN
130 D=RND:PRINT D:PAUSE 2
140 AVER=AVER+D
150 NEXT A
160 PRINT "Average is ";AVER/QUAN:PAUSE
170 GOTO 100
```

To generate a sequence of integer random numbers, you can use INTRND. INTRND generates a random number between 1 and the number that you give it. For example, the program below generates ten random numbers between 1 and 100.

```
100 FOR J=1 TO 10
110 PRINT INTRND(100):PAUSE 1.5
120 NEXT J
```

Subroutines

Many times in a program you may find that you need to use the same group of lines in several places. By writing these lines as a subroutine, you can eliminate the need to duplicate them. Then when you need to execute these lines, you transfer control to the subroutine. When the subroutine has finished executing, control is transferred back.

The GOSUB Statement

To transfer control to a subroutine, you can use the GOSUB statement followed by a line number. The line number is the first program line in the subroutine. The last line of a subroutine must be a RETURN statement that transfers control back to the statement after the GOSUB statement. A subroutine may also transfer control to another subroutine, allowing nesting of subroutines. Refer to chapter 5 for a description of GOSUB.

A subroutine can use and change the values of any variables in the main program which includes it.

When the GOSUB statement is executed, the following process takes place.

1. A pointer to the statement after the GOSUB statement is stored by the computer.
2. Program control transfers to the line specified by the GOSUB statement.
3. The statements of the subroutine are executed.
4. Program control transfers to the address contained in the pointer when the RETURN statement is encountered.
5. Execution resumes with the statement following the GOSUB statement.

The ON GOSUB Statement

The ON GOSUB statement is another way to call a subroutine. ON GOSUB determines which subroutine to call according to the value of the variable following the word ON.

100 ON XVAL GOSUB 200, 400, 600

Transfers control to the subroutine at line 200 if XVAL is 1, to line 400 if XVAL is 2, and to line 600 if XVAL is 3.

XVAL is rounded to the nearest integer.

Subprograms

Like subroutines, subprograms eliminate the need to write duplicate program lines. However, subprograms operate very differently from subroutines. Subprograms are executed by using the CALL statement followed by the subprogram's name and, optionally, a list of arguments enclosed in parentheses. When a program includes subprograms, they must follow the main program.

CC-40 BASIC Subprograms

The first statement in a subprogram must be the SUB statement followed by an optional list of parameters. If a subprogram needs data from the main program, the data must be passed through the parameters. The variables in a main program are restricted for use by the main program and any subroutines in the main program. The variables in a subprogram are restricted for use in the subprogram and any subroutines within the subprogram. Therefore, variable names may be duplicated in a main program and a subprogram. A subprogram may call other subprograms, but must not call itself, either directly or indirectly.

The last statement in a subprogram must be a SUBEND. When the SUBEND statement is executed, control returns to the statement following the CALL statement that called the subprogram. Control may also be returned by the SUBEXIT statement before the end of the subprogram.

Argument List

Data is passed to the subprogram through the argument list of the CALL statement. Each argument in the argument list has a corresponding variable in the parameter list of the subprogram. The arguments of the argument list can be constants, variables, arrays, or expressions. Arguments can be passed by reference or by value.

Passing Arguments By Reference

Arguments that are passed by reference can be variables or arrays. Arrays are always passed by reference. When arguments are passed by reference, the subprogram uses those variables or arrays from the calling program. If a parameter in a subprogram is changed, its corresponding argument in the calling program is also changed.

In the program segment below, the subprogram uses the variables A, B, and L(3) and can change the stored values of these variables. When the SUBEND statement at line 990 is executed, program control returns to line 210 in the main program.

```
200 CALL MARINE(A,B,L(3))
210 ...
:
900 SUB MARINE(F,L,Z)
:
990 SUBEND
```

Passing Arguments By Value

Arguments that are passed by value can be variables, constants, or expressions. To pass a variable by value, the variable must be enclosed in a set of parentheses. Constants and expressions are always passed by value. When arguments are passed by value, the subprogram can use the values of the arguments, but the subprogram cannot alter the values of the arguments.

In the program segment below, the values of the variables A and L(3) and the expression L(3)+1 are passed by value to the subprogram TRACTION, along with the constant 17. TRACTION cannot alter the values of the variables A and L(3) in the calling program or subprogram, but it can alter the value of the variable B. When the SUBEND statement at line 990 is executed, program control returns to line 210 in the calling program or subprogram.

```
200 CALL TRACTION((A),B,(L(3)),17,L(3)+1)
210 ...
:
900 SUB TRACTION(N,L,Z,D,P)
:
990 SUBEND
```

The ATTACH and RELEASE Statements

You can reduce the execution time of a program that repeatedly calls a subprogram by using the ATTACH statement when you have sufficient free memory. When a subprogram is attached, the variables are initialized when the ATTACH is executed and not each time the subprogram is called. The values of the variables are maintained when the subprogram terminates.

To release the memory that is used when a subprogram is attached, use the RELEASE statement. The variables in the subprogram are then initialized each time the subprogram is called, and are not maintained when the subprogram terminates.

Built-in Subprograms

The CC-40 has many built-in subprograms that you can access. The following sections describe these subprograms.

Expanding Memory

You can add to the internal memory of the CC-40 by using CALL ADDMEM. CALL ADDMEM appends the Random Access Memory (RAM) in the *Memory Expansion* cartridge to resident memory. See chapter 3 and appendix J for a description of memory expansion.

Using Memory

The CC-40 provides the capability of using assembly language programs and subprograms. The function FRE can be used to determine the amount of memory available and the following BASIC subprograms can be used to access it: GETMEM, POKE, PEEK, EXEC, LOAD, IO, and RELMEM. Refer to chapter 5 for more information.

The FRE Function

The FRE function is used to determine how much memory is being used for the operating system and the program in memory and how much memory is available.

The GETMEM Subprogram

The GETMEM subprogram is used to reserve the memory that you have determined is available from the FRE function. You can store data and assembly language programs and subprograms there. The amount of memory reserved should be significantly less than the largest block available. Sufficient memory space must remain available for statements that require additional temporary memory. GETMEM requires four bytes of memory for its own operation.

The POKE Subprogram

The POKE subprogram is used to write data or an assembly language program or subprogram in reserved memory. If you use POKE indiscriminately, you may erase programs and/or files. You cannot do any physical harm to the CC-40 with POKE, but you may have to reset the system.

The PEEK Subprogram

The PEEK subprogram is used to read the data in memory locations.

The LOAD Subprogram

The LOAD subprogram is used to load into computer memory an assembly language subprogram from external storage. More than one subprogram may be loaded into memory and if space permits, they may reside in memory with a BASIC program.

The EXEC Subprogram

The EXEC subprogram is used to execute an assembly language program or subprogram.

The IO Subprogram

The IO subprogram is used to perform control operations on peripheral devices.

The RELMEM Subprogram

The RELMEM subprogram is used to release the memory you reserved with GETMEM.

Language Prompting

You can use SETLANG to display system messages in either English or German. Many of the *Solid State Software*™ cartridges provide messages in languages in addition to English. By using GETLANG, you can determine which language is currently in use. The statement below sets the language code to German.

```
CALL SETLANG(1)
```

To reset the language code to English, enter **CALL SETLANG(0)**.

Display Assignments

With CALL CHAR you can define your own displayable characters. With CALL INDIC you can turn the display indicators on and off.

The CHAR Subprogram

CALL CHAR can define up to 7 displayable characters at one time. In the example below, character codes 1 and 2 are defined to be up and down arrows by using CALL CHAR. For a description of CALL CHAR, refer to chapter 5.

```
100 CALL CHAR(1,“040404041F1F0E04”)
110 CALL CHAR(2,“040E1F1F04040404”)
```

The INDIC Subprogram

There are 17 indicators in the display that you can turn on and off. The six indicators at the bottom of the display are reserved for your use. If the other indicators are turned on and off, undesirable results may occur. For more information on CALL INDIC, refer to chapter 5.

To turn on indicator one, the following statement can be used.

```
130 CALL INDIC(1)
```

The program below uses both CALL CHAR and CALL INDIC. CALL CHAR is used to define up and down arrows for character codes 1 and 2. The user is allowed six chances to guess the number the computer has stored. Each time a wrong guess is made, one of the six indicators in the display is turned on by CALL INDIC. When the number is guessed or when the six chances have been used, a message is printed.

```
100 CALL CHAR(1,“040404041F1F0E04”)
110 CALL CHAR(2,“040E1F1F04040404”)
120 DISPLAY BEEP,“GUESS A NUMBER BETWEEN 1 AND
    25”:PAUSE 1.5
130 DISPLAY BEEP,“YOU HAVE 6 CHANCES”:PAUSE 1
140 DISPLAY BEEP,“INDICATORS RECORD YOUR
    GUESSES”:PAUSE 1
150 COUNT=1
160 RANDOMIZE
170 SNUM=INTRND(25)
180 PRINT “ENTER YOUR GUESS ”;
190 ACCEPT AT(28)VALIDATE(DIGIT),GUESS
200 IF GUESS=SNUM THEN 280
210 CALL INDIC(COUNT)
220 IF COUNT=6 THEN 360
230 IF GUESS<SNUM THEN 260
240 PRINT CHR$(1);GUESS;“Try a smaller number”;
250 COUNT=COUNT+1:GOTO 190
260 PRINT CHR$(2);GUESS;“Try a larger number”;
270 COUNT=COUNT+1:GOTO 190
280 PRINT “WOW!!***”;GUESS;“*** is correct”:PAUSE 2
290 FOR A=1 TO 0 STEP-1:X=1
300 FOR B=1 TO 6
310 CALL INDIC(B,A):PAUSE .1
320 DISPLAY AT(X)BEEP,“YEA!”:X=X+5
330 NEXT B
340 NEXT A
350 GOTO 420
360 PRINT “6 Chances!”;SNUM;“was the number”:PAUSE 4
370 FOR A=1 TO 30 STEP 10
380-DISPLAY AT(A) BEEP,“BOO!”:PAUSE .2:NEXT A
390 FOR J=1 TO COUNT
400 CALL INDIC(J,0)
410 NEXT J
420 PRINT “Press ENTER to play again”:PAUSE
430 GOTO 150
```

The KEY Subprogram

The KEY subprogram is used to determine which key, if any, is pressed. For example, CALL KEY(K,S) assigns to K the ASCII code of the current key that is pressed. S is set equal to 1 if the key pressed is different from the one the last time the KEY subprogram was called. S is set equal to −1 if the same key is pressed that the last call to KEY returned, and to 0 if no key is pressed.

For example, the following section of a program waits for a key to be pressed and then checks if the key was **Y** or **y**.

```
:
150 CALL KEY(K,S)
160 IF S=0 THEN 150
170 IF K=ASC("y") or K=ASC("Y") THEN 250
:
```

The VERSION Subprogram

The VERSION subprogram is used to determine the version of BASIC that is being used. CALL VERSION(V) sets V equal to 10, the BASIC used on the CC-40.

The CLEANUP Subprogram

You can eliminate any variables that are not being used in the current program in memory by calling the subprogram CLEANUP. CLEANUP cannot be called from a program.

The DEBUG Subprogram

The DEBUG subprogram is used to access the debug monitor to allow you to read and change memory locations and run and debug your assembly language programs and subprograms. Refer to appendix I for a description of DEBUG.

Handling Errors in a BASIC Program

The CC-40 provides a means of processing errors in a BASIC program by using the ON ERROR statement and the ERR subprogram. When a program is executed, the error handler is automatically set to display a message and stop program execution when an error occurs. However, you can modify the error handler to cause it to execute a subroutine when an error occurs by using ON ERROR followed by a line number.

The line number must be the beginning of a subroutine. In the subroutine, you can call the ERR subprogram to obtain the error code of the error that occurred. You can then compare error codes in the subroutine and determine what caused the error. The subroutine must end with a RETURN statement. Refer to appendix K for a complete list of the error codes.

For example, in the following program, the ON ERROR statement causes any errors that occur to be handled by the subroutine starting at line 300. The program accepts the name of the next program to run. The computer searches for the program. If the program is not found, an error occurs. The subroutine prints a prompt for another program name to be entered. The program continues execution when the program to be executed is found. If the error occurred for any other reason, the error code and the line number are printed and the program stops.

```
190 ON ERROR 300
200 INPUT "ENTER PROGRAM NAME ";PROG$
210 RUN PROG$
300 REM ERROR HANDLING SUBROUTINE
310 CALL ERR(CODE,TYPE,FILE,LINE)
320 IF LINE<>210 THEN RETURN 360
330 IF CODE<>15 THEN RETURN 360
340 PRINT "Prog. not found, press CLR":PAUSE
350 RETURN 190
360 REM PRINT ERROR SUBROUTINE
370 PRINT "ERROR";CODE;" IN LINE";LINE:PAUSE
```

Handling Breaks in a BASIC Program

The CC-40 provides a means of processing breakpoints that occur in a BASIC program. When a program is executed, the computer automatically halts program execution and displays a message when a breakpoint occurs. However, by using ON BREAK, you can cause breakpoints to be ignored (including the **[BREAK]** key) or to be treated as errors. If breakpoints are treated as errors, the ON ERROR statement can process them as described above. See ON BREAK in chapter 5 for more information.

Handling Warnings in a BASIC Program

The CC-40 provides a way to handle warnings that occur in a BASIC program. When a warning occurs while a program is executing, the computer automatically displays a warning message and then continues program execution when **[CLR]** or **[ENTER]** is pressed . However, by using ON WARNING, you can cause a warning message not to be displayed or a warning to be treated as an error.

Debugging

You may find that a program does not work the way you intended. The errors that are in it are logical errors, called "bugs" in computer usage. Testing a program to find these bugs is called "debugging" a program.

Finding Bugs

Remember that a program is doing exactly what it was told to do. When the program is not working properly, think about what could be going wrong, then devise tests to perform within the program to aid you in finding the bugs.

Debugging Aids

When you are debugging a program, you can use the following aids to help track down the error.

The **[BREAK]** key stops program execution when the key is pressed. At this point you can clear the break message and print or change the values of variables.

The BREAK statement allows you to stop a program at specific lines to determine what is happening in the program. You can print or change the values of the variables.

130 BREAK

Stops the program when the BREAK statement is executed.

230 BREAK 240,250

Sets breakpoints immediately before lines 240 and 250.

BREAK 300,350,380

Can be entered for immediate execution either before you RUN a program or while you are in the middle of a break to set breakpoints before lines 300, 350 and 380.

The CONTINUE command causes the computer to continue program execution after a breakpoint. Press **[CLR]** and type **CONTINUE** (or **CON**) and press **[ENTER]**.

The UNBREAK statement is used to remove the breakpoints you have set in a program. The only breakpoints that are removed are the ones that are set immediately before a line. In lines 130 and 230 in the previous example, UNBREAK can only remove the breakpoints set before lines 240 and 250. Line 130 always halts program execution when it is executed.

Using External Devices

Programs can be saved on external devices and then reloaded into memory and run. Data can be stored on external devices such as the TI *Wafertape*™ peripheral and programs created to update this data. External devices such as printers can be used to provide information in a form you can read. When the computer is transmitting data to or receiving data from an external device, the I/O display indicator is tumed on. You cannot use the keyboard at this time (including the **[OFF]** key).

If a file is open when you press the **[OFF]** key, the file is automatically closed before the computer is turned off. (Generally the term file refers to data stored on a mass storage device. However, in CC-40 BASIC a file refers to any information sent to an external device, even a printer.)

You can save programs on an external device and later run these saved programs by using the SAVE and OLD commands and the RUN statement. SAVE writes the program in memory to an external device in the internal machine format. The OLD command is used to load a SAVEd program into memory when you want to edit the program or VERIFY that it was loaded correctly. To execute the program, use the RUN statement.

Note: If you attempt to load (with OLD) or RUN a data file rather than a program file, you may have to press the reset key to reset the sytem.

You can list a program to an external device such as a printer by using the LIST command. LIST writes the program in memory to an external device in ASCII characters, the same form you see in the display.

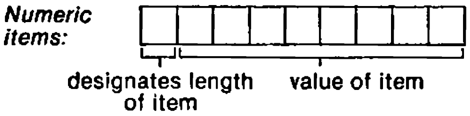
You can store, update, and print data to an external device by using the BASIC input/output statements. You must first open a file on an external device with the OPEN statement before you can use a BASIC input/output statement to access the file. The OPEN statement is used to inform the computer how the.data on the file is stored and the number that you will use to access the file. You do not use the OPEN statement when you use the BASIC commands, SAVE, OLD, or LIST.

Data Format

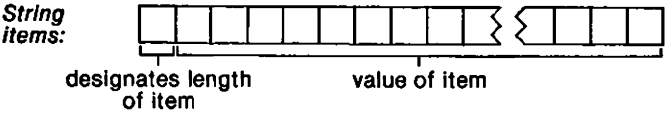
If data is to be stored, updated, or printed, you must specify to the computer the data format or how the data is recorded. When data is printed for people to read, the data should be written in ASCII characters (like the characters you see in the display). This type of data format is called display. When display-type data is printed, the numeric and string items are written according to the specifications in PRINT and appear the same as if the items were displayed in the CC-40.

If the data is stored on a mass storage device, the data should be recorded in the internal machine-code format. Data written in this internal-type format is stored in binary code, the type the computer uses to process data. Storing data in this format expedites processing and reduces the storage space required because the computer does not have to convert internal format to display characters and back again. When internal-type data is used, the numeric and string items are stored as shown below.

- Numeric items are stored in a form which occupies from 3 through 9 bytes of memory. The first byte is used to store the length of the numeric data item and the remaining 2 through 8 bytes are used to store the data value.



- String data is stored in the same manner, except that the maximum length for a string item extends through 256 bytes. The first byte is used to store the length of the string data and the remaining 0 through 255 bytes are used to store the string value.



Data Records

Data is stored, updated, and printed in a form called a *record.* A record consists of one or more of the processing units called *fields* and a collection of records is called a *file.* Records are numbered from 0 through 32767 where record #0 is the first record of the file, record #1 is the second record of the file, and so on. After a file is created, the CC-40 retrieves and updates data from the file in terms of records.

Record Length

When you write records to an external device, you can specify the *maximum* length for the records. If you do not specify a maximum record length, the computer assumes a default value according to the peripheral device you are using. When you design your records, be familiar with the lengths of the fields that make up a record. Plan your record so that you allow for the largest length needed.

The record length you specify determines how much space is reserved in the computer for storing a record of the file. If you attempt to write a record that is longer than the record length you specified, the computer breaks the record into smaller parts as described in PRINT (with files) in chapter 5. If you write a record that is smaller than the record length you specified, the record occupies only as much space in the file as is required to write its fields of data. When a record is read from a mass storage device, the computer determines the length of the record by indicators that were written when the record was created.

File Organization

When you store and update files on mass storage devices, the records can be arranged in sequence or in random order. If you want data to be stored so that you read it in sequence from the beginning, the file should be organized sequentially. Data stored in a sequential file is read the same as you would read data in a DATA statement. Files kept on tape must be sequential files. When you use external devices to print data for people to read, the records are always processed in sequence beginning with the first record.

If you want to process data directly without reading through all the data in sequence, the file should be organized as a relative (or random access) file. You specify that a file is a relative one when you use the OPEN statement to open the file. With relative files, you can access a particular record by using the REC clause in the INPUT, LINPUT, PRINT, and RESTORE statements. Relative files can also be accessed sequentially. Only certain types of devices support relative files.

File Processing Keywords

CC-40 BASIC provides an extensive range of file-processing features including sequential and random file organization and processing, variable length records, display and internal data formats, and file accessibility. This section describes the CC-40 BASIC keywords which are provided to facilitate file processing—FORMAT, DELETE, OPEN, INPUT, LINPUT, PRINT, CLOSE, EOF, and RESTORE. Refer to chapter 5 for a complete description of these keywords.

The FORMAT Command

The FORMAT command initializes the medium on an external storage device. You must format a new medium (such as a tape) before writing on it. If you format a medium that has data already written on it, all the data is erased. For example, the command below formats the tape on device 1.

```
FORMAT 1
```

The DELETE Statement

The DELETE statement can be used to delete a file from an external storage device, as well as to delete lines from a program (described earlier in "Editing Program Lines").

DELETE "2.MFILE"

Deletes the file named MFILE on device 2.

The OPEN Statement

The OPEN statement sets up a link between a peripheral device and a file number to be used in all the BASIC statements that refer to the file. In the OPEN statement you specify file attributes such as file accessibility, file organization, record length, and file type. The computer then creates the file according to the specifications in the OPEN statement. When you use the OPEN statement to open a file that already exists, the file attributes you specify must match those you used when the file was created (except how the file can be accessed).

For each opened file, the computer keeps an internal counter that points to the next record to be accessed. The counter is incremented by 1 each time a record is read or written. For random access files, be sure to use the REC clause if you read and write records on the same file within a program. Since the same internal counter is incremented when records are either read from or written to the same file, you could skip some records and write over others if REC is not used.

The following section describes the attributes that can be specified in the OPEN statement and the default values that are assumed if an attribute is omitted.

File Accessibility: The open-mode attribute of the OPEN statement specifies how the file can be accessed. UPDATE is assumed if no open-mode is specified.

Open-Mode Attribute	File Accessibility
INPUT	The computer can only *read* from the file.
OUTPUT	The computer can only *write* to the file.
UPDATE	The computer can both *read* from and *write* to the file.
APPEND	The computer can write data *only* at the end of the file. The records that already exist on the file cannot be accessed.

File Organization: RELATIVE for random access file or omitted for sequential file.

File Types (Data Formats): DISPLAY or INTERNAL. If file type is omitted, DISPLAY is assumed.

Record Length: VARIABLE followed by a numeric expression for the record length. If this option is omitted, the maximum record length is established by the peripheral device.

In the example below, the OPEN statement opens a file that is to be referenced as #5 in all of the BASIC statements that access the file. The file is opened on device 100 (which is assumed to-support relative files) with the file-name AFILE. The attributes of the file are relative (random access) organization, internal data format, INPUT open-mode, and a maximum record length of 64 bytes.

```
100 OPEN #5,"100.AFILE", RELATIVE, INTERNAL, INPUT,
    VARIABLE 64
```

The statement below opens a file named BFILE on device 1 as #7. The file can be both read from and written to (UPDATE open-mode), has sequential organization, and is recorded in display-type data.

```
150 OPEN #7,"1.BFILE"
```

The INPUT # Statement

The INPUT # statement is used to read data values from a file. You must use the same file-number to read this file as you did to open the file. When the INPUT # statement is executed, the data read from the file is assigned to the variables listed in the INPUT # statement.

Filling the INPUT # Variable-List

When the computer reads a file, it retrieves and stores an entire record in a temporary storage area called an input/output (I/O) buffer. Values are then assigned to the variables in the variable-list from left to right, using the data items (or fields) in the I/O buffer. A separate buffer is provided for each opened file.

If the variable-list of the INPUT statement is longer than the number of fields held in the I/O buffer, the computer retrieves the next record from the file and uses its fields to complete the variable-list. When a variable-list has been filled with the corresponding values, the fields left in the buffer are discarded unless the INPUT statement ends with a comma (as described later in "Pending Input Conditions").

The statements below open a file that is referred to as #3. The computer reads a record into the input buffer and assigns the fields in the record to the variables in the INPUT statement. If there are more fields in the buffer than are needed to assign to the variables, the remaining fields are discarded. If there are not enough fields to assign, the computer reads another record.

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,A$,J,K,L,B$,P,Q,R
```

Pending Input Conditions

An INPUT statement that ends with a comma creates a pending input condition. Any remaining fields in the input buffer are maintained for the next INPUT statement that reads the file. If this next INPUT statement has no REC clause, the computer starts assigning the remaining fields in the buffer to the variables in the INPUT statement. If the INPUT statement contains a REC clause, the remaining fields are discarded and the specified record is read into the I/O buffer. If a pending input condition exists when a PRINT, RESTORE, or CLOSE statement accesses the file, the remaining fields are also discarded.

The statements below open a file and create a pending input condition. After the variables are assigned in the first INPUT statement, any fields left in the input buffer are retained. When the next INPUT statement is executed, the remaining fields are assigned to the variables.

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,A$,J,K,L,B$,P,Q,R,
120 INPUT #3,C$,A,B,C
```

Input # and Data Formats

When the INPUT statement reads display-type data, the fields are separated by the commas that appear between the fields. Display-type records look like the data in a DATA statement. Numeric and string items must appear with their separators. Each field in a display-type record is checked to ensure that numeric values are placed in numeric variables.

In the example below, the first time the INPUT statement is executed, it assigns the fields in the first record to the variables. The second time the INPUT statement is executed, the fields in the second record are assigned to the variables. Note that there is no field in the buffer for the last variable, so the next record is read. When the INPUT statement attempts to assign a field to the last variable, an error occurs. The variable is a numeric variable but the field is not a numeric value.

```
(Record #0 on file #3) Jones, 95,98,65,32,78
(Record #1 on file #3) Smith, 67,87,66,90
(Record #2 on file #3) Lee,89,88,90,67,90
```

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,NAME$,A,B,C,D,E
120 GOTO 110
```

When the INPUT statement reads internal-type data, the length byte stored with each data item is used to separate the fields. The only validation performed on internal-type data is to ensure that numeric data is from 2 through 8 characters long.

The LINPUT # Statement

Like the INPUT # statement, the LINPUT # statement is used to read records from a file. However, LINPUT places all commas, leading and trailing spaces, semicolons, and quotation marks into string variables. The INPUT statement places these symbols into variables only if they are enclosed in quotation marks.

The PRINT # Statement

The PRINT # statement writes data values to a file. You must use the same file-number in opening and writing to the file. When the PRINT # statement is executed, the values of the items in the print-list are written to the file.

To write a record to the end of a sequential file, you can use the open-mode APPEND (but you cannot access the other records in the file). For UPDATE mode you must first read to the end of a sequential file before you write the new record. Using the PRINT statement before the end-of-file is reached results in a loss of data because the PRINT statement always defines a new end-of-file each time it is executed.

The values of the variables in the PRINT statement are written in a temporary storage area called an I/O buffer. A separate buffer is provided for each open file number. If the PRINT statement ends with a comma or a semicolon, a pending print is created.

Pending Print Conditions

When a PRINT statement ends with a comma or semicolon, the values of the print-list are retained in the I/O buffer for the next PRINT statement that writes to the file. If this next PRINT statement has no REC clause, the computer places the values of the print-list into the I/O buffer immediately following the fields already there. If the PRINT statement has a REC clause, the computer writes the pending print •hat is in the I/O buffer to the file at the position indicated by the internal counter. Then the new PRINT statement is executed.

If a pending print condition exists and an INPUT statement that accesses the file is encountered, the pending print record is written to the file at the position indicated by the internal counter and the internal counter is incremented. Then the INPUT statement is performed as usual. If a pending print exists when a CLOSE or RESTORE statement accesses the file, the pending print is written before the file is closed or restored.

For example, the following statements open a file for output, accept data from the keyboard, and write it to the file until a **$END** is entered.

```
100 OPEN #6,"1.PENDING",INTERNAL,OUTPUT
110 INPUT A$
120 IF A$="$END" THEN CLOSE #6:STOP
130 INPUT D,E
140 PRINT #6,A$,D,E,
150 GOTO 110
```

PRINT # and Data Formats

Refer to PRINT (with files) for information on how the PRINT statement writes a record in internal- or display-type data format. Note that if you print a file in display-type format that the computer will later read, the file must look the same as it does in a DATA statement. You must include the comma separators and quotation marks needed by the INPUT statement. When the data is read from the file, the computer separates the fields by the comma separators placed between them.

If the file in the example above had been opened with a display-type data format, the PRINT to the file must write print separators between the values for them to be read later.

```
100 OPEN #6,"1.PENDING",DISPLAY,OUTPUT
110 INPUT A$
120 IF A$="$END" THEN CLOSE #6:STOP
130 INPUT D,E
140 PRINT #6,A$; ",";D; ","; E; "," ;
150 GOTO 110
```

The CLOSE Statement

The CLOSE statement breaks the link between the file-number and the peripheral device. You cannot access this file until you OPEN it again. If you attempt to close a file that is not open, an error occurs. The CLOSE statement can be used to delete a file on some peripheral devices.

The following statements open the file CFILE on device 2, read three fields, and close the file.

```
100 OPEN #3,"2.CFILE",INTERNAL
110 INPUT #3, A$,D,E
120 CLOSE #3
```

The EOF Function

The EOF function determines if an end-of-file has been reached. The EOF function can be placed before an INPUT statement to test the file status before attempting to read from the file. The value that is returned by EOF is 0 if you are not at the end of the file and −1 if you are at the end of the file.

For example, the statements below open a file and check if the end-of-file has been reached before trying to read a record. When the end-of-file is reached, the file is closed.

```
100 OPEN #3,"2.CFILE",INTERNAL
105 IF EOF(3) THEN CLOSE #3:STOP
110 INPUT #3, A$,D,E
115 PRINT A$;D;E:PAUSE 1
117 GOTO 105
```

The RESTORE Statement

The RESTORE statement can be used to reposition an open file at record zero (for a sequential file) or at a specific record (for a relative file). If RESTORE refers to a relative file and the REC clause is not used, the file is repositioned to record zero.

For example, the statements below open a file referred to as #5, accept data from the keyboard and write the processed data to the file. The file is then repositioned to the first record. A printer is opened with a file number of 1. The data is read from file #5 and printed on file #1. When the end-of-file is reached, the message End of data is displayed.

```
100 OPEN #5,“1.RESFILE”,INTERNAL
110 INPUT “Enter street: ”;ST$
120 INPUT “Name: ”;A$
130 IF A$=“END$” THEN 170
140 INPUT “Address: ”;B$,“Zip: ”;C$
150 PRINT #5,A$&“family”,B$&“ ”&ST$;“794”&C$
160 GOTO 110
170 RESTORE #5
180 OPEN #1,“50”,OUTPUT
190 IF EOF(5) THEN PRINT “End of data”:PAUSE:CLOSE #5:STOP
200 INPUT #5,NAME$,ST$,ZIP$
210 PRINT #1,NAME$,ST$,ZIP$
220 GOTO 190
```

CHAPTER V
REFERENCE SECTION

This chapter is an alphabetical list of the CC-40 BASIC command, statement, and function keywords. Each keyword is explained in the following sections.

The Format section gives the complete syntax of the keyword, using the following conventions.

- KEYWORDS are capitalized.
- *Variables* are in italics.
- All parentheses are mandatory. Parentheses included with an optional element must be included when the optional element is used.
- Optional elements are enclosed in [brackets].
- Items that may be repeated are indicated by ellipses (...).
- Items representing alternative forms are presented one above the other and are enclosed in {braces}.

The Description gives the keyword's use or function and includes the options that the keyword can use.

The Cross Reference section refers to similar and complementary keywords, where appropriate.

The Example section gives examples of the keyword's use, where appropriate.

ABS

Format

ABS(*numeric-expression*)

Description

The ABS function gives the absolute value of *numeric-expression*. If *numeric-expression* is positive or zero, ABS returns the value of *numeric-expression*. If *numeric-expression* is negative, ABS returns the negative of the value. The result of ABS is always positive or zero.

Examples

```
370 PRINT ABS(42.3):PAUSE
```

Prints 42.3.

```
140 VV=ABS(-6.124)
```

Sets VV equal to 6.124.

ACCEPT

Format

ACCEPT [[AT(*column*)] [SIZE(*numeric-expression*)] [BEEP]
[ERASE ALL] [VALIDATE(*data-type, ...*)] [NULL(*expression*)] ,]
variable

Description

The ACCEPT statement suspends program execution until data is entered from the keyboard. The options available with ACCEPT make it more versatile for keyboard entry than the INPUT statement. ACCEPT can accept data at any display position, sound an audible tone (beep), erase all or part of the display, limit the number and type of characters accepted, and provide a default value for the input variable.

AT(*column*) positions the cursor and the beginning of the input field at the specified *column*, which must be from 1 to 31. If AT is omitted, input begins in column one unless a previous input/output statement left the cursor positioned in columns 2 through 31, in which case input is accepted at the cursor location.

SIZE(*numeric-expression*) allows up to the absolute value of *numeric-expression* characters to be input. If *numeric-expression* is positive, the input field is cleared before input is accepted. If *numeric-expression* is negative, the input field is not cleared, thus allowing a default value previously placed into the field by a DISPLAY or PRINT statement to be entered. If SIZE is omitted, the ACCEPT statement clears the display from the current cursor position to the end of the 80-column line. If SIZE is used, the cursor is left in the first position following the input field for subsequent input/output statements.

BEEP sounds a short tone for each BEEP in the statement, to indicate that the computer is ready to accept input.

ERASE ALL clears the entire display before accepting input, and positions the cursor to column one. If AT is used, the data is accepted starting at the position specified by *column*.

(continued)

ACCEPT

(continued)

VALIDATE(*data-type*) allows only certain characters to be entered from the keyboard. Note that default values are not validated. *Data-type* specifies which characters are acceptable.

Data-type can be a string expression which specifies the characters that are permitted. Only one string expression may be specified for *data-type.* The following can also be used as *data-types.* If more than one *data-type* is specified, a character from any of the types specified is acceptable.

ALPHA	permits all alphabetic characters.
UALPHA	permits only uppercase alphabetic characters.
DIGIT	permits 0 through 9.
NUMERIC	permits 0 through 9, ".", "+", "–", and "E".
ALPHANUM	permits all alphabetic characters and 0 through 9.
UALPHANUM	permits only uppercase alphabetic characters and 0 through 9.

NULL(*expression*) provides a default value to be assigned to the variable if **[ENTER]** is pressed with a blank (or null) input field.

During the execution of an ACCEPT statement, the following types of entries are also permitted.

- The **[FN]** key can be used to input keywords and user-assigned strings from the keyboard.
- A numeric expression can be entered if *variable* is numeric. The expression is evaluated and the result is assigned to *variable.*
- **[SHIFT] [ENTER]** can be pressed to cause the input data to be ignored, the value of *variable* to remain unchanged, and the program to proceed to the next statement. If NULL is included, it is also ignored.

Note: When an ACCEPT statement is waiting for data, **[CLR]** clears only the input field, **[CTL]** ↑ (home) and **[CTL]** ← (back tab) move the cursor to the beginning of the input field, and **[CTL]** → (right arrow) has no effect.

(continued)

(continued)

Cross Reference

INPUT, LINPUT

Examples

```
100 ACCEPT AT(3) ERASE ALL,T
```

Clears the display, accepts data starting in column 3, and places the data into variable T.

```
320 ACCEPT VALIDATE("yn") SIZE(1),A$
```

Accepts a one character field consisting of either y or n into the variable A$.

```
430 ACCEPT AT(3) SIZE(-5) BEEP VALIDATE(DIGIT,"+-"),X
```

Beeps, then accepts up to 5 characters for the variable X, starting at column 3. The input characters must consist of digits or the characters + or −. Because the SIZE specification is negative, the input field is not erased prior to accepting input.

```
570 ACCEPT NULL(PI),C
```

Accepts data for C. If no data has been entered when **[ENTER]** is pressed, the value of PI is stored in the variable C.

ACS

Format

ACS(*numeric-expression*)

Description

The ACS (arccosine) function calculates the angle whose cosine is *numeric-expression.* The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ACS function for the three angle settings is shown below.

Units	**Range of Calculated Angles**
Degrees	0 ≤ ACS(X) ≤ 180
Radians	0 ≤ ACS(X) ≤ PI
Grads	0 ≤ ACS(X) ≤ 200

Examples

```
100 DEG
```

Selects DEG angle setting.

```
110 PRINT ACS(1):PAUSE
```

Prints 0.

```
220 RAD
```

Selects RAD angle setting.

```
230 T=ACS(0.75)
```

Sets T equal to .72273424781339.

SUBPROGRAM

ADDMEM

Format

CALL ADDMEM

Description

The ADDMEM subprogram allows the Random Access Memory (RAM) contained in an installed *Memory Expansion* cartridge to be appended to the useable resident memory. The amount of memory added is described in appendix J.

CALL ADDMEM cannot be used in a program. The error message `No RAM in cartridge` is displayed if no *Memory Expansion* cartridge is installed when CALL ADDMEM is executed.

When the memory in a *Memory Expansion* cartridge has been appended to resident memory, the system is initialized if a loss of memory is detected (i.e. power is lost or the cartridge is removed) or the reset key is pressed.

The memory in a *Memory Expansion* cartridge remains appended to the resident memory until a NEW ALL command is executed, the computer is turned on without the cartridge installed, the batteries are removed, or the system is initialized.

Example

```
CALL ADDMEM
```

Allows the use of memory supplied by the installed *Memory Expansion* cartridge. See chapter 3 or appendix J for more information.

ASC

Format

ASC(*string-expression*)

Description

The ASC function returns the ASCII character code of the first character of *string-expression*. The message `Bad argument` is displayed if *string-expression* is a null string. A list of the ASCII codes is given in appendix D. The ASC function is the inverse of the CHR$ function.

Cross Reference

CHR$

Examples

```
100 PRINT ASC("A"):PAUSE
```

Prints 65.

```
130 B=ASC("1")
```

Sets B equal to 49.

```
790 DISPLAY ASC("HELLO"):PAUSE
```

Displays 72.

ASN

Format

ASN(*numeric-expression*)

Description

The ASN (arcsine) function calculates the angle whose sine is *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ASN function for the three angle settings is shown below.

Units	**Range of Calculated Angles**
Degrees	$-90 \leq \text{ASN(X)} \leq 90$
Radians	$-\text{PI}/2 \leq \text{ASN(X)} \leq \text{PI}/2$
Grads	$-100 \leq \text{ASN(X)} \leq 100$

Examples

```
140 DEG
```

Selects DEG angle setting.

```
150 PRINT ASN(1):PAUSE
```

Prints 90.

```
240 RAD
```

Selects RAD angle setting.

```
250 B=ASN(.9)
```

Sets B equal to 1.119769514999.

ATN

Format

ATN(*numeric-expression*)

Description

The ATN (arctangent) function calculates the angle whose tangent is *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ATN function for the three angle settings is shown below.

Units	**Range of Calculated Angles**
Degrees	–90 < ATN(X) < 90
Radians	– PI/2 < ATN(X) < PI/2
Grads	– 100 < ATN(X) < 100

Examples

```
130 GRAD
```

Selects GRAD angle setting.

```
140 PRINT ATN(30):PAUSE
```

Prints 97.87871952.

```
810 RAD
```

Selects RAD angle setting.

```
820 Q=ATN(2.5)
```

Sets Q equal to 1.190289949683.

ATTACH

Format

ATTACH *sub-name1* [, *sub-name2* ...]

Description

The ATTACH statement is used to preserve the values of variables used in subprogram(s) between calls to the subprogram(s). When the ATTACH statement is executed, memory space is allocated for the variables and the values are initialized. The variables are not initialized when the subprogram is called and are not destroyed when the subprogram terminates.

An ATTACH statement may appear in the main program or in any subprogram. A subprogram can ATTACH itself. The message `Program not found` is displayed if a specified *sub-name* cannot be found. If a specified *sub-name* is an assembly language subprogram, the message `Bad program type` is displayed.

Attaching a repeatedly used subprogram reduces execution time. However, while the subprogram remains attached, the memory space for the variables remains allocated. ATTACH should be used only when sufficient memory space is available. (Refer to FRE for more information.)

The RELEASE statement is used to release an attached subprogram.

Cross Reference

FRE, RELEASE

(continued)

ATTACH

(continued)

Example

The following program illustrates how to attach a subprogram.

```
100 FOR J=1 TO 5
110 CALL X
120 NEXT J
```

Prints 0 0 0 0 0 because the variable values in subprogram X are initialized each time it is called.

```
130 ATTACH X:PRINT
```

Attaches subprogram X and clears the display.

```
140 FOR J=1 TO 5
150 CALL X
160 NEXT J
```

Prints 0 1 2 3 4 because the variable values are not initialized when X is called and are not destroyed when X is terminated.

```
170 SUB X
180 PRINT J;:PAUSE 2
190 J=J+1
200 SUBEND
```

BREAK

Format

BREAK [*line-number-list*]

Description

The BREAK statement is used to suspend program execution at specific points, called breakpoints, in a program. Breakpoints can be specified in two ways. If *line-number-list* is not given with the BREAK statement, a breakpoint occurs when the BREAK statement is executed. If *line-number-list* is given with the BREAK statement, breakpoint(s) are set immediately before the line(s) listed in *line-number-list*. The **[BREAK]** key also causes the program to stop as if a BREAK statement had been executed.

When a breakpoint occurs, the message `BREAK` is displayed.

A breakpoint set immediately before a program line remains in the program until the UNBREAK statement is used to remove it or until the line is edited or deleted.

BREAK is useful in debugging a program. When program execution halts at a breakpoint, variables can be printed and calculations can be performed to determine why a program is not executing correctly. The CONTINUE command can be used to resume program execution.

Cross Reference

CONTINUE, ON BREAK, UNBREAK

Examples

```
150 BREAK
```

Causes a breakpoint when the BREAK statement is executed.

```
100 BREAK 120,130
```

Causes breakpoints before execution of lines 120 and 130.

```
BREAK 10,400,130
```

Causes breakpoints before execution of lines 10, 400, and 130.

CALL

Format

CALL *subprogram-name* [(*argument-list*)]

Description

The CALL statement transfers control to a subprogram. The first subprogram found with the given *subprogram-name* is executed. After the subprogram is executed, program control returns to the first statement following the CALL statement. The valid types of subprograms are listed below in the order in which the search for the subprogram is performed.

1. Built-in subprograms
2. Assembly language subprograms which are loaded with CALL LOAD
3. BASIC subprograms defined using SUB
4. Subprograms located in *Solid State Software* cartridges

Argument-list is used to pass data to the subprogram. The number and types of arguments in *argument-list* must match the parameters in the *parameter-list* of the subprogram or an error occurs.

Each built-in subprogram is discussed under its own entry in this manual. Assembly language subprograms are discussed in the Editor/Assembler manual. BASIC subprograms are discussed in chapter 4 and in this chapter under SUB.

Cross Reference

SUB

Examples

```
CALL CLEANUP
```

Deletes unused variable names from the system.

```
100 CALL SETLANG(1)
```

Changes the language setting to German. This subprogram requires a language number parameter.

SUBPROGRAM

CHAR

Format

CALL CHAR(*character-code, pattern-identifier*)

Description

The CHAR subprogram defines special display characters. The characters are defined in a 5-by-8 grid by specifying which dots are "on" and which are "off." Up to seven special characters can be defined at one time. The characters can be displayed by using CHR$ in a DISPLAY or PRINT statement. If a special character is in the display when the pattern definition is changed, the displayed character changes immediately.

Note: Characters defined with the CHAR subprogram are not retained when the computer is turned off.

Character-code specifies which special display character is to be defined. *Character-code* must be a value from 0 through 6.

Pattern-identifier is a string expression whose value defines the pattern for one or more special display characters.

- The first 16 characters of *pattern-identifier* define the specified *character-code*. If *pattern-identifier* is less than 16 characters, the remaining characters are considered to be zeros.
- If *pattern-identifier* is greater than 16 characters, the extra characters define the next sequential *character-code*, until all the *pattern-identifier* characters have been assigned to a *character-code*.
- If *pattern-identifier* has enough characters to define past *character-code* 6, the extra characters are ignored.
- If *pattern-identifier* is a null string, an error occurs.

Each pair of characters in *pattern-identifier* describes the pattern in one row of the grid. The left character is 1 or 0, indicating that the left block is "on" or "off," respectively. The right character is a hexadecimal digit (0 through F) whose binary equivalent is used to determine which dots are on and off, as described above. The rows are described from top to bottom. Note that there is a slight break in the display between the top seven rows and the eighth row.

(continued)

CHAR SUBPROGRAM

(continued)

The following table shows all possible on/off conditions for each row, and the binary and hexadecimal codes for each condition.

					Binary Code	Hexadecimal
Dot Pattern					**(0 = Off: 1 = On)**	**Code**
					00000	00
				*	00001	01
			*		00010	02
			*	*	00011	03
		*			00100	04
		*		*	00101	05
		*	*		00110	06
		*	*	*	00111	07
	*				01000	08
	*			*	01001	09
	*		*		01010	0A
	*		*	*	01011	0B
	*	*			01100	0C
	*	*		*	01101	0D
	*	*	*		01110	0E
	*	*	*	*	01111	0F
*					10000	10
*				*	10001	11
*			*		10010	12
*			*	*	10011	13
*		*			10100	14
*		*		*	10101	15
*		*	*		10110	16
*		*	*	*	10111	17
*	*				11000	18
*	*			*	11001	19
*	*		*		11010	1A
*	*		*	*	11011	1B
*	*	*			11100	1C
*	*	*		*	11101	1D
*	*	*	*		11110	1E
*	*	*	*	*	11111	1F

(continued)

(continued)

Cross Reference

CHR$

Examples

To define the dot pattern pictured below, type the following line.

CALL CHAR(0,"04150E04040E1504")

	Left Block	Right Blocks				Hex Codes
ROW 1			*			04
ROW 2	*		*		*	15
ROW 3		*	*	*		0E
ROW 4			*			04
ROW 5			*			04
ROW 6		*	*	*		0E
ROW 7	*		*		*	15
ROW 8			*			04

To display the special character, enter **PRINT CHR$(0)**. Note that the underline cursor also appears in the display in this case.

To define the dot pattern pictured below, type the following line.

CALL CHAR(4,"0A110B12")

	Left Block	Right Blocks				Hex Codes
ROW 1		*		*		0A
ROW 2	*				*	11
ROW 3		*		*	*	0B
ROW 4	*			*		12
ROW 5						00
ROW 6						00
ROW 7						00
ROW 8						00

Since rows 5 through 8 are not specified, they are assumed to be zeros.

To display the special character, enter **PRINT CHR$(4)**. Note that the underline cursor also appears in the display in this case.

CHR$

Format

CHR$(*numeric-expression*)

Description

The CHR$ function returns the character corresponding to the ASCII character code specified by *numeric-expression*. The CHR$ function is the inverse of the ASC function. A list of the ASCII character codes for each character in the standard character set is given in appendix D.

CHR$ is used in PRINT and DISPLAY statements to display special characters defined with the CHAR subprogram or the extended character set not available on the keyboard (see appendix D). With peripherals, CHR$ can be used for control operations such as advancing a printer to a new page.

Cross Reference

ASC, CHAR

Examples

```
840 PRINT CHR$(72):PAUSE
```

Prints H.

```
900 X$=CHR$(33)
```

Sets X$ equal to !.

SUBPROGRAM

CLEANUP

Format

CALL CLEANUP

Description

The CLEANUP subprogram deletes unused variable names from the system. When CALL CLEANUP is executed, all variable names which are not used in the program currently in memory are removed and all open files are closed. If CALI. CLEANUP is executed when a program is stopped at a breakpoint, the CONTINUE command cannot be used to resume program execution.

CALL CLEANUP cannot be used in a program.

CLOSE

Format

CLOSE *#file-number* [, DELETE]

Description

The CLOSE statement terminates the association between a file and its current *file-number*. The file or device cannot be accessed by the program unless it is reopened. After a file is closed, *file-number* can be assigned to another file or device. If an attempt is made to CLOSE a file that is not open, an error occurs.

Any of the following actions close all open files.

- Editing the program or subprogram.
- Entering a NEW, RENUMBER, RUN, OLD, SAVE, or VERIFY command.
- Listing the program to a peripheral device.
- Calling the ADDMEM or CLEANUP subprogram.
- Turning the system off or pressing the reset key.

Normal program termination also closes all open files.

Some peripheral devices allow a file to be deleted at the time it is closed by adding DELETE to the statement. The manual for each peripheral device describes the use of DELETE.

Cross Reference

OPEN

Example

```
790 CLOSE #6
```

Closes file #6.

CONTINUE

Format

CONTINUE [*line-number*]

Description

The CONTINUE (or CON) command is used to resume execution after a breakpoint occurs. A program or subprogram may be continued at the line specified by *line-number*. *Line-number* must refer to a line number in the main program if the main program is stopped. If a subprogram is stopped, *line-number* must refer to a line number in that subprogram. Using an improper *line-number* produces unpredictable results.

The following actions do not allow a CONTINUE to resume execution after a breakpoint:

- Editing the program or subprogram.
- Entering a NEW, OLD, RENUMBER, RUN, SAVE, or VERIFY command.
- Listing the program to a peripheral device.
- Calling the ADDMEM or CLEANUP subprogram.
- Turning the system off or pressing the reset key.

Cross Reference

BREAK

COS

Format

COS(*numeric-expression*)

Description

The COS function calculates the trigonometric cosine of *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using thls function. See appendix E for a description of the limits of *numeric-expression*.

Examples

```
140 GRAD
```

Selects GRAD angle setting.

```
150 PRINT COS(30):PAUSE
```

Prints .8910065242.

```
240 RAD
```

Selects RAD angle setting.

```
300 T=COS(PI)
```

Sets T equal to −1.

DATA

Format

DATA *data-list*

Description

The DATA statement is used with the READ statement to assign values to variables. When a READ statement is executed, the values in *data-list* are assigned to the variables specified in the *variable-list* of the READ statement. *Data-list* consists of numeric or string constants, separated by commas. Leading and trailing spaces are ignored. A string constant that contains commas or leading or trailing spaces must be enclosed in quotes. A quotation mark within a quoted string is represented by two quotation marks. A null string is represented by two adjacent commas.

A DATA statement must be the only statement on a line. It may be located anywhere in a program or subprogram. If a program has more than one DATA statement, the DATA statements are read in sequential order beginning with the lowest numbered line.

The RESTORE statement can be used to reread DATA statements or to alter the order in which DATA statements are read.

Cross Reference

READ, RESTORE

(continued)

DATA

(continued)

Example

The program below reads and prints several numeric and string constants.

```
100 FOR A=1 TO 5
110 READ B,C
120 PRINT B;C:PAUSE 1.1
130 NEXT A
```

Lines 100 through 130 read five sets of data and print their values, two to a line.

```
140 DATA 2,4,6,7,8
150 DATA 1,2,3,4,5
160 DATA """THIS HAS QUOTES"""
170 DATA "NO QUOTES HERE"
180 DATA NO QUOTES HERE EITHER
190 FOR A=1 TO 7
200 READ B$
210 PRINT B$:PAUSE 1.1
220 NEXT A
```

Lines 190 through 220 read seven data elements and print each on its own line.

```
230 DATA 1, NUMBER,,TI
```

SUBPROGRAM

DEBUG

Format

CALL DEBUG

Description

The DEBUG subprogram is used to test assembly language subprograms. CALL DEBUG allows access to the assembly language debugger, which is briefly described in appendix I. Refer to the Editor/Assembler manual for more information.

DEG

Format

DEG

Description

The DEG statement sets the units for angle calculations to degrees. After the DEG angle setting is selected, all entered and calculated angles are measured in degrees. This setting is changed to RAD when NEW ALL is entered or the system is initialized.

Cross Reference

GRAD, RAD

DELETE

Format

DELETE $\left\{\begin{array}{l}\textit{line-group}\ [,\ \textit{line-group}\ ...] \\ \textit{"device.filename"}\end{array}\right\}$

Description

The DELETE (or DEL) statement is used to remove lines from a program in memory or to remove a file from external storage. *Line-group* specifies program lines to be deleted and may consist of the following.

Line-group	Effect
a single line number	Deletes that line.
line number –	Deletes that line and all following lines.
– line number	Deletes that line and all preceding lines.
line number – line number	Deletes that inclusive range of lines.

DELETE *line-group* cannot be used in a program line.

If *line-group* specifies a single line number that does not exist, the message `Line not found` is displayed. However, any remaining *line-groups* are deleted when the **[ENTER]** or **[CLR]** key is pressed. If the initial line of a range does not exist, the next higher numbered line is used as the initial line. If the final line does not exist, the next lower numbered line is used as the final line.

Device.filename is used to delete a file from an external storage device. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. *Device.filename* may be a string expression. Refer to the peripheral manuals for the device number for each peripheral device and for specific information about the form of *filename*.

You may also delete data files on some peripheral devices by using DELETE in the CLOSE statement. Refer to the appropriate peripheral manual for more information.

(continued)

DELETE

(continued)

Cross Reference

CLOSE

Examples

```
DELETE 10-50,90,110-220
```

Deletes lines 10 through 50, 90, and 110 through 220.

```
DELETE 900-
```

Deletes lines 900 through the end of the program.

```
DELETE -500, 750
```

Deletes all lines through 500 and line 750.

```
DELETE "1.file"
```

Deletes “file” from device 1.

DIM

Format

DIM *array-name(integer1* [, *integer2*] [, *integer3*]) [,...]

Description

The DIM statement specifies the characteristics of an array and reserves the necessary memory space for it. *Array-name* is a string or numeric variable name. The number of values in parentheses following *array-name* determines the number of dimensions in the array. Arrays with up to three dimensions are allowed. The values in parentheses represent the maximum values of the subscripts in each dimension of the array.

The lowest value of a subscript is zero. Therefore, the number of elements in each dimension is one more than the maximum subscript. For example, an array defined by DIM A(6) is a one dimensional array with seven elements, A(0) through A(6). If an array is not defined in a DIM statement, the maximum value of each subscript is 10.

When execution of a program begins, each element of a numeric array is set to zero, and each element of a string array is set to the null string.

An array can be dimensioned only once. A DIM statement must appear in the program at a lower numbered line than any other reference to its array. Remarks (REM) and tail remarks (!) are the only statements which may appear after a DIM statement on a multiple statement line. A DIM statement cannot appear in an IF THEN ELSE statement.

Examples

```
120 DIM X$(30)
```

Reserves space in the computer's memory for 31 elements of the array called X$. Each element is initialized to the null string.

```
430 DIM D(100),B(10,9)
```

Reserves space in the computer's memory for 101 elements of the array called D and 110 (11 times 10) elements of the array called B. Each element of each array is initialized to zero.

DISPLAY

Format

DISPLAY [[AT(*column*)] [BEEP] [ERASE ALL]
[SIZE(*numeric-expression*)] [USING *line-number* *string-expression* :] [*print-list*]

Description

The DISPLAY statement formats and displays the value(s) included in *print-list*. The options available with DISPLAY can be used to display data starting at any column position, sound an audible tone (beep), erase all or part of the display, limit the total number of characters displayed, and specify the format of the display.

AT(*column*) positions the cursor and the beginning of the display field at the specified column position from 1 through 80.

The evaluation of the TAB function and comma separators is relative to the specified starting position. However, if SIZE is not specified and the evaluation of *print-list* continues on a new line, the new line begins in column 1, not in the column specified by AT.

When AT is omitted, output starts at the current cursor position as left by previous input/output statements. If the current position is greater than 80, the cursor is reset to column 1. When AT is omitted, the TAB function and comma separators are always relative to column 1.

BEEP sounds a short tone for each BEEP in the statement.

ERASE ALL clears the entire 80-column line. If AT is omitted, the cursor position is set to column 1.

SIZE(*numeric-expression*) limits the total number of characters to the absolute value of *numeric-expression*. If *numeric-expression* is larger than the number of remaining positions, the display field extends from the current cursor position to the end of the 80-column line. The length of the display field, defined by SIZE becomes the new record length for purposes of

(continued)

DISPLAY

(continued)

evaluating the TAB function and comma separators in *print-list.* The specified field is always cleared prior to displaying data. Termination of the DISPLAY statement leaves the cursor in the first position following the display field. If SIZE is omitted and there is no trailing separator after *print-list*, termination of the DISPLAY statement clears the display from the last item displayed to the end of the 80-column line.

USING may be used to specify an exact format for the output. If USING is specified, it must appear last in the option list. Refer to IMAGE and USING for a description of format definition and its effect upon the output of the DISPLAY statement.

Print-list consists of numeric and string expressions, separated by commas or semicolons. For more details, see PRINT.

Cross Reference

IMAGE, PAUSE, PRINT, TAB, USING

Examples

```
120 DISPLAY AT(7),Y:PAUSE
```

Displays the value of Y starting at column 7 and clears everything following the number. The value actually appears in column 8 since the sign precedes the number.

```
150 DISPLAY N:PAUSE
```

Displays the value of N in column 1 of the display and clears the rest of the display.

```
190 DISPLAY ERASE ALL,B:PAUSE
```

Clears the entire display before displaying the value of B.

```
370 DISPLAY AT(C) SIZE(19) BEEP,X:PAUSE
```

Clears 19 characters starting at position C, beeps, and displays the value of X starting at position C.

END

Format

END

Description

The END statement terminates a program and may be used interchangeably with the STOP statement. Although the END statement may appear anywhere, ·it is usually placed as the last line in a program. The END statement is not required. A program automatically stops when the highest numbered line is executed.

The END statement closes all open files.

Cross Reference

STOP

EOF

Format

EOF(*file-number*)

Description

The EOF function is used to test whether there is another record to be read from a file. The value of *file-number* indicates the file to be tested and must correspond to the number of an open file. EOF returns a value which indicates the current position in the file as follows.

Value	Position
0	Not end-of-file
−1	Logical end-of-file

The logical end-of-file occurs when all records on the file have been input.

When using pending INPUT (see chapter 4), EOF does not indicate whether pending input data remains in memory.

Cross Reference

INPUT (with files)

Examples

```
140 PRINT EOF(3):PAUSE
```

Prints -1 if file #3 has reach the end-of-file and 0 if it has not reached the end-of-file.

```
710 IF EOF(27) THEN 1150
```

Transfers control to line 1150 if the end-of-file has been reached for file #27.

ERR

SUBPROGRAM

Format

CALL ERR(*error-code*, *error-type* [, *file-number*, *line-number*])

Description

The ERR subprogram returns the error code, error type and, optionally, the file number and line number of the last uncleared error. When an error occurs, a subroutine can be called (see ON ERROR) that contains CALL ERR. The error is cleared when this error-processing subroutine terminates with a RETURN.

Error-codes range from 0 through 127. The meaning of each error code is listed in appendix K.

Error-type is always 0 unless *error-code* is 0, which is an input/output (I/O) error. For an I/O error, *error-type* is an I/O error code specified by each I/O device. The range for I/O error codes is 1 through 255.

File-number is 0 unless the error is an I/O error. For an I/O error, *file-number* is the file number used in the I/O statement that caused the error.

Line-number is the number of the line being executed when the error occurred. It is not always the line that is the source of the problem since an error may occur because of values generated or actions taken elsewhere in a program.

If no error has occured, CALL ERR returns all values as zeros.

Cross Reference

ON ERROR, RETURN (with ON ERROR)

Examples

```
170 CALL ERR(A,B)
```

Sets A equal to the *error-code* and B equal to the *error-type* of the most recent uncleared error.

```
390 CALL ERR(W,X,Y,Z)
```

Sets W equal to the *error-code*, X equal to the *error-type*, Y equal to the *file-number*, and Z equal to the *line-number* of the most recent uncleared error.

SUBPROGRAM

EXEC

Format

CALL EXEC(*execution-address* [, *argument-list*])

Description

The EXEC subprogram is used to execute assembly language subprograms located at specific memory addresses. Normally, the POKE statement has been used to store these subprograms in memory that has been reserved by a CALL GETMEM statement. *Execution-address* is the memory address at which subprogram execution is to begin and must be a numeric expression from 0 through 65535.

Argument-list is used to pass values to and from the subprogram being executed.

Details on writing assembly language subprograms are provided in the Editor/Assembler manual.

Cross Reference

GETMEM, PEEK, POKE, RELMEM

EXP

Format

EXP(*numeric-expression*)

Description

The EXP function returns the result of e^x, where x is *numeric-expression*. The value of e is 2.71828182846.

Examples

```
150 Y=EXP(7)
```

Sets Y equal to the value of e raised to the seventh power, which is 1096.633158429.

```
390 L=EXP(4.394960467)
```

Sets L equal to the value of e raised to the 4.394960467 power, which is 81.04142688867.

FOR TO STEP

Format

FOR *control-variable* = *initial-value* TO *limit* [STEP *increment*]

Description

The FOR TO STEP statement is used with the NEXT statement to form a loop, which is a series of statements performed a specific number of times. *Control-variable* is an unsubscripted numeric variable that acts as a counter for the loop. *Initial-value*, *limit*, and *increment* are numeric expressions.

When the FOR statement is executed, *initial-value* is assigned to *control-variable*. If *initial-value* exceeds *limit*, the loop is skipped and execution continues with the statement after the NEXT statement. Otherwise, the statements following the FOR statement are executed until the corresponding NEXT statement is executed. *Increment* is then added to *control-variable*. If *control-variable* is not greater than *limit*, execution returns to the statement following the FOR statement.

When *control-variable* becomes greater than *limit*, control transfers to the statement following the NEXT statement. *Control-variable* then equals the value it had the last pass through the loop plus the value of *increment*.

A loop that is contained entirely within another loop is called a nested loop. Nested loops must use different control variables. Program execution can be transferred out of a loop using GOTO, GOSUB, or IF THEN ELSE and then returned back into the loop.

If a NEXT statement is executed before its corresponding FOR statement, an error occurs.

STEP specifies the *increment* that is added to *control-variable* each time the loop is executed. If STEP is omitted, the *increment* is one. If the *increment* is negative, *control-variable* is decreased each time through the loop and *limit* should be less than *initial-value*. The loop is skipped if *initial-value* is less than *limit*. Otherwise, the loop is executed until *control-variable* is less than *limit*.

(continued)

FOR TO STEP

(continued)

Cross Reference

NEXT

Examples

```
140 FOR A=1 TO 5 STEP 2
⋮
190 NEXT A
```

Executes the statements between FOR and NEXT A three times, with A having values of 1, 3, and 5. After the loop is finished, A has a value of 7.

```
250 FOR J=7 TO -5 STEP -.5
⋮
350 NEXT J
```

Executes the statements between FOR and NEXT J 25 times, with J having values of 7, 6.5, 6, ..., −4, −4.5, and −5. After the loop is finished, J has a value of −5.5.

```
700 FOR X=1 TO 2 STEP -1
⋮
780 NEXT X
⋮
```

Does not execute the loop because *increment* is negative and the *initial-value* is already less than the *limit.*

FORMAT

Format

FORMAT *device*

Description

The FORMAT statement initializes the current medium on an external storage device. Formatting a storage medium destroys all previously stored data.

Device is the number associated with each physical device and can be from 1 through 255. Refer to the peripheral manuals to obtain the device code for each peripheral device.

Example

```
140 FORMAT 1
```

Initializes the tape currently in the *Wafertape* drive. All data previously stored on the tape is destroyed.

FRE

Format

FRE(*numeric-expression*)

Description

The FRE function returns information about the current use of memory in the computer. Memory space is divided into four types as follows.

- Space reserved for operation of the system.
- Space occupied by the current program in memory.
- Space temporarily reserved by a running program.
- Space currently available (free space).

The value of *numeric-expression* specifies the type of information desired as follows.

Value	Meaning
0	Total memory space *not* reserved for system operation.
1	Total space occupied by the program currently in memory. The value returned includes 11 bytes for program overhead.
2	Total amount of free space and temporarily reserved space.
3	Size of the largest block of free memory space.
4	Total amount of free memory space.
5	Number of individual blocks of free memory space.

Example

```
300 A=FRE(3)
```

Sets A equal to the number of bytes available in the largest contiguous block of free memory. This statement is useful for determining how much memory can be reserved by the GETMEM subprogram.

SUBPROGRAM

GETLANG

Format

CALL GETLANG(*numeric-variable*)

Description

The GETLANG subprogram places the code of the international language being used to display system messages and errors into *numeric-variable*. The language identification code is set using the SETLANG subprogram. The language is set to English when the system is initialized.

The following are the assigned language codes.

0 = English
1 = German
2 = French
3 = Italian
4 = Dutch
5 = Swedish
6 = Spanish

Cross Reference

SETLANG

Example

```
120 CALL GETLANG(A)
```

Places the code of the current language setting into A.

GETMEM SUBPROGRAM

Format

CALL GETMEM(*numeric-expression*, *numeric-variable*)

Description

The GETMEM subprogram is used to reserve memory space for storing data and assembly language programs. *Numeric-expression* specifies the number of bytes to reserve and must be a value from 1 through 32765. The error message `Memory full` is displayed if the number of bytes specified is not available.

The lowest address of the reserved memory space is stored in *numeric-variable*. This value must be retained if RELMEM is to be used to release the memory for other uses. The highest address of the reserved memory space can be calculated as follows.

highest memory address = *numeric-variable* + *numeric-expression* – 1

When space has been reserved, CALL POKE and CALL PEEK can be used to access the memory directly. Data may be placed in the reserved area one byte at a time with the CALL POKE subprogram and read with the CALL PEEK subprogram. If an assembly language subprogram is loaded into reserved memory using CALL POKE, CALL EXEC may be used to execute it. The assembly language program must not use memory space outside of the reserved area.

In addition to the requested amount of memory, GETMEM requires four bytes of memory for its own operation. Thus, if the FRE function is used to obtain the size of the largest available block of memory, that value must be reduced by four to obtain the largest block which can be allocated by GETMEM.

The largest block of memory allocated by GETMEM should be significantly less than the largest block available. Sufficient memory space must remain available for statements that require additional temporary memory.

(continued)

(continued)

Cross Reference

EXEC, FRE, PEEK, POKE, RELMEM

Example

```
140 CALL GETMEM(100, ADDRESS)
```

Reserves a block of 100 bytes of memory and places the lowest address of the reserved memory area into ADDRESS.

GOSUB

Format

GOSUB *line-number*

Description

The GOSUB statement transfers control to the subroutine that begins at *line-number*. The statements of the subroutine are executed until a RETURN statement is encountered. A RETURN statement returns control to the statement immediately following the GOSUB statement.

Subroutines may be called any number of times in a program and may call themselves or other subroutines. The GOSUB statement cannot be used to transfer control into or out of a subprogram.

Cross Reference

ON GOSUB, RETURN

Example

```
100 GOSUB 200
```

Transfers control to line 200. The statement at line 200 and all the statements that follow are performed until RETURN is encountered. RETURN transfers control to the statement following the GOSUB statement.

GOTO

Format

GOTO *line-number*

Description

The GOTO statement transfers control unconditionally to another line within a program. When a GOTO statement is executed, control is passed to the first statement on the line specified by *line-number.*

The GOTO statement cannot be used to transfer control into or out of a subprogram.

Example

```
100 GOTO 300
```

Transfers control to line 300.

GRAD

Format

GRAD

Description

The GRAD statement sets the units for angle calculations to grads. After the GRAD angle setting is selected, all entered and calculated angles are measured in grads. This setting is changed to RAD when NEW ALL is entered or the system is initialized.

Cross Reference

DEG, RAD

IF THEN ELSE

Format

IF *condition* THEN *action1* [ELSE *action2*]

Description

The IF THEN ELSE statement performs one of two specified actions based on a specified condition. If *condition* is true, *action1* is performed. If *condition* is false, *action2* is performed. If ELSE is omitted and *condition* is false, control is transferred to the next line.

Condition can be either a relational expression or a numeric expression. When a relational expression is evaluated, the result is 0 if it is false and −1 if it is true. When a numeric expression is evaluated, a zero value is considered to be false and a nonzero value is considered to be true.

Action1 and *action2* may be line numbers, statements, or groups of statements separated by colons. If a line number is used, control is transferred to that line. If statements are used, those statements are performed.

The IF THEN ELSE statement must be contained on one line and is terminated by the end of the line. IF THEN ELSE statements can be nested by including an IF THEN ELSE statement in *action1* or *action2*. If a nested IF THEN ELSE statement does not contain the same number of THEN and ELSE clauses, each ELSE is matched with the closest unmatched THEN.

IF THEN ELSE statements cannot contain DIM, IMAGE, SUB, or SUBEND statements.

Examples

```
100 IF Y<5 THEN 150
```

If the value of Y is less than 5, statement 150 is executed. If Y is greater than or equal to 5, the next statement is executed.

(continued)

IF THEN ELSE

(continued)

```
140 IF MBB=0 THEN 200
150 PRINT "NON-ZERO":PAUSE 2
```

If MBB is zero, control passes to line 200. If MBB is not zero, NON-ZERO is displayed and program execution halts for 2 seconds before executing the next statement.

```
230 IF X>5 THEN GOSUB 300 ELSE X=X+5
```

If the value of X is greater than 5, GOSUB 300 is executed. When the subroutine is completed, control returns to the line following the IF THEN ELSE statement. If X is 5 or less, X is set equal to X+5 and control passes to the next line.

```
250 IF Q THEN C=C+1:GOTO 500 ELSE L=L/C:GOTO 300
```

If Q is not zero (true), C is set equal to C+1 and control is transferred to line 500. If Q is zero (false), L is set equal to L/C and control is transferred to line 300.

```
290 IF A$="Y" THEN COUNT=COUNT+1:DISPLAY AT(4), "HERE WE
    GO AGAIN!":PAUSE 1.5:GOTO 400
```

If A$ is equal to "Y", COUNT is incremented by 1, a message is displayed, and control is transferred to line 400. If A$ is not equal to "Y", control passes to the next line.

```
350 IF HRS<=40 THEN PAY=HRS*WAGE ELSE
    PAY=HRS*WAGE+.5*WAGE *(HRS-40):OT=1
```

If HRS is less than or equal to 40, PAY is set equal to HRS*WAGE and control passes to the next line. If HRS is greater than 40, PAY is set equal to HRS*WAGE+.5*WAGE*(HRS−40), OT is set equal to 1, and control passes to the next line.

```
700 IF A=1 THEN IF B=2 THEN C=3 ELSE D=4
```

If A is equal to 1 and B is equal to 2, C is set equal to 3 and control passes to the next line. If A is equal to 1 and B is not equal to 2, D is set equal to 4 and control passes to the next line. If A is not equal to 1, control passes to the next line.

IMAGE

Format

IMAGE *string-constant*

Description

The IMAGE statement is used to define an output format. The format is used by placing the line number of the IMAGE statement in the USING option of DISPLAY or PRINT (see USING in this chapter). *String-constant* may be enclosed in quotation marks. If *string-constant* is not enclosed in quotation marks, leading and trailing blanks are ignored.

The IMAGE statement must be the only statement on a program line and must appear in the program or subprogram which uses it. When an IMAGE statement is encountered, execution immediately continues with the next line of the program.

A format definition is divided into format fields and literal fields. When a PRINT or DISPLAY statement uses a format definition, the format fields are replaced by the values of the print items and the literal fields are printed as they appear in the format definition. An explanation of a format definition is given below.

Format Definition

The three characters which may be used to define a format field are the number sign (#), the decimal point (.), and the exponentiation symbol (^). The number sign defines a character position in the format field. It is replaced by one of the characters from the ASCII representation of the value of the print item. The decimal point is used in a decimal format field to specify the position of the decimal point. The exponentiation symbol (^) is used in an exponential format field to specify the number of positions in which to print the exponent value. All other characters are literal and thus form literal fields.

The five types of fields in a format definition are integer, decimal, exponential, string, and literal. The rules which apply to each type are listed below.

(continued)

IMAGE

(continued)

Integer Field

- Up to 14 significant digits may be specified.
- An integer field is composed of number signs.
- When the number does not fill the field, the number is right-justified.
- When the number is longer than the field, asterisks (*) are printed in place of the value.
- Non-integer values are rounded to the nearest integer.
- When the number is negative, one number sign is used for the minus sign.

Decimal Field

- Up to 14 significant digits may be specified.
- A decimal field is composed of number signs and a single decimal point. The decimal point may appear anywhere in the format field.
- The number is placed with the decimal point in the specified position.
- When the integer part of the value is longer than the integer part of the format, asterisks (*) are printed instead of the value.
- The number is rounded to the number of places specified to the right of the decimal point.
- When the number is negative, at least one number sign must precede the decimal point to be used for the minus sign.

Exponential Field

- Up to 14 significant digits may be specified.
- An exponential field consists of a decimal or integer field, which defines the mantissa, followed by 4 or 5 exponentiation symbols which define the exponent. When fewer than 4 are used, they are treated as literal characters. When more than 5 are used, the first 5 are used to define the exponential field, and the remainder are considered to be literal characters.
- The number is rounded according to the mantissa definition.

(continued)

(continued)

- When the mantissa definition specifies positions to the left of the decimal point, one of these positions is always used for the sign, which is a minus sign if negative and a space if positive.

String Field

- The size of the field is limited only by the size of the string which defines the format.
- A string field is an integer, decimal, or exponential field. In addition to the number signs, the decimal point and the exponentiation symbols define character positions.
- When the string is shorter than the field, it is left-justified.
- When the string is longer than the field, asterisks(*) are printed instead of the value.

Literal Field

- The size of the field is limited only by the size of the string which defines the format.
- A literal field is composed of characters which are not format characters. However, decimal points and exponentiation symbols may also appear in literal fields.
- Literal fields appear in the printed output exactly as they appear in the format definition.

Cross Reference

DISPLAY, PRINT, USING

(continued)

IMAGE

(continued)

Examples

The following program prints two numbers per line using the IMAGE statement.

```
100 FOR COUNT=1 TO 6
110 READ A,B
120 PRINT USING 150;A,B:PAUSE
130 NEXT COUNT
140 DATA -99,-9.99,-7,-3.459,0,0,14.8,12.75,795,852,
    -984,64.7
150 IMAGE THE ANSWERS ARE ### AND ##.##
```

The following show the results with the given values.

Values		Appearance
−99	−9.99	THE ANSWERS ARE -99 AND -9.99
−7	−3.459	THE ANSWERS ARE -7 AND -3.46
0	0	THE ANSWERS ARE 0 AND .00
14.8	12.75	THE ANSWERS ARE 15 AND 12.75
795	852	THE ANSWERS ARE 795 AND *****
−984	64.7	THE ANSWERS ARE *** AND 64.70

A program similar to the one above allows the use of characters with IMAGE DEAR #####,. The following show the results with certain values.

Value	Appearance
JOHN	DEAR JOHN ,
NANCY	DEAR NANCY,
KENNETH	DEAR *****,

(continued)

IMAGE

(continued)

The program below illustrates a use of IMAGE. It reads and prints seven numbers and their total. The amounts are printed with the decimal points lined up.

```
100 IMAGE $####.##
110 IMAGE " ####.##"
```

Lines 100 and 110 set up the images. They are the same except for the dollar sign. To keep the blank space where the dollar sign was, the *string-constant* in line 110 is enclosed in quotation marks.

```
120 DATA 233.45,-147.95,8.4, 37.263,-51.299,85.2,464
130 TOTAL=0
140 FOR A=1 TO 7
150 READ AMOUNT
160 TOTAL=TOTAL+AMOUNT
170 IF A=1 THEN PRINT USING 100, AMOUNT:PAUSE ELSE PRINT
    USING 110,AMOUNT:PAUSE
```

Prints the values using the IMAGE statements.

```
180 NEXT A
190 PRINT USING "$####.##", TOTAL:PAUSE
```

Uses the format directly in the PRINT statement.

INDIC SUBPROGRAM

Format

CALL INDIC(*indicator-number* [, *indicator-state*])

Description

The INDIC subprogram turns the display indicators on or off. *Indicator-number* identifies a specific indicator and must be a numeric expression which rounds to an integer value from 0 through 17.

Indicator-state is used to turn the indicator on or off. A non-zero value turns the indicator on and a zero value turns it off. If *indicator-state* is omitted, the indicator is turned on.

Indicator-numbers 1 through 6 are available for definition in a program. They are turned off when a new program is run or the computer is reset.

The other indicators are used by the system. Changing the status of a system indicator can cause erroneous results.

The numbers assigned to the display indicators are listed below.

Value	Indicator
0	ERROR
1-6	User indicators
7	LOW
8	◀
9	SHIFT
10	CTL
11	FN
12	DEG
13	RAD
14	GRAD
15	I/O
16	UCL
17	▶

WITH KEYBOARD

INPUT

Format

INPUT [*input-prompt*;] *variable-list* [, *input-prompt*; *variable-list*] [...]

Description

The INPUT statement is used to enter data from the keyboard. When INPUT is executed, program execution is suspended until data is entered.

Input-prompt is a string expression that must be followed by a semicolon. If a string constant is used, it must be enclosed in quotes. *Input-prompt* is displayed beginning at the current cursor position as left by previous input/output statements. If *input-prompt* is omitted, a question mark followed by a space is used for the prompt.

Following the prompt, the flashing cursor is displayed. If the resultant cursor position is greater than 31, the display is cleared and the cursor position is set to column 1 prior to displaying the prompt. When *input-prompt* is greater than 30 characters, it is truncated to 30 characters.

Variable-list is a list of variables separated by commas. The variables may be numeric or string, subscripted or unsubscripted. When more than one variable follows *input-prompt*, the prompt is displayed for the first variable only. Thereafter, the question mark prompt is used until another *input-prompt* is encountered. Each value is assigned to the corresponding variable name before the computer prompts for the next value.

When entering numeric variables, a numeric expression can be entered instead of a numeric constant. The expression is evaluated and the result is assigned to the variable. When entering string variables, leading and trailing spaces are ignored. Thus, if a string value includes commas, leading spaces, or trailing spaces, it must be enclosed in quotes. A quotation mark within a quoted string is represented by two quotation marks.

(continued)

INPUT WITH KEYBOARD

(continued)

If **[SHIFT] [ENTER]** is pressed during data entry, the input is ignored and the value of the variable remains unchanged. Execution proceeds to the next prompt or variable or to the next statement if the INPUT statement is completed.

If an error occurs during data entry, a descriptive error message is displayed. After the **[ENTER]** or **[CLR]** key is pressed, the INPUT statement reprompts and the data can be entered in the correct form.

When data is entered, the following validations are made.

- If more than one value at a time is entered, the message `Illegal syntax` is displayed and the data must be reentered one at a time.
- If a string constant is entered for a numeric variable, the message `String-number mismatch` is displayed and a numeric value must be entered.
- If a number whose absolute value is greater than 9.9999999999999E+127 is entered, the message `Overflow` is displayed and the value must be reentered.
- If a number whose absolute value is less than 1E−128 is entered, the value is replaced with 0 and no message is displayed.

Note: When an INPUT statement is waiting for data, **[CLR]** clears only the input field, **[CTL]** ↑ (home) and **[CTL]** ← (back tab) move the cursor to the beginning of the input field, and **[CTL]** → (right arrow) has no effect.

(continued)

(continued)

Cross Reference

ACCEPT, INPUT (with files), LINPUT

Examples

```
100 INPUT X
```

Causes the computer to display the question-mark prompt and wait for an input value. When **[ENTER]** is pressed, the entered value is stored in the variable X.

```
100 INPUT X$,Y,"ENTER Z";Z(A)
```

Causes the computer to display the question-mark prompt and wait for an input value for X$. When **[ENTER]** is pressed, the entered value is assigned to X$. The question-mark prompt is again displayed and the computer waits for a value to be entered for Y. Then `ENTER Z` is displayed and the computer waits for an input value for Z(A). The subscript is evaluated for Z(A) before the data value is stored.

INPUT

WITH FILES

Format

INPUT #*file-number* [, REC *numeric-expression*], *variable-list*

Description

The INPUT statement is used to read data from files that have been opened in INPUT or UPDATE mode. Each variable in *variable-list* is assigned a value from the file.

File-number is a number from 0 through 255 that refers to an open file or device. File number 0 refers to the keyboard and display and is always open. See INPUT (with keyboard). *File-number* is rounded to the nearest integer.

Variable-list is a list of variables separated by commas. The variables may be numeric or string, subscripted or unsubscripted. The data values in the current record are assigned to the variables in the list. If the current record does not contain enough data, another record is read. Successive records are read until each of the variables is assigned a value or the end-of-file is encountered.

The computer interprets data differently when reading DISPLAY and INTERNAL type data. See "Using External Devices" in chapter 4.

Display-type data has the same form as data entered from the keyboard. The values in each record are separated by commas. Leading and trailing spaces are ignored unless they are part of a string value enclosed in quotation marks. A quotation mark within a quoted string is represented by two quotation marks. When the INPUT statement encounters two adjacent commas, a null string is assigned to the variable. Each item is checked to ensure that numeric values are placed in numeric variables and string values in string variables.

Internal-type data is in binary format, the format used internally during execution. Each value is preceded by its length. The INPUT statement uses the lengths to separate and assign the values to the variables. The only validation performed by the INPUT statement is to ensure that numeric data is from 2 to 8 bytes long.

(continued)

(continued)

When an INPUT statement terminates, any remaining data values in the current record are ignored. The next INPUT statement which accesses the file reads another record. However, when *variable-list* ends with a comma, the input is left pending. That is, the remaining values in the current record are maintained. The next INPUT statement which accesses the file assigns the next available data value.

If pending input data exists when a PRINT, RESTORE, or CLOSE statement accesses the file, the pending data is discarded. If pending output data exists when an INPUT statement is encountered, the pending data is output before the INPUT statement is executed.

REC *numeric-expression* is used when *file-number* refers to a relative record file. *Numeric-expression* specifies the record to be read from the file. The first record of a file is record zero. See "Using External Devices" in chapter 4 and refer to individual peripheral manuals for information about relative record files and the use of the REC clause.

Cross Reference

CLOSE, INPUT, OPEN, PRINT, RESTORE

(continued)

INPUT WITH FILES

(continued)

Examples

```
100 INPUT #1,X$
```

Stores in X$ the next value available in the file that was opened as #1.

```
250 INPUT #23,X,A,LL$
```

Stores in X, A, and LL$ the next three values from the file that was opened as #23.

```
320 INPUT #3,A,B,C,
```

Stores in A, B, and C the next three values from the file that was opened as #3. The comma after C creates a pending input condition.

The following program formats the tape in the *Wafertape* peripheral (thereby destroying any data that was previously on the tape), opens it in update mode, and prints five values to the file MYFILE on the tape. The values are then reread and displayed.

```
100 FORMAT 1
110 OPEN #1,"1.MYFILE",INTERNAL, UPDATE
120 FOR A=1 TO 5
130 READ DATAOUT
140 PRINT #1,DATAOUT
```

Lines 120 through 140 read five records from the DATA statement and write them to file #1.

```
150 PRINT DATAOUT;"IS WRITTEN TO FILE #1.":PAUSE 1.5
160 NEXT A
170 RESTORE #1
180 FOR B=1 TO 5
190 INPUT #1,DATAIN
200 PRINT DATAIN;"IS RECORD #";B:PAUSE 1.5
210 NEXT B
```

Lines 180 through 210 read the five records that were written on file #1 and then display their values.

```
220 CLOSE #1, DELETE
```

Deletes the file.

```
230 DATA 15,30,72,36,94
```

INT

Format

INT(*numeric-expression*)

Description

The INT function returns the largest integer less than or equal to *numeric-expression*.

Examples

```
250 P=INT(3.999999999)
```

Sets P equal to 3.

```
470 DISPLAY AT(7),INT(4.0):PAUSE
```

Displays 4 in column 8.

```
610 K=INT(-3.0000001)
```

Sets K equal to −4.

INTRND

Format

INTRND(*numeric-expression*)

Description

The INTRND function returns an integer random number between 1 and the rounded value of *numeric-expression*. The message `Bad argument` is displayed if *numeric-expression* rounds to a value less than one.

This function is equivalent to the expression INT(RND*INT(X+.5))+1.

Examples

```
170 A=INTRND(5*EXP(2))
```

Sets A equal to a random integer value between 1 and 37.

```
330 PRINT INTRND(53):PAUSE
```

Prints a random integer value between 1 and 53.

SUBPROGRAM IO

Format

$$\text{CALL IO}\left\{\begin{array}{l}(\textit{device, command}\ [,\ \textit{status-variable}]\ )\\(\textit{string-variable}\ [,\ \textit{status-variable}]\ )\end{array}\right\}$$

Description

The IQ subprogram performs special control operations which are not available in CC-40 BASIC, but may be supported by some peripherals. Proper use of this subprogram requires knowledge of input/output (I/O) data structures and specific peripheral capabilities. Refer to the peripheral manuals for examples on the use of the IO subprogram and the Editor/Assembler manual for more information.

Device is the number associated with the peripheral device and can be from 1 through 255.

Command is a numeric code that specifies the operation to be performed by the device.

String-variable contains from 2 through 12 characters which represent the data required for the I/O operation. The data passed to the IO subprogram are interpreted as binary values. The *string-variable* is always returned with 12 characters. The data length and status may be modified. The format of the string is shown below.

Field Name	Field Length	Description
device	1	peripheral device code
command	1	operation command code
file number	1	file number as assigned in BASIC
record number	2	record number within a file
buffer length	2	size of the buffer for data received from the peripheral
data length	2	number of characters to be sent to the peripheral
status	1	status code returned by the device
buffer pointer	2	highest address of the buffer

(continued)

IO

SUBPROGRAM

(continued)

Specific requirements for this data are given in the peripherals manuals and the Editor/Assembler manual.

Status-variable is a numeric variable in which information regarding the result of the operation is stored. If no I/O error occurred, *status-variable* is zero. If an I/O error occurred, *status-variable* contains the corresponding error code. The inclusion of a *status-variable* affects the computer's response to the occurrence of an I/O error. If an I/O error occurs when *status-variable* is given, no error message is displayed and the error cannot be handled by ON ERROR. If an error occurs when *status-variable* is omitted, the message is displayed or the error can be handled by ON ERROR.

Cross Reference

ON ERROR

Example

```
140 CALL IO(1,1)
```

Closes device 1. (A command code of 1 is a CLOSE operation.)

SUBPROGRAM

KEY

Format

CALL KEY(*return-variable, status-variable*)

Description

The KEY subprogram assigns the ASCII code of a key pressed from the keyboard to *return-variable*. If no key is pressed, *return-variable* is set equal to 255. See appendix D for a list of the ASCII codes.

Status-variable is used to store a value which represents the status of the key pressed. A value of 1 means a new key was pressed since the last CALL KEY was executed. A value of −1 means the same key was pressed as was returned in the previous CALL KEY. A value of 0 means no key was pressed.

Example

```
340 CALL KEY(K,S)
350 IF S=0 THEN 340
360 PRINT K;CHR$(K)
370 PAUSE
```

Returns in K the ASCII code of any key pressed and in S a value indicating the status of the key pressed.

KEY$

Format

KEY$

Description

The KEY$ function halts program execution until a single key is pressed. When a key is pressed, execution of the program continues immediately and KEY$ returns a one character string that corresponds to the key pressed. Refer to appendix D for a list of the ASCII character codes.

If **[BREAK]** is pressed while KEY$ is waiting for a response, the break occurs as usual.

Example

The following program continues if Y is pressed and stops if N is pressed.

```
100 PRINT "Press Y to continue, N to stop"
110 A$=KEY$
120 IF A$="Y" OR A$="y" THEN 140
130 IF A$="N" OR A$="n" THEN 150 ELSE 110
140 PRINT "Continue":PAUSE 1.5 :GOTO 100
150 PRINT "Stop":PAUSE
```

LEN

Format

LEN(*string-expression*)

Description

The LEN function returns the number of characters in *string-expression*. A space counts as a character.

Examples

```
170 PRINT LEN("ABCDE"):PAUSE
```

Prints 5.

```
230 X=LEN("THIS IS A SENTENCE.")
```

Sets X equal to 19.

```
910 DISPLAY LEN(""):PAUSE
```

Displays 0.

LET

Format

[LET] { *numeric-variable* [, *numeric-variable* ...] = *numeric-expression*
string-variable [, *string-variable* ...] = *string-expression* }

Description

The LET statement assigns the value of an expression to the specified variable(s). The computer evaluates the expression on the right and places the result into the variable(s) on the left. If more than one variable is specified, they must be separated with commas. The LET is optional, and is omitted in the examples in this manual. All subscripts on the left are evaluated before any assignments are made.

Examples

```
110 LET T=4
```

Sets T equal to 4.

```
170 X,Y,Z=12.4
```

Sets X, Y, and Z equal to 12.4.

```
200 A=3<5
```

Sets A equal to −1 since it is true that 3 is less than 5.

```
350 L$,D$,B$="B"
```

Sets L$, D$, and B$ equal to “B”.

LINPUT

Format

```
        ⎧[input-prompt;] string-variable                       ⎫
LINPUT ⎨[#file-number, [REC numeric-expression,] ]            ⎬
        ⎩   string-variable                                    ⎭
```

Description

The LINPUT statement assigns an entire input record or the remainder of a pending input record to *string-variable*. Unlike INPUT, LINPUT performs no editing on the input data. Thus, all characters including commas, leading and trailing spaces, semicolons, and quotation marks are placed into *string-variable.*

Input-prompt is a string expression that must be followed by a semicolon. If a string constant is used, it must be enclosed in quotes. *Input-prompt* is displayed beginning at the current cursor position as left by previous input/output statements. If *input-prompt* is omitted, a question mark followed by a space is used for the prompt.

Following the prompt, the flashing cursor is displayed. If the resultant cursor position is greater than 31, the display is cleared and the cursor position is set to column 1 prior to displaying the prompt. When *input-prompt* is greater than 30 characters, it is truncated to 30 characters.

LINPUT can also be used to read display-type data from a file or a device. *File-number* is the number of an open file. If the specified file has pending input, the remainder of the pending record is read. The message `Bad input data` is displayed if the record or partial record is longer than 255 characters.

The optional REC clause may be used with devices which support relative record (random access) files. *Numeric-expression* specifies the record to be accessed. Refer to the appropriate peripheral manual for more information concerning relative files.

(continued)

LINPUT

(continued)

Cross Reference

INPUT

Examples

```
300 LINPUT L$
```

Causes the computer to display the question-mark prompt and store the entered data in L$.

```
470 LINPUT "NAME: ";NM$
```

Causes the computer to display `NAME:` and store the entered data in NM$.

LIST

Format

```
      ⎧ [line-group]                  ⎫
LIST  ⎨ ["device.name"]               ⎬
      ⎩ ["device.name", line-group]   ⎭
```

Description

The LIST command is used to list program lines. If *line-group* is not included, the entire program is listed. When *line-group* is given, only those lines are listed. *Line-group* may specify any of the following line ranges.

Line-group	Effect
a single line number	Lists that line.
line number –	Lists that line and all following lines.
– line number	Lists that line and all preceding lines.
line number – line number	Lists that inclusive range of lines.

When *device.name* is given, the lines are listed to the specified device. If *device.name* is omitted, the lines are shown in the display. During a listing to the display, the lines may be edited.

To suspend a listing to a device, press and hold any key until the listing stops. Pressing the key again resumes the listing. Pressing [BREAK] terminates any listing. Pressing ↑ terminates a listing to the display.

Examples

```
LIST 100
```

Lists line 100 to the display.

```
LIST 100-200
```

Lists all lines from 100 through 200 to the display.

```
LIST "50"
```

Lists the entire program to peripheral device 50 (presumably a printer).

```
LIST "50.R=C", -200
```

Lists all lines up to and including line 200 to peripheral device 50.

LN

Format

LN(*numeric-expression*)

Description

The LN function calculates the natural logarithm of *numeric-expression*. *Numeric-expression* must be greater than zero or the error message `Bad argument` is displayed. The LN function is the inverse of the EXP function.

Cross Reference

EXP

Examples

```
710 PRINT LN(3.4):PAUSE
```

Prints the natural logarithm of 3.4, which is `1.223775432`.

```
850 X=LN(EXP(2.7))
```

Sets X equal to the natural logarithm of e raised to the 2.7 power, which equals 2.7.

```
910 S=LN(SQR(T))
```

Sets S equal to the natural logarithm of the square root of the value of T.

SUBPROGRAM

LOAD

Format

CALL LOAD("*device.filename*")

Description

The LOAD subprogram loads assembly language subprograms from an external storage device into computer memory. These subprograms are run using the CALL EXEC statement.

More than one subprogram may be loaded into memory. When space permits, assembly language subprograms may reside in memory in addition to BASIC programs and subprograms. When loaded in this manner, these subprograms are appended to the memory space reserved for system operation.

Device.filename identifies the device where the assembly language subprogram is stored and the particular file to be loaded. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. An error occurs if the LOAD subprogram determines that the contents of the specified file are not an assembly language subprogram. Refer to the appropriate peripheral manuals for the proper device code and for specific information about the form of *filename*.

Loaded subprograms remain in memory until NEW ALL is entered or the system is initialized.

Cross Reference

EXEC

Examples

```
CALL LOAD("1.MYSUBS")
```

Loads the subprogram in file MYSUBS on device 1 into memory.

```
100 INPUT "ENTER FILE NAME",A$
110 CALL LOAD ("1."&A$)
```

Loads the assembly language subprogram entered by the program user.

LOG

Format

LOG(*numeric-expression*)

Description

The LOG function calculates the common logarithm of *numeric-expression*. *Numeric-expression* must be greater than zero or the error message `Bad argument` is displayed.

Examples

```
150 PRINT LOG(3.4):PAUSE
```

Prints the common logarithm of 3.4, which is .531478917.

```
230 S=LOG(SQR(T))
```

Sets S equal to the common logarithm of the square root of the value of T.

NEW

Format

NEW [ALL]

Description

The NEW command prepares the computer for a new program by deleting the program and variables currently in memory. All open files are closed.

The NEW ALL command deletes the current program and variables in memory, clears the user-assigned strings and assembly language subprograms, cancels any expansion of memory implemented by CALL ADDMEM, clears all display indicators, sets the angle mode to RAD, and closes all open files.

NEXT

Format

NEXT [*control-variable*]

Description

The NEXT statement is always paired with a FOR TO STEP statement for construction of a loop. If *control-variable* is given, it must be the same as *control-variable* in the FOR TO STEP statement. If *control-variable* is omitted, NEXT is paired with the most recent, unmatched FOR TO STEP statement. It is good programming practice to include *control-variable*.

When FOR TO STEP...NEXT loops are nested, the NEXT statement for the inside loop must appear before the NEXT statement for the outside loop.

See FOR TO STEP for a description of the looping process.

Cross Reference

FOR TO STEP

Example

The program below illustrates a use of the NEXT statement. The values printed are 30 and -2.

```
100 TOTAL=0
110 FOR COUNT=10 TO 0 STEP -2
120 TOTAL=TOTAL+COUNT
130 NEXT COUNT
140 PRINT TOTAL;COUNT:PAUSE
```

NUMBER

Format

NUMBER [*initial-line*] [, *increment*]

Description

The NUMBER (or NUM) command generates sequenced line numbers. These line numbers are displayed with a trailing space for convenience when entering program lines. All that needs to be typed in are the statement(s). After **[ENTER]** is pressed, the line is stored in memory and the next line number is displayed.

If *initial-line* and *increment* are not specified, the line numbers start at 100 and increase in increments of 10. Otherwise, lines are numbered according to the *initial-line* and *increment* specified. If a line already exists, that line is displayed and may then be replaced or changed using the edit functions. If the line number is altered, the sequence of generated line numbers continues from the new line number.

To terminate the numbering process, press **[ENTER]** when a line comes up with no statements on it or press **[BREAK]** when any line is displayed.

Cross Reference

RENUMBER

Examples

```
NUM 110
```

Instructs the computer to number starting at 110 with increments of 10.

```
NUM 105,5
```

Instructs the computer to number starting at line 105 with increments of 5.

NUMERIC

Format

NUMERIC(*string-expression*)

Description

The NUMERIC function tests whether *string-expression* is a valid representation of a numeric constant. NUMERIC returns a value of −1 (true) if *string-expression* is a valid numeric constant, and 0 (false) if *string-expression* is not a valid numeric constant.

Leading and trailing blanks in *string-expression* are ignored. NUMERIC can be used to test if the VAL function will work correctly on a string which is meant to represent a number.

Cross Reference

VAL

Example

The following program segment determines if an entry from the keyboard is a valid numeric constant. If it is not, an error message is displayed until data is reentered. If the data is a numeric constant, it is stored in variable A.

```
100 LINPUT "ENTER VALUE: ";A$
110 IF NOT NUMERIC(A$) THEN LINPUT "ERROR, REENTER: ";
    A$:GOTO 110
120 A=VAL(A$)
```

OLD

Format

OLD "*device.filename*"

Description

The OLD command loads a program from an external device into memory. OLD closes all open files and removes the program currently in memory before loading the program. A BASIC program can be stored on *device.filename* with the SAVE command.

Device.filename identifies the device where the program is stored and the name of the file. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. Refer to the peripheral manuals for the device code for each peripheral device and for specific information about the form of *filename.*

Note: If *filename* specifies a data file rather than a program file, it may be necessary to press the reset key.

Cross Reference

SAVE

Example

```
OLD "1.MYPROG"
```

Loads the program MYPROG into the computer's memory from peripheral device 1.

ON BREAK

Format

```
           ┌STOP ┐
ON BREAK  ─┤NEXT ├─
           └ERROR┘
```

Description

The ON BREAK statement determines the action taken when a breakpoint occurs. After the ON BREAK statement is executed, breakpoints are handled according to the option selected.

ON BREAK STOP restores the normal function of BREAK, which is to halt program execution and display the standard breakpoint message. This option is set when a program is run.

ON BREAK NEXT causes breakpoints to be ignored. When a breakpoint that immediately precedes a line number is encountered, the breakpoint is ignored and the program line is executed. The **[BREAK]** key is also ignored. However, a BREAK statement that does not contain a *line-number-list* halts the program even though ON BREAK NEXT is in effect. ON BREAK NEXT can be used to ignore breakpoints which you have specified in a program for debugging purposes. Note: Since the **[BREAK]** key is ignored, the reset button must be pressed to stop a program that does not stop normally.

ON BREAK ERROR causes breakpoints to be treated as errors, which allows the ON ERROR statement to be used to process breakpoints. See ON ERROR for more information.

The ON BREAK statement remains in effect until another ON BREAK statement changes it. When a subprogram ends, the ON BREAK status in effect when the subprogram was called is again in effect.

Cross Reference

BREAK, ON ERROR

(continued)

ON BREAK

(continued)

Example

The program below illustrates the use of ON BREAK. When the message Break is displayed, press **[CLR]** and enter **CONTINUE.**

```
100 BREAK 140
```

Sets a breakpoint in line 140.

```
110 ON BREAK NEXT
```

Sets breakpoint handling to ignore breakpoints.

```
120 BREAK
```

A breakpoint occurs in line 120 in spite of line 110. Press **[CLR]** and **CONTINUE.**

```
130 FOR A=1 TO 500
140 PRINT "(BREAK) IS DISABLED"
150 NEXT A
```

The **[BREAK]** key does not work while lines 130 through 150 are being executed.

```
160 ON BREAK STOP
```

Restores the normal use of **[BREAK]**.

```
170 FOR A=1 TO 500
180 PRINT "NOW (BREAK) WORKS"
190 NEXT A
```

The **[BREAK]** key again works while lines 170 through 190 are being executed.

ON ERROR

Format

$$\text{ON ERROR} \left\{ \begin{array}{l} \text{STOP} \\ \textit{line-number} \end{array} \right\}$$

Description

The ON ERROR statement determines the action taken when an error occurs during the execution of a program. After the ON ERROR statement is executed, any errors that occur are handled according to the option selected.

ON ERROR STOP restores the normal way of handling errors which is to halt program execution and print a descriptive error message. This option is set when a program is run.

ON ERROR *line-number* transfers control to the specified line when an error occurs. *Line-number* must be the beginning of an error-processing subroutine. Once an error has occurred and control has been transferred, error handling reverts to ON ERROR STOP. If the ON BREAK ERROR option was selected, it is changed to ON BREAK NEXT. For an error-processing subroutine to handle any new errors, an ON ERROR *line-number* must be executed again.

The ON ERROR statement remains in effect until another ON ERROR statement changes it. If a subprogram ends, and no errors occurred while the subprogram was executing, the ON ERROR status in effect when the subprogram was called is again in effect. If an error occurred in a subprogram, any changes in the error handling status made by the error handler is in effect when the subprogram ends.

The main program and subprograms can share the same error-processing subroutine. Subroutines called by GOSUB cannot be shared.

Cross Reference

ON BREAK, ON WARNING, RETURN (with ON ERROR)

(continued)

ON ERROR

(continued)

Example

The program below illustrates the use of ON ERROR.

```
100 ON ERROR 150
```

Causes any error to pass control to line 150.

```
110 X$="A"
120 X=VAL(X$)
```

Causes an error.

```
130 PRINT X;"SQUARED IS";X*X:PAUSE 2
140 STOP
150 REM ERROR SUBROUTINE
160 ON ERROR 220
```

Causes the *next* error to pass control to line 220.

```
170 CALL ERR(CODE,TYPE,FILE,LINE)
```

Determines the error using CALL ERR.

```
180 IF LINE< >120 THEN RETURN 220
```

Transfers control to line 220 if the error is not in the expected line.

```
190 IF CODE< >29 THEN RETURN 220
```

Transfers control to line 220 if the error is not the one expected.

```
200 X$="5"
```

Changes the value of X$ to an acceptable value.

```
210 RETURN
```

Returns control to the line in which the error occurred.

```
220 REM UNKNOWN ERROR
230 PRINT "ERROR";CODE;" IN LINE";LINE:PAUSE
```

Reports the nature of the unexpected error and the program stops.

ON GOSUB

Format

ON *numeric-expression* GOSUB *line-number1* [, *line-number2* ...]

Description

The ON GOSUB statement determines which subroutine to execute by evaluating *numeric-expression*. If the value of *numeric-expression* is 1, the subroutine starting at *line-number1* is executed; if 2, the subroutine starting at *line-number2* is executed, and so forth. Each line number must be the first statement of a subroutine. If *numeric-expression* is 0, negative, or larger than the list of line numbers, the error message `Bad value` is displayed. If *numeric-expression* is a decimal number, it is rounded.

After the RETURN statement of the subroutine is executed, control returns to the statement following ON GOSUB. ON GOSUB may not be used to transfer control into or out of a subprogram.

Cross Reference

GOSUB, RETURN (with GOSUB)

Examples

```
140 ON X GOSUB 1000,2000,300
```

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
240 ON P-4 GOSUB 200,250,300,800,170
```

Transfers control to 200 if P−4 is 1 (P is 5), 250 if P−4 is 2, 300 if P−4 is 3, 800 if P−4 is 4, and 170 if P−4 is 5.

ON GOTO

Format

ON *numerie-expression* GOTO *line-number1* [, *line-number2* ...]

Description

The ON GOTO statement determines where to transfer control by evaluating *numeric-expression*. If the value of *numeric-expression* is 1, control is transferred to *line-number1*; if 2, control is transferred to *line-number2*, and so forth. If *numeric-expression* is 0, negative, or greater than the list of line numbers, the error message `Bad value` is displayed. If *numeric-expression* is a decimal number, it is rounded.

ON GOTO may not be used to transfer control into or out of a subprogram.

Cross Reference

GOTO

Examples

```
130 ON X GOTO 1000,2000,300
```

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3. The equivalent statement using an IF THEN ELSE statement is `130 IF X=1 THEN 1000 ELSE IF X=2 THEN 2000 ELSE IF X=3 THEN 300 ELSE PRINT "Bad value": PAUSE:STOP`, which is more than 80 characters.

```
210 ON P-4 GOTO 200,250,300,800,170
```

Transfers control to 200 if P−4 is 1 (P is 5), 250 if P−4 is 2, 300 if P−4 is 3, 800 if P−4 is 4, and 170 if P−4 is 5.

ON WARNING

Format

```
              ⎧PRINT ⎫
ON WARNING   ⎨ NEXT  ⎬
              ⎩ERROR ⎭
```

Description

The ON WARNING statement determines the action taken when a warning occurs during the execution of a program. After the ON WARNING statement is executed, any warning is handled according to the ON WARNING option selected.

ON WARNING PRINT restores the normal use of warnings which is to print a descriptive warning message and continue program execution after the **[ENTER]** or **[CLR]** key is pressed. This option is selected when a program is run.

ON WARNING NEXT causes the program to continue execution without printing any message.

ON WARNING ERROR causes the occurrence of a warning to be treated as an error, allowing effective handling of warnings with ON ERROR statements.

The ON WARNING statement remains in effect until another ON WARNING statement changes it. When a subprogram ends, the ON WARNING status in effect when the subprogram was called is again in effect.

Cross Reference

ON ERROR

(continued)

ON WARNING

(continued)

Example

The program below illustrates the use of ON WARNING.

```
100 ON WARNING NEXT
```

Sets warning handling to go to the next statement.

```
110 PRINT 110,5/0:PAUSE
```

Prints the result without any message.

```
120 ON WARNING PRINT
```

Sets warning handling to the normal option, which is to print a message and allow execution to continue when a warning occurs.

```
130 PRINT 130,5/0:PAUSE
```

Prints the warning. When **[ENTER]** or **[CLR]** is pressed, prints 130 followed by the value of 5/0.

```
140 ON WARNING ERROR
```

Sets warning handling to treat warnings as errors.

```
150 PRINT 150,5/0:PAUSE
```

Prints the warning message and treats the warning as an error.

```
160 PRINT 160:PAUSE
```

Not executed because execution stops in line 150.

OPEN

Format

OPEN #*file-number*, "*device.filename*" [, *file-organization*] [, *file-type*] [, *open-mode*] [, *record-length*]

Description

The OPEN statement enables a BASIC program to use data files and peripheral devices by providing a link between *file-number* and a file or device. In setting up this link, the OPEN statement specifies how the file or device can be used (for input or output) and how the file is organized. The OPEN statement must be executed before any BASIC statement in a program attempts to use a file or device requiring a file number.

If an OPEN statement references a file that already exists, the attributes in the OPEN statement must be the same as the attributes of the file.

File-number is a number from 1 through 255 that the OPEN statement associates with a file or device. This *file-number* is used by all the input/output statements that access the file or device. File number 0 is the keyboard and display of the computer. It cannot be used for other files and is always open. If *file-number* specifies a file that is already open, an error occurs. *File-number* is rounded to the nearest integer.

Device.filename is an actual peripheral device number and other device dependent information. *Device.filename* may be a string expression. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* supplies information to the peripheral device for the OPEN statement. For example, with an external storage device, *filename* specifies the name of the file. With other devices, *filename* specifies options such as parity, data rate, etc. Refer to the peripheral manuals for the device code for each peripheral device and for specific information about the form of *filename*.

(continued)

OPEN

(continued)

The file attributes listed below may be in any order or may be omitted. When an attribute is omitted, defaults are used.

File-organization specifies either a sequential or a relative (random access) file. Records in a sequential file are read or written in sequence from beginning to end. Records in a RELATIVE (or random access) file can be read or written in any record order, including sequentially. Omit *file-organization* for sequential files or specify RELATIVE for random access files.

File-type may be either DISPLAY or INTERNAL. DISPLAY specifies that the data is written in ASCII format. INTERNAL specifies that the data is written in binary format. Binary records take up less space, are processed more quickly by the computer, and are more efficient for recording data on external storage devices. However, if the information is going to be printed or displayed for people to read, DISPLAY format should be used. If *file-type* is omitted, DISPLAY is assumed.

Open-mode instructs the computer to process the file in UPDATE, INPUT, OUTPUT, or APPEND mode. UPDATE specifies that data may be both read from and written to the file. INPUT specifies that data may only be read from the file. OUTPUT specifies that data may only be written to the file. APPEND specifies that data may only be written at the end of the file. If *open-mode* is omitted, UPDATE is assumed.

Note that if a file already exists on external storage, specifying OUTPUT mode results in new data being written over the existing data.

Record-length consists of the word VARIABLE followed by a numeric expression that specifies the maximum record length for the file. The maximum allowable record is dependent on the device used. If record length is omitted, the peripheral device specifies a default *record-length*.

(continued)

OPEN

(continued)

Cross Reference

CLOSE, INPUT, LINPUT, PRINT, RESTORE (Also see chapter 4.)

Examples

```
100 OPEN #23,"1.X",INTERNAL,UPDATE
```

Opens the file named "X" on peripheral device 1 and enables any input/output statement to access the file by using the number 23. The type of the file is INTERNAL. Since the file is opened in UPDATE mode, data can be both read from and written to the file.

```
150 OPEN #243,A$&".ABC",INTERNAL
```

If A$ equals "1", opens a file on device 1 with a name of ABC. The file type is INTERNAL, UPDATE mode is assumed, and the device specifies the default record length.

PAUSE

Format

PAUSE $\left\{\begin{array}{l}[\textit{numeric-expression}]\\ [\text{ALL}]\end{array}\right\}$

Description

The PAUSE statement suspends program execution either for a specified number of seconds or until the **[CLR]** or **[ENTER]** key is pressed. If *numeric-expression* is omitted, the underline cursor is displayed in column one to indicate an indefinite pause is occurring. The cursor control keys can then be used to view the contents of the 80-column line. Execution continues when either **[ENTER]** or **[CLR]** is pressed.

If *numeric-expression* is given, PAUSE suspends program execution for the number of seconds in the absolute value of *numeric-expression.* If *numeric-expression* is positive, the timed pause can be overridden by pressing **[ENTER]** or **[CLR]**. If negative, the timed pause cannot be overridden. The effective resolution is approximately one tenth of a second. If *numeric-expression* is less than .1, the program does not pause. During a timed pause, the cursor is not displayed and the display cannot be scrolled.

The PAUSE ALL statement suspends program execution each time a complete output line is sent to the display. Execution continues when the **[CLR]** or **[ENTER]** key is pressed. PAUSE ALL remains in effect until a timed PAUSE of length zero is executed.

PAUSE ALL remains in effect when a subprogram is called. If PAUSE ALL is modified in a subprogram, it is again in effect when the subprogram ends.

Cross Reference

DISPLAY, PRINT

(continued)

PAUSE

(continued)

Examples

```
120 PAUSE 2.2
```

Halts execution for 2.2 seconds or until the **[CLR]** or **[ENTER]** key is pressed.

```
190 PAUSE
```

Halts execution until the **[CLR]** or **[ENTER]** key is pressed.

The following program changes degrees Fahrenheit to degrees Celsius.

```
100 PRINT "ENTER DEG: ";
```

Prints the prompt `ENTER DEG:` . The pending print, created by the semicolon at the end of the PRINT statement, causes the prompt to be displayed until data is entered.

```
110 ACCEPT DG
120 PRINT DG;"DEG =";(DG-32)*5/9;"DEGREES C":PAUSE
```

Prints the answer. The PAUSE statement that follows the PRINT statement causes the answer to be displayed until the **[ENTER]** or **[CLR]** key is pressed.

```
130 GOTO 100
```

SUBPROGRAM PEEK

Format

CALL PEEK(*address, numeric-variable1* [, *numeric-variable2* ...])

Description

The PEEK subprogram is used to read the contents of memory locations. Starting at the memory location specified by *address*, the value of that byte of memory is assigned to *numeric-variable1*, the value of the next byte to *numeric-variable2*, and so forth. The number of variables listed determines how many bytes are read.

Address must be a numeric expression from 0 to 65535. The values assigned to the variables are in the range 0 through 255.

Cross Reference

POKE

Example

```
100 CALL PEEK(2096,X1,X2,X3,X4)
```

Returns the values in locations 2096, 2097, 2098, and 2099 In variables X1, X2, X3, and X4, respectively.

PI

Format

PI

Description

The PI function returns the value of π as 3.14159265359.

Example

```
130 VOLUME=4/3*PI*R^3
```

Sets VOLUME equal to four thirds times PI times the radius cubed, which is the volume of a sphere with a radius of R.

SUBPROGRAM

POKE

Format

CALL POKE(*address*, *byte1* [, *byte2* ...])

Description

The POKE subprogram is used to write data into memory locations. The value of *byte1* is stored in the memory location specified by *address*, the value of *byte2* is stored in the next memory location, and so forth.

The value of each data byte can be from 0 through 255. If the value is greater than 255, it is repeatedly reduced by 256 until it is from 0 through 255. Using a byte value greater than 32767 causes an error.

Indiscriminate use of this statement may destroy the program currently in memory and require that the computer be reset to continue.

Cross Reference

PEEK

Example

```
200 CALL POKE(ADDR,162,10,17)
```

Places the values 162, 10, and 17 in the locations ADDR, ADDR+1, and ADDR+2 respectively.

POS

Format

POS(*string1*, *string2*, *numeric-expression*)

Description

The POS function returns the position of the first occurrence of *string2* in *string1*. The search begins at the position specified by *numeric-expression*. If no match is found, the function returns a value of zero.

Examples

```
110 X=POS("PAN","A",1)
```

Sets X equal to 2 because A is the second letter in PAN.

```
140 Y=POS("APAN","A",2)
```

Sets Y equal to 3 because the A in the third position in APAN is the first occurrence of A in the portion of APAN that was searched.

```
170 Z=POS("PAN","A",3)
```

Sets Z equal to 0 because A was not in the part of PAN that was searched.

```
290 R=POS("PABNAN","AN",1)
```

Sets R equal to 5 because the first occurrence of AN starts with the A in the fifth position in PABNAN.

WITH DISPLAY

PRINT

Format

PRINT [USING *line-number*
string-expression ,] [*print-list*]

Description

The PRINT statement may be used to format and write data to the display. USING may be used to specify a format for the items in *print-list*. Refer to IMAGE and USING for a description of format definition and its effect upon the PRINT statement. If *print-list* is omitted, the PRINT statement clears the display.

Print-list consists of print items and print separators. Print items are numeric and string expressions that are displayed and TAB functions that control print positioning. Print separators are commas or semicolons that indicate the position of print items in the display.

Print Items

During execution of a PRINT statement, the values of the expressions in *print-list* are displayed in order from left to right in the positions determined by the print separators and TAB functions.

- *String expressions* are evaluated to produce a string result. String constants must be enclosed in quotation marks. Blank spaces are not inserted before or after a string. To print a blank space before or after a string, include it in the string or insert it separately with quotes.
- *Numeric-expressions* are evaluated and displayed with a trailing space. Positive values are printed with a leading space (instead of a plus sign) and negative numbers are printed with a leading minus sign.
- The TAB function specifies the starting position in the print line for the next item in the *print-list*. See TAB for more information.

(continued)

PRINT

(continued)

Print Separators

You must place at least one print separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon prints the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma prints the next print item at the beginning of the next print field. The print fields are 15 characters long and are located at columns 1, 16, 31, 46, 61, and 76 for an 80-column line. If the current column position is past the start of the last print field, the comma causes the next printed item to be displayed in the next line.

If a print item is longer than the remainder of the current line, it is displayed at the start of the next line. If a numeric print item fits on the current line without its trailing space, it is printed on the current line. If a print item is longer than 80 characters, the first 80 characters are printed on one line and the remaining characters are printed on successive lines, 80 characters at a time.

Pending Prints

If the *print-list* is not followed by a comma or a semicolon, the remainder of the 80-column line is cleared. Therefore, the next input/output statement must begin a new line.

Using a comma or a semicolon after *print-list* creates a pending print which causes the remainder of the line not to be cleared. Instead the computer spaces over to the start of the next field if a comma ended the PRINT statement, or does not space at all if a semicolon ended the statement. The next I/O statement displays or accepts information beginning at the current column position unless the statement changes the position.

A pending print can be used to create an input prompt for the ACCEPT or INPUT (with display) statement. The next INPUT statement places its prompt after the pending print. See ACCEPT and INPUT (with display) for more information.

(continued)

(continued)

Numeric Formats

Numbers are printed in either normal decimal form or scientific notation. Scientific notation is used when more significant digits can be shown.

When a number is printed in normal decimal form, the following conventions are observed.

- Integers are printed without a decimal point.
- Non-integers are printed with a decimal point. Trailing zeros in the fractional part are omitted. If the number has more than ten significant digits, the value is rounded to ten digits.
- A number whose absolute value is less than one is printed without a zero to the left of the decimal point.

A number printed in scientific notation is in the following form.

mantissa E exponent

When a number is printed in scientific notation, the following conventions are observed.

- The mantissa is printed with 7 or fewer digits with one digit always to the left of the decimal.
- Trailing zeros are omitted in the fractional part of the mantissa.
- The exponent is displayed with a plus or minus sign followed by a two or three digit exponent.
- When the exponent is two digits, the mantissa is limited to seven digits. When the exponent is three digits, the mantissa is limited to six digits. When necessary, the mantissa is rounded to the appropriate number of digits.

Cross Reference

ACCEPT, DISPLAY, IMAGE, INPUT, PAUSE, TAB, USING

(continued)

PRINT WITH DISPLAY

(continued)

Examples

```
100 PRINT
```

Prints a blank line.

```
210 PRINT "THE ANSWER IS";ANSWER:PAUSE
```

Prints THE ANSWER IS immediately followed by the value of ANSWER.

```
320 PRINT X,Y/2:PAUSE
```

Prints the value of X and in the next field the value of Y/2.

```
450 PRINT "NAME: ";
460 ACCEPT N$
```

Prints NAME: and accepts the entry after the prompt.

WITH FILES

PRINT

Format

PRINT *#file-number* [, REC *numeric-expression*]
[, USING *line-number* / *string-expression*] [, *print-list*]

Description

The PRINT statement may be used to format and write data to a file or device. *File-number* is a number from 0 through 255 that refers to an open file or device. The file must have been opened in OUTPUT, UPDATE, or APPEND mode. *File-number* 0 refers to the display, which is always open. *File-number* is rounded to the nearest integer.

REC *numeric-expression* may appear only when *file-number* refers to a relative record file. Refer to chapter 4 and the individual peripheral manuals for information about relative record files and the proper use of REC. *Numeric-expression* is evaluated to designate the specific record number of the file to which to write.

USING may be used to specify an exact format for a display-type file. Refer to the IMAGE and USING sections for a description of format definition and its effect upon the PRINT statement. Including USING in a reference to an internal-type data file results in an error.

Print-list consists of print items and print separators. Print items are numeric and string expressions that are displayed and TAB functions that control print positioning. Print separators are commas or semicolons that indicate the position of print items in the display.

Print-list is interpreted in order from left to right. The form of the output depends upon the type (DISPLAY or INTERNAL) of file or device. See OPEN and chapter 4 for a description of *file-type*.

(continued)

(continued)

Display-type Files

During execution of a PRINT statement that refers to a display-type file or device, *print-list* is evaluated as follows.

- *String-expressions* are evaluated to produce a string result. String constants must be enclosed in quotation marks. Blank spaces are not inserted before or after a string. To print a blank space before or after a string, include it in the string or insert it separately with quotes.
- *Numeric-expressions* are evaluated and displayed with a trailing space. Positive values are printed with a leading space (instead of a plus sign) and negative numbers are printed with a leading minus sign.
- The TAB function specifies the starting position in the print line for the next item in *print-list*. See TAB for more information.

You must place at least one print separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon prints the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma prints the next print item at the beginning of the next print field. The print fields are 15 characters long and are located at columns 1, 16, 31, and so forth. If the current column position is past the start of the last print field, the comma causes the next printed item to be printed in the next record.

If a print item is longer than the remainder of the current record, the current record is printed and the print item is printed at the start of the next record. If a numeric print item fits in the current record without its trailing space, it is printed in the current record. If a print item is longer than the record length, it is divided into segments that are the length of the record until the last segment is the length of the record or less. The segments are then printed in successive records.

(continued)

(continued)

Internal-type Files

During execution of a PRINT statement that refers to an internal-type file or device, *print-list* is evaluated as follows.

- String expressions are evaluted and printed in the record in internal string representation.
- Numeric expressions are evaluated and printed in the record in internal numeric representation.
- The TAB function causes an error when used in printing to an internal-type file.

You must place at least one separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon prints the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma functions exactly the same as the semicolon separator.

If a print item is longer than the remainder of the current record, the current record is printed and the print item is written at the start of the next record. If a print item is longer than the record length, an error occurs.

Pending Prints

If the *print-list* ends without a comma or a semicolon, the record is immediately written to the file. The next input/output statement which accesses the file begins a new record.

Using a comma or a semicolon after *print-list* creates a pending print. If the *print-list* ends with a comma or semicolon, the current record is not written. The computer spaces over to the start of the next field if a comma ended the PRINT statement, or does not space at all if a semicolon ended the statement. The next output statement which accesses this file prints data on this same record, beginning at the current column position unless the statement changes the position.

(continued)

PRINT

WITH FILES

(continued)

When *print-list* is omitted, but there is a pending output record, the PRINT statement writes the pending record. When there is no pending record, the result depends upon the file type. If the file is display-type, the PRINT statement writes a blank (zero length) record. If the file is internal-type, an error occurs because internal-type files do not support zero length records.

Cross Reference

IMAGE, INPUT (with files), OPEN, TAB, USING

Examples

```
150 PRINT #32,A,B,C,
```

Causes the values of A, B, and C to be printed to the next record of the file that was opened as number 32. The final comma creates a pending print condition. The next PRINT statement accessing file #32 is printed to the same record as this PRINT statement.

The program below writes data to a file.

```
100 OPEN #5,"1.MYPROG",INTERNAL,UPDATE
```

Opens file number 5. MYPROG is created if it does not already exist on device number 1.

```
110 DIM A(50)
```

Dimensions an array for 51 values.

```
120 B=0
```

Initializes the summation varlable.

```
130 FOR J=1 TO 50
```

Lines 130 through 180 facilitate data input.

```
140 PRINT "ENTER VALUE";
150 ACCEPT A(J)
160 B=B+A(J)
170 PRINT #5, A(J);
```

Value of A(J) is written to the file.

```
180 NEXT J
190 PRINT #5,B
```

Value of summation variable is written to the file.

```
200 CLOSE #5
```

RAD

Format

RAD

Description

The RAD statement sets the units for angle calculations to radians. After the RAD angle setting is selected, all entered and calculated angles are measured in radians. The RAD setting is selected when NEW ALL is entered or the system is initialized.

Cross Reference

DEG, GRAD

RANDOMIZE

Format

RANDOMIZE [*numeric-expression*]

Description

The RANDOMIZE statement sets the random number generator to an unpredictable sequence.

If RANDOMIZE is followed by a *numeric-expression*, the same sequence of random numbers is produced each time the statement is executed with that value. Different values give different sequences.

Example

The program below illustrates a use of the RANDOMIZE statement. It accepts a value for *numeric-expression* and prints the first 10 random numbers obtained using the RND function. Press **[BREAK]** to stop the program.

```
100 INPUT "SEED: ";S
110 RANDOMIZE S
120 FOR A=1 TO 10:PRINT A;RND:PAUSE 1.1
130 NEXT A
140 GOTO 100
```

READ

Format

READ *variable-list*

Description

The READ statement is used with the DATA statement to assign values to variables. *Variable-list* consists of string and numeric variables, either subscripted or unsubscripted, separated by commas. The value read in the DATA statement must correspond to the type of the variable to which it is assigned in READ. Note that any number is a valid string. When two adjacent commas are encountered in the data list, a null string is read.

The READ statement begins reading from the first DATA statement in the current program or subprogram and proceeds to the next DATA statement when the current data list has been read. A single READ statement may read from more than one DATA statement, and several READ statements may read from a single DATA statement. If a READ statement does not read all of the current data list, the next READ statement begins with the first unread item in the list. An attempt to read data after all the data in the current program or subprogram has been read results in an error.

The RESTORE statement can be used to alter the order in which DATA statements are read.

READ can read data only from a DATA statement that is in the same program or subprogram as the READ statement. Each time a subprogram is called, data is read from the first DATA statement whether or not the subprogram has been attached. (See ATTACH in this chapter.)

Cross Reference

ATTACH, DATA, RESTORE

RELEASE

Format

RELEASE *sub-name1* [, *sub-name2* ...]

Description

The RELEASE statement is used to release attached subprograms. (See ATTACH in this chapter). When RELEASE is executed, the allocated memory space for the subprogram variables is released, and thus the values are destroyed.

Releasing a repeatedly used subprogram increases execution time for a program. However, the subprogram variables do not require memory space between calls to the subprogram.

A RELEASE statement may appear in the main program or in any subprogram, including a subprogram that it releases. If a *sub-name* is specified for an active subprogram, the variables are not released until the subprogram terminates. If *sub-name* specifies an assembly language program, an error occurs. If a specified *sub-name* is not attached or does not exist, that *sub-name* parameter is ignored.

Cross Reference

ATTACH

(continued)

(continued)

Example

The following program illustrates the use of the RELEASE statement.

```
100 ATTACH X
110 FOR J=1 TO 5
120 CALL X
130 NEXT J
          Prints 0 1 2 3 4 because the variable values are not
          initialized when X is called and are not destroyed when X
          is terminated.
140 RELEASE X:PRINT
          Releases subprogram X and clears the display.
150 FOR J=1 TO 5
160 CALL X
170 NEXT J
          Prints 0 0 0 0 0 because the variable values in
          subprogram X are initialized each time it is called.
180 SUB X
190 PRINT J;:PAUSE 2
200 J=J+1
210 SUBEND
```

RELMEM

Format

CALL RELMEM(*numeric-expression*)

Description

The RELMEM subprogram releases memory previously reserved by the GETMEM subprogram. The value given in *numeric-expression* must be the same address returned by GETMEM when the memory space was reserved. If the wrong value is specified for *numeric-expression*, the contents of memory, including the program, can be lost.

Cross Reference

GETMEM, PEEK, POKE

Example

The following example acquires some memory with a CALL GETMEM. POKE is used to store an assembly language program in the memory. When the subprogram is no longer needed, the memory is returned to the system with CALL RELMEM.

```
100 CALL GETMEM (50,ADDR)
110 CALL POKE (ADDR,...)
120 CALL EXEC (ADDR)
:
220 CALL RELMEM(ADDR)
```

REM

Format

REM [*character-string*]

Description

The REM statement allows you to enter explanatory remarks into your program. Remarks may give any type of information, but usually explain a section of a program. *Character-string* may include any displayable character.

Remarks are not executed, but they do take up space in memory. Any character that follows REM, including the statement separator symbol (:) is considered part of the remark. Therefore, if REM is part of a multiple statement line, it must be the last statement on the line.

The exclamation point (!) is called a tail remark symbol and may be used instead of the word REM. The exclamation point can appear as the first statement on a line or after the last statement in a multiple statement line. If the exclamation point appears after a statement, the statement separator (:) is not needed. Using the tail remark symbol saves space in the listed form of the program.

Example

```
150 REM BEGIN SUBROUTINE
```

Identifies a section beginning a subroutine.

```
270 SUBTOTAL=L+B ! Calculate subtotal
```

Identifies statements which perform a specific calculation.

RENUMBER

Format

RENUMBER [*initial-line*] [, *increment*]

Description

The RENUMBER (or REN) command changes the line numbers of a program. If no *initial-line* is provided, the renumbering starts with 100. If no *increment* is given, an *increment* of 10 is used.

REN also changes all references to line numbers so that they refer to the same lines of code as before. If a statement refers to a line number that does not exist, a warning is displayed and the line number is replaced with 32767, which is not a valid line number.

If the values entered for *initial-line* and *increment* result in the creation of line numbers larger than 32766, the error message `Bad line number` is displayed and the program is left unchanged.

Example

```
REN
```

Renumbers all lines to start with 100 and increment by 10.

RESTORE

Format

RESTORE { [*line-number*]
[#*file-number* [, REC *numeric-expression*]] }

Description

The RESTORE statement is used to control the order in which data is read from DATA statements or from a file.

RESTORE specifies that the next READ statement executed accesses the first item in the DATA statement specified by *line-number. Line-number* must be in the same program or subprogram as the RESTORE statement. If no *line-number* is given, the DATA statement with the lowest numbered line in the current program or subprogram is used. If *line-number* is not a DATA statement, the next DATA statement following it is used.

RESTORE #*file-number* positions that file to the first record. The next input/output statement that refers to *file-number* accesses the first record in the file. Any pending output data is written to the file before the RESTORE statement is executed. Any pending input data is ignored. *File-number* 0 refers to a DATA statement as described above.

REC may be used with devices which support relative record (random access) files. *Numeric-expression* specifies the record to which the random access file is positioned. The next input/output statement that refers to that file accesses that record. Refer to the peripheral manuals for information about relative files.

Note: The first record of a file is record zero.

(continued)

RESTORE

(continued)

Cross Reference

DATA, INPUT, LINPUT, PRINT, READ

Examples

```
150 RESTORE
```

Sets the next DATA statement to be read to the first DATA statement in the program.

```
200 RESTORE 130
```

Sets the next DATA statement to be read to the DATA statement at line 130 or, if line 130 is not a DATA statement, to the next DATA statement after line 130.

```
230 RESTORE #1
```

Sets file #1 to the first record in the file, which is record 0.

RETURN

WITH GOSUB

Format

RETURN

Description

RETURN used with GOSUB transfers control back to the statement following the GOSUB or ON GOSUB statement which was last executed. A subroutine may contain more than one RETURN statement.

Cross Reference

GOSUB, ON GOSUB

RETURN WITH ON ERROR

Format

$$\text{RETURN}\begin{Bmatrix}\text{[NEXT]}\\ \textit{[line-number]}\end{Bmatrix}$$

Description

RETURN ends an error-processing subroutine. An error-processing subroutine is called when an error occurs after an ON ERROR *line-number* statement has been executed. The error-processing subroutine can contain any BASIC statements, including another ON ERROR statement.

RETURN with no option transfers control to the statement in which the error occurred and the statement is executed again.

RETURN NEXT transfers control to the statement following the one in which the error occurred.

RETURN *line-number* transfers control to the line specified. The specified line must be in the same program or subprogram as the error-processing subroutine even though the error may have occurred in some other subprogram.

Cross Reference

ON ERROR

(continued)

RETURN WITH ON ERROR

(continued)

Example

The program below illustrates the use of RETURN with ON ERROR.

```
100 ON ERROR 150
        Transfers control to line 150 when an error occurs.
120 X=VAL("D")
        Causes an error, so control is transferred to line 160.
130 PRINT "Done":PAUSE 2
        Prints Done.
130 STOP
140 REM ERROR HANDLING
150 IF A>4 THEN 200
        Checks to see if the error has occurred four times and
        transfers control to 200 if it has.
160 A=A+1
        Increments the error counter by one.
170 PRINT A;"errors":PAUSE 2
        Prints the number of errors which have occurred.
180 ON ERROR 150
        Resets the error handling to transfer to line 150.
190 RETURN
        Returns to the line that caused the error and executes it
        again.
200 PRINT "Last error":PAUSE 2:RETURN NEXT
        Is executed only after the error has occurred four times.
        Prints Last error and returns to the line following the one
        that caused the error.
```

RND

Format

RND

Description

The RND function returns the next pseudo-random number in the current sequence of pseudo-random numbers. The number returned is greater than or equal to zero and less than one. Unless the RANDOMIZE statement is used to create an unpredictable sequence, RND generates the same sequence each time a program is run.

Cross Reference

INTRND, RANDOMIZE

Example

```
100 PRINT 10*RND:PAUSE
```

Prints a random number greater than or equal to 0 and less than 10.

RPT$

Format

RPT$(*string-expression*, *numeric-expression*)

Description

The RPT$ function returns a string that is *numeric-expression* repetitions of *string-expression*. If RPT$ produces a string longer than 255 characters, the excess characters are discarded and the warning message `String-truncation` is displayed.

Examples

```
100 M$=RPT$("ABCD",4)
```

Sets M$ equal to "ABCDABCDABCDABCD".

```
100 CALL CHAR(0,RPT$("0000FFFF",8))
```

Defines characters 0 through 3 with the string "0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF 0000FFFF0000FFFF0000FFFF".

```
100 PRINT USING RPT$("#",40);X$ :PAUSE
```

Prints the value of X$ using an image that consists of 40 number signs.

RUN

Format

$$\text{RUN}\left\{\begin{array}{l}[\textit{line-number}]\\ [\text{``}\textit{program-name}\text{''}]\\ [\text{``}\textit{device.filename}\text{''}]\end{array}\right\}$$

Description

The RUN statement starts execution of a program. The statement RUN entered with no options starts execution of the program currently in memory beginning with the lowest numbered line.

RUN *line-number* starts execution of the program in memory at the specified *line-number.*

RUN "*program-name*" searches the *Solid State Software*™ cartridge and starts execution of *program-name* when it is found. If *program-name* is not found or refers to a subprogram, an error occurs. A string expression may be used to specify *program-name*.

RUN "*device.filename*" deletes the program currently in memory, loads the contents of *filename* from *device* into memory, and executes it. A string expression may be used to specify *device.filename*. Note: If *filename* specifies a data file rather than a program file, it may be necessary to press the reset key.

Before a program is executed, the following process takes place.

- Variables are initialized. Numeric variables are set to zero and string variables are set to null strings.
- Certain errors, such as a FOR statement without a NEXT statement or a line reference out of range, are detected.
- All open files are closed.
- ON BREAK STOP, ON WARNING PRINT, and ON ERROR STOP are selected.
- The angle mode selected is left unchanged.

(continued)

RUN

(continued)

Examples

RUN

Causes the computer to begin execution of the program in memory, starting with the lowest numbered line.

RUN 200

Causes the computer to begin execution of the program in memory starting at line 200.

RUN "1.PRG3"

Causes the computer to load and begin execution of the program in file PRG3 on device 1.

RUN "STAT"

Executes the program STAT in the *Solid State Software* cartridge.

The program below illustrates the use of the RUN statement to execute a program from a program. A menu is created to allow the person using the program to choose what other program to run. The other programs should run this program rather than ending in the usual way, so that the menu is given again after they are finished.

```
100 PRINT "Enter 1, 2, or 3 for programs":PAUSE 2
110 PRINT "... or enter 4 to stop":PAUSE 2
120 INPUT "YOUR CHOICE: ";C
130 IF C=1 THEN RUN "1.PRG1"
140 IF C=2 THEN RUN "1.PRG2"
150 IF C=3 THEN RUN "1.PRG3"
160 IF C=4 THEN STOP
170 GOTO 100
```

SAVE

Format

SAVE "*device.filename*" [, PROTECTED]

Description

The SAVE command allows you to copy the BASIC program in memory to an external device. SAVE removes any variables from the system which are not used in the program. By using the OLD command, you can later recall the program into memory.

Device.filename identifies the device where the program is to be stored and the file name. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the file which contains the program.

When PROTECTED is specified, the program in memory is left unprotected but the copy on the external storage device is saved in protected format. A protected program cannot be listed, edited, or saved.

Cross Reference

OLD, VERIFY

Examples

```
SAVE "1.PRG1"
```

Saves the program in memory to device 1 under the name PRG1.

```
SAVE "2.PRG2",PROTECTED
```

Saves the program in memory to device 2 under the name PRG2. The program may be loaded into memory and run, but it may not be edited, listed, or resaved.

SEG$

Format

SEG$(*string-expression*, *position*, *length*)

Description

The SEG$ function returns a substring of a string. The string returned starts at *position* in *string-expression* and extends for *length* characters. If *position* is beyond the end of *string-expression*, the null string ("") is returned. If *length* extends beyond the end of *string-expression*, only the characters through the end are returned.

Examples

```
100 X$=SEG$("FIRSTNAME LASTNAME",1,9)
```

Sets X$ equal to "FIRSTNAME".

```
200 Y$=SEG$("FIRSTNAME LASTNAME",11,8)
```

Sets Y$ equal to "LASTNAME".

```
240 Z$=SEG$("FIRSTNAME LASTNAME",10,1)
```

Sets Z$ equal to " ".

```
280 PRINT SEG$(A$,B,C):PAUSE
```

Prints the substring of A$ starting at character B and extending for C characters.

SETLANG SUBPROGRAM

Format

CALL SETLANG(*numeric-expression*)

Description

The SETLANG subprogram selects the language in which system messages and errors are displayed. *Numeric-expression* is a number that is the code of a specific language. The following are the assigned language codes.

0 = English
1 = German
2 = French
3 = Italian
4 = Dutch
5 = Swedish
6 = Spanish

If *numeric-expression* is 0 or 1, all system messages and errors are displayed in English or German, respectively. If *numeric-expression* selects any other language that is supported in a *Solid State Software*™ cartridge, prompts and messages in the cartridge are displayed in the chosen language, but all system messages and errors are displayed in English.

The language code is maintained by the *Constant Memory*™ feature. Therefore, the language code setting is not altered by turning the computer on or off, and remains in effect until it is changed or the system is initialized. When the system is initialized, the language code is set to zero (English).

Cross Reference

GETLANG

SGN

Format

SGN(*numeric-expression*)

Description

The SGN function returns the mathematical signum function. If *numeric-expression* is positive, a 1 is returned. If it is zero, a 0 is returned and if it is negative, a −1 is returned.

Examples

```
140 IF SGN(A)=1 THEN 300 ELSE 400
```

Transfers control to line 300 if A is positive and to line 400 if A is zero or negative.

```
790 ON SGN(X)+2 GOTO 200,300,400
```

Transfers control to line 200 if X is negative, line 300 if X is zero, and line 400 if X is positive.

SIN

Format

SIN(*numeric-expression*)

Description

The sine function gives the trigonometric sine of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current angle mode in effect (see DEG, GRAD, and RAD). See appendix E for a description of the limits of *numeric-expression*.

Example

```
150 DEG
160 PRINT SIN(3*21.5+4):PAUSE
```

Prints .930417568.

SQR

Format

SQR(*numerlc-expression*)

Description

The SQR function returns the positive square root of *numeric-expression*. SQR(X) is equivalent to X∧(1/2). *Numeric-expression* must not be a negative number.

Examples

```
150 PRINT SQR(4):PAUSE
```

Prints 2.

```
780 X=SQR(2.57E5)
```

Sets X equal to the square root of 257,000, which is 506.9516742255.

STOP

Format

STOP

Description

The STOP statement stops program execution. It can be used interchangeably with the END statement except that it may not be placed after subprograms.

Cross Reference

END

Example

The program below illustrates the use of the STOP statement. The program adds the numbers from 1 to 100.

```
100 TOT=0
110 NUMB=1
120 TOT=TOT+NUMB
130 NUMB=NUMB+1
140 IF NUMB>100 THEN PRINT TOT:PAUSE 2:STOP
150 GOTO 120
```

STR$

Format

STR$(*numeric-expression*)

Description

The STR$ function returns the string representation of the value of *numeric-expression*. No leading or trailing spaces are included. The STR$ function is the inverse of the VAL function.

Cross Reference

LEN, VAL

Examples

```
150 NUM$=STR$(78.6)
```

Sets NUM$ equal to "78.6".

```
220 LL$=STR$(3E15)
```

Sets LL$ equal to "3.E+15".

```
330 J$=STR$(A*4)
```

Sets J$ equal to a string equal to the value obtained when A is multiplied by 4. For instance, if A is equal to −8, J$ is set equal to "−32".

SUB

Format

SUB *subprogram-name* [*(parameter-list)*]

Description

The SUB statement is the first statement in a subprogram and must be the first statement on the line. A subprogram is a group of statements separated from the main program. A subprogram is used to perform the same task in several different places without duplicating the statements in several places.

Subprograms are accessed by CALL *subprogram-name* [*(argument-list)*]. *Subprogram-name* consists of 1 to 15 characters. The first character must be an alphabetic character or an underline. The remaining characters may be alphanumeric characters or underlines. The CALL statement searches for subprograms in a specific order (see CALL for the order) and executes the first subprogram found with *subprogram-name.* If the name of one of your subprograms is the same as a built-in subprogram, the built-in subprogram is executed.

Parameter-list defines the information passed to the subprogram. A parameter may be a simple string variable, a simple numeric variable, or an array. An array is listed as a parameter by writing the array name followed by parentheses. A one-dimensional array is written as A( ), a two-dimensional array as A(,), and a three-dimensional array as A(,,).

Information is passed to the subprogram through the *argument-list* of the CALL statement. The arguments of *argument-list* and the parameters of *parameter-list* need not have the same names. However, the number and the types of arguments in *argument-list* must match the number and types of parameters in *parameter-list* of the SUB statement.

(continued)

(continued)

Information is passed to a subprogram either by reference or by value. If an argument is passed by reference, the subprogram uses the variables from the calling program. If the corresponding parameter in the subprogram is changed, the argument in the calling program is also changed. A simple variable, an element of an array, or an array listed in *argument-list* is passed by reference. Arrays are always passed by reference.

If an argument is passed by value, only the value of the argument is passed to the subprogram. If the corresponding parameter in the subprogram is changed, it does not alter the value of the argument in the calling program. Any type of expression in *argument-list* is evaluated and passed by value to the subprogram. Simple variables may be passed by value by enclosing them in parentheses.

All variables used in a subprogram other than those in *parameter-list* are local to that subprogram, so the same variable names may be used in the main program and in other subprograms. Changing the values of local variables in a program or subprogram does not affect the values of local variables in any other program or subprogram.

Any local variables in the subprogram are initialized each time the subprogram is called, unless the subprogram has been attached. Attaching a subprogram causes the values of the variables to be retained between calls until the subprogram has been released. See ATTACH and RELEASE.

A subprogram terminates when a SUBEXIT or SUBEND statement is executed. Control is returned to the statement following the CALL statement.

(continued)

SUB

(continued)

Subprograms appear after the main program. A subprogram cannot contain another subprogram. When a SUB statement is encountered in a main program, it terminates as if a STOP statement had been executed. Only remarks and END statements may appear between the SUBEND of one program and the SUB of the next subprogram.

The ON BREAK, ON WARNING, ON ERROR, and PAUSE ALL statements in effect when a CALL is executed remain in effect while the subprogram is executing. If the subprogram changes any of these statements, they are changed back when the subprogram terminates. Subprograms cannot share any subroutines except error-processing subroutines.

Cross Reference

ATTACH, CALL, ON BREAK, ON ERROR, ON WARNING, RELEASE, RETURN, SUBEND, SUBEXIT

(continued)

SUB

(continued)

Examples

`100 SUB MENU`

Marks the beginning of a subprogram. No parameters are passed or returned.

`220 SUB MENU(COUNT,CHOICE)`

Marks the beginning of a subprogram. The variables COUNT and CHOICE may be used and/or have their values changed in the subprogram and their corresponding arguments in the calling statement changed.

`330 SUB PAYCHECK(DATE,(Q),SSN,PAYRATE,TABLE(,))`

Marks the beginning of a subprogram. The variables DATE, SSN, PAYRATE, and the array TABLE with two dimensions may be used and/or have their values changed in the subprogram and their corresponding arguments in the calling statement changed. The variable Q cannot be altered by the subprogram.

SUBEND

Format

SUBEND

Description

The SUBEND statement marks the end of a subprogram. When SUBEND is executed, control is passed to the statement following the statement that called the subprogram. The SUBEND statement must always be the last statement in a subprogram and cannot be in an IF THEN ELSE statement. Only remarks and END statements may appear between a SUBEND statement and the next SUB statement.

Cross Reference

SUB, SUBEXIT

SUBEXIT

Format

SUBEXIT

Description

The SUBEXIT statement terminates execution of a subprogram. When it is executed, control is passed to the statement following the statement that called the subprogram. The SUBEXIT statement may appear as many times as needed in a subprogram.

Cross Reference

SUB, SUBEND

TAB

Format

TAB(*numeric-expression*)

Description

The TAB function is used In a PRINT or DISPLAY statement to select a specific column position for a printed item. If *numeric-expression* is less than or equal to zero, the position is set to one. If *numeric-expression* is greater than the length of a record for the device being used, then *numeric-expression* is repeatedly reduced by the record length until it is less than the record length.

If the current position is less than or equal to the specified position, the TAB function spaces over to the specified position. If the current position is greater than the specified position, the TAB function proceeds to the next record and spaces over to the specified position.

The TAB function is treated as a *print-item* and must be separated from other print items by a print separator. The print separator before TAB is evaluated before the TAB function and the print separator following TAB is evaluated after the TAB function. Normally, semicolons are used before and after TAB.

In a DISPLAY statement, the TAB function is relative to the beginning of the display field. If AT is used, the TAB function Is relative to the specified column position. If more than one line of output is displayed, subsequent lines begin in column one. Any TAB functions are then relative to column one.

If SIZE is used, the value specified in SIZE is the absolute limit of the number of characters displayed. This limit is the record length used in evaluating any TAB functions.

(continued)

TAB

(continued)

Cross Reference

DISPLAY, PRINT

Examples

```
100 PRINT TAB(12);35:PAUSE
```

Prints the number 35 starting at column 13.

```
190 PRINT 356;TAB(18);"NAME":PAUSE
```

Prints 356 at the beginning of the line and NAME starting at column 18.

```
710 DISPLAY AT(10) SIZE(20),"MGB";TAB(10);"ADDR":PAUSE
```

Prints MGB starting at column 10 and ADDR starting at column 19.

TAN

Format

TAN(*numeric-expression*)

Description

The TAN (tangent) function gives the trigonometric tangent of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current angle mode in effect (see DEG, GRAD, and RAD). See appendix E for a description of the limits of *numeric-expression*.

Example

```
250 RAD
260 PRINT TAN(20):PAUSE
```

Prints 2.237160944.

UNBREAK

Format

UNBREAK [*line-list*]

Description

The UNBREAK statement removes all breakpoints. If *line-list* is specified, only the breakpoints for those lines listed are removed.

Cross Reference

BREAK

Examples

```
UNBREAK
```

Removes all breakpoints.

```
400 UNBREAK 100,130
```

Removes the breakpoints set before lines 100 and 130.

USING

Format

USING $\left\{\begin{array}{l}\textit{line-number}\\ \textit{string-expression}\end{array}\right\}$

Description

USING can be in a PRINT or DISPLAY statement to format the output. If *line-number* is given, the format is specified in that line by an IMAGE statement. *Line-number* must refer to a line in the current program or subprogram. See IMAGE. If *string-expression* is given, the format is defined by USING.

When USING is present, the following changes occur in the evaluation of the *print-list* of PRINT or DISPLAY.

- Comma print separators are treated as semicolons.
- The TAB function causes an error.
- The print items are formatted according to fields specified in the format definition. If the number of print items in *print-list* exceeds the number of fields in the format, the current formatted record is written. The remaining values are written in the next record, using the format definition again, from the beginning. The format is used as many times as is necessary to complete the *print-list*. A new record is generated each time the format is used. When the number of print items is less than the number of fields in the definition, output stops when the first field is encountered for which there is no print item.
- If a formatted item is too long for the remainder of the current record, it is divided into segments. The first segment fills the remainder of the current record and any remaining segments are written on the next record.

Cross Reference

DISPLAY, IMAGE, PRINT

VAL

Format

VAL(*string-expression*)

Description

The VAL function returns the numerical value of *string-expression*. Leading and trailing spaces are ignored. The VAL function is the inverse of the STR$ function.

If *string-expression* is not a valid representation of a number, an error occurs. To avoid this error, the *string-expression* may be checked first with the NUMERIC function.

Cross Reference

NUMERIC, STR$

Examples

```
170 NUM=VAL("78.6")
```

Sets NUM equal to 78.6.

```
190 LL=VAL("3E15")
```

Sets LL equal to 3.E + 15.

```
300 PRINT VAL("$3.50"):PAUSE
```

Causes an error because the string does not represent a valid numeric constant.

VERIFY

Format

VERIFY *"device.filename"* [, PROTECTED]

Description

The VERIFY command checks that data was saved on an external storage device or was loaded into memory correctly. VERIFY is used after a SAVE or OLD command to compare the program in memory to the program on the external storage device. If a difference is found, an error message is displayed. Both input/output errors 12 and 24 indicate a verification error.

Device.filename identifies the device and the file in which the program is stored. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the file.

Like SAVE, VERIFY removes any variable names which are not used in the program. If the program is protected, then PROTECTED must be specified in the VERIFY command.

Cross Reference

OLD, SAVE

Examples

```
SAVE "1.MYPROG"
```
Saves the file named MYPROG to device 1.

```
VERIFY "1.MYPROG"
```
Verifies whether the file was stored correctly.

```
OLD "1.STAT"
```
Reads the file named STAT into memory from device 1.

```
VERIFY "1.STAT"
```
Verifies whether the file was read correctly.

SUBPROGRAM VERSION

Format

CALL VERSION(*numeric-variable*)

Description

The VERSION subprogram returns a value indicating the version of BASIC that is being used. The BASIC used on the CC-40 returns a value of 10.

Example

```
170 CALL VERSION(V)
```

Sets V equal to 10.

APPENDIX A
COMMANDS & STATEMENTS

Commands & Statements

The following is a list of all CC-40 BASIC commands and statements in alphabetical order. Commands are listed first. Statements-are listed next. Most statements can be executed immediately as well as used in a program line. Those statements that can be used only in a program line have an asterisk (*) after them. Commands and statements that can be abbreviated have the acceptable abbreviation in *italics*.

Commands

CALL ADDMEM
CALL CLEANUP
*CON*TINUE
LIST
NEW
*NUM*BER
OLD
*REN*UMBER
SAVE
VERIFY

Statements

ACCEPT *
ATTACH
BREAK
CALL
CALL CHAR
CLOSE
DATA
CALL DEBUG
DEG
*DEL*ETE
DIM
DISPLAY
END
CALL ERR
CALL EXEC
FOR TO STEP
FORMAT
CALL GETLANG
CALL GETMEM
GOSUB *
GOTO *
GRAD
IF THEN ELSE
IMAGE
CALL INDIC
INPUT *
CALL IO
CALL KEY
LET
LINPUT *
CALL LOAD
NEXT
ON BREAK
ON ERROR
ON GOSUB *
ON GOTO *
ON WARNING
OPEN
PAUSE
CALL PEEK
CALL POKE
PRINT
RAD
RANDOMIZE
READ *
RELEASE
CALL RELMEM
REM
RESTORE
RETURN *
RUN
CALL SETLANG
STOP
SUB *
SUBEND *
SUBEXIT *
UNBREAK
CALL VERSION

APPENDIX B
BUILT-IN FUNCTIONS

Built-In Functions

The following list gives a brief description of each CC-40 BASIC function in alphabetical order.

Function	Value Returned and Comments
ABS	Absolute value of a numeric expression.
ACS	Trigonometric arccosine of a numeric expression given in the angular measure indicated in the display.
ASC	The numeric ASCII code of the first character of a string expression.
ASN	Trigonometric arcsine of a numeric expression given in the angular measure indicated in the display.
ATN	Trigonometric arctangent of a numeric expression given in the angular measure indicated in the display.
CHR$	A one-character string that corresponds to an ASCII code.
COS	Trigonometric cosine of a numeric expression calculated using the angular measure indicated in the display.
EOF	End-of-file condition of a file.
EXP	Exponential value (e^x) of a numeric expression.
FRE	Information about available memory.
INT	Integer value of a numeric expression.
INTRND	Integer random number with a specified maximum value.
KEY$	A one-character string that corresponds to a key pressed.
LEN	Number of characters in a string expression.
LN	Natural logarithm of a numeric expression.
LOG	Common logarithm of a numeric expression.
NUMERIC	Number that denotes whether a string expression is a valid representation of a numeric constant.
PI	The value of π (3.14159265359).
POS	Position of the first occurrence of one string expression within another.
RND	Random number from 0 to 1.
RPT$	String that is a specific number of repetitions of a string expression.

(continued)

(continued)

Function	Value Returned and Comments
SEG$	Substring of a string expression, starting at a specified point in that string and ending after a certain number of characters.
SGN	Sign of a numeric expression.
SIN	Trigonometric sine of a numeric expression calculated using the angular measure indicated in the display.
SQR	Square root of a numeric expression.
STR$	String equivalent of a numeric expression.
TAB	Column position for the next item in the *print-list* of PRINT or DISPLAY.
TAN	Trigonometric tangent of a numeric expression calculated using the angular measure indicated in the display.
VAL	Numeric value of a string expression which represents a number.

APPENDIX C
RESERVED WORDS

Reserved Words

The following is a list of all CC-40 BASIC reserved words. A reserved word may not be used as a variable name, but may be a portion of a variable name.

ABS
ACCEPT
ACS
ALL
ALPHA
ALPHANUM
AND
APPEND
ASC
ASN
AT
ATN
ATTACH
BEEP
BREAK
CALL
CHR$
CLOSE
CON
CONTINUE
COS
DATA
DEG
DEL
DELETE
DIGIT
DIM
DISPLAY
ELSE
END
EOF
ERASE
ERROR
EXP
FOR
FORMAT
FRE
GOSUB
GOTO
GRAD
IF
IMAGE
INPUT
INT
INTERNAL
INTRND
KEY$
LEN
LET
LINPUT
LIST
LN
LOG
NEW
NEXT
NOT
NULL
NUM
NUMBER
NUMERIC
OLD
ON
OPEN
OR
OUTPUT
PAUSE
PI
POS
PRINT
PROTECTED
RAD
RANDOMIZE
READ
REC
RELATIVE
RELEASE
REM
REN
RENUMBER
RESTORE
RETURN
RND
RPT$
RUN
SAVE
SEG$
SGN
SIN
SIZE
SQR
STEP
STOP
STR$
SUB
SUBEND
SUBEXIT
TAB
TAN
THEN
TO
UALPHA
UALPHANUM
UNBREAK
UPDATE
USING
VAL
VALIDATE
VARIABLE
VERIFY
WARNING
XOR

APPENDIX D
ASCII CODES & KEYCODES LIST

ASCII Codes and Keycodes

The following table lists the ASCII character codes in decimal and hexadecimal notation. The ASCII codes produced and/or character(s) displayed when the key or key sequence is pressed are shown in the column titled CHARACTER. The characters that can be displayed using the CHR$ function are shown in the column titled DISPLAYED USING CHR$. The keys that are pressed to generate the ASCII code are shown in the column titled KEY SEQUENCE.

User-defined character codes (0-6) and the user-assigned keys (codes 128-137) are shown as two asterisks (**).

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
00	00	NULL	**	[CTL] 0
01	01	SOH	**	[CTL] A
02	02	STX	**	[CTL] B
03	03	ETX	**	[CTL] C
04	04	EOT	**	[CTL] D
05	05	ENQ	**	[CTL] E
06	06	ACK	**	[CTL] F
07	07	BEL		[CTL] G
08	08	BS		[CTL] H
09	09	HT		[CTL] I
10	0A	LF		[CTL] J
11	0B	VT		[CTL] K
12	0C	FF		[CTL] L
13	0D	CR		[CTL] M or [ENTER]
14	0E	SO		[CTL] N
15	0F	SI		[CTL] O
16	10	DLE		[CTL] P
17	11	DC1		[CTL] Q
18	12	DC2		[CTL] R
19	13	DC3		[CTL] S
20	14	DC4		[CTL] T
21	15	NAK		[CTL] U
22	16	SYN		[CTL] V
23	17	ETB		[CTL] W
24	18	CAN		[CTL] X
25	19	EM		[CTL] Y
26	1A	SUB		[CTL] Z

(continued)

APPENDIX D
ASCII CODES & KEYCODES LIST

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
27	1B	ESC		[CTL] [CLR]
28	1C	FS		[CTL] =
29	1D	GS		[CTL] ;
30	1E	RS		[CTL] .
31	1F	US		[CTL] ,
32	20	Space	Space	Space
33	21	!	!	[SHIFT] !
34	22	"	"	[SHIFT] "
35	23	#	#	[SHIFT] #
36	24	$	$	[SHIFT] $
37	25	%	%	[SHIFT]/
38	26	&	&	[SHIFT] &
39	27	'	'	[SHIFT] '
40	28	(	(	[SHIFT] (
41	29	)	)	[SHIFT])
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	[SHIFT] :
59	3B	;	;	;
60	3C	<	<	[SHIFT],
61	3D	=	=	=
62	3E	>	>	[SHIFT].
63	3F	?	?	[SHIFT] ?

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	**HEX**	**Character**	**Using CHR$**	**Sequence**
64	40	@	@	[CTL] 2
65	41	A	A	[SHIFT] A
66	42	B	B	[SHIFT] B
67	43	C	C	[SHIFT] C
68	44	D	D	[SHIFT] D
69	45	E	E	[SHIFT] E
70	46	F	F	[SHIFT] F
71	47	G	G	[SHIFT] G
72	48	H	H	[SHIFT] H
73	49	I	I	[SHIFT] I
74	4A	J	J	[SHIFT] J
75	4B	K	K	[SHIFT] K
76	4C	L	L	[SHIFT] L
77	4D	M	M	[SHIFT] M
78	4E	N	N	[SHIFT] N
79	4F	O	O	[SHIFT] O
80	50	P	P	[SHIFT] P
81	51	Q	Q	[SHIFT] Q
82	52	R	R	[SHIFT] R
83	53	S	S	[SHIFT] S
84	54	T	T	[SHIFT] T
85	55	U	U	[SHIFT] U
86	56	V	V	[SHIFT] V
87	57	W	W	[SHIFT] W
88	58	X	X	[SHIFT] X
89	59	Y	Y	[SHIFT] Y
90	5A	Z	Z	[SHIFT] Z
91	5B	[	[	[CTL] 8
92	5C	¥	¥	[CTL] /
93	5D	]	]	[CTL] 9
94	5E	^	^	[SHIFT] ^
95	5F	_	_	[CTL] 5
96	60	`	`	[CTL] 3
97	61	a	a	A
98	62	b	b	B
99	63	c	c	C
100	64	d	d	D

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
101	65	e	e	E
102	66	f	f	F
103	67	g	g	G
104	68	h	h	H
105	69	i	i	I
106	6A	j	j	J
107	6B	k	k	K
108	6C	l	l	L
109	6D	m	m	M
110	6E	n	n	N
111	6F	o	o	O
112	70	p	p	P
113	71	q	q	Q
114	72	r	r	R
115	73	s	s	S
116	74	t	t	T
117	75	u	u	U
118	76	v	v	V
119	77	w	w	W
120	78	x	x	X
121	79	y	y	Y
122	7A	z	z	Z
123	7B	{	{	[CTL] 6
124	7C	|	|	[CTL] 1
125	7D	}	}	[CTL] 7
126	7E	→	→	[CTL] 4
127	7F	DEL	←	[SHIFT] ↓
128	80	**		[FN] 0
129	81	**		[FN] 1
130	82	**		[FN] 2
131	83	**		[FN] 3
132	84	**		[FN] 4
133	85	**		[FN] 5
134	86	**		[FN] 6
135	87	**		[FN] 7
136	88	**		[FN] 8
137	89	**		[FN] 9
138	8A			
139	8B			

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
140	8C	.		
141	8D			[SHIFT] /
142	8E			[SHIFT] *
143	8F			[SHIFT] −
144	90			[SHIFT] +
145	91			[CTL] *
146	92			[CTL] −
147	93			[CTL] +
148	94	DELETE		[FN] ←
149	95			[FN] →
150	96			[FN] ↑
151	97	NUMBER		[FN] ↓
152	98	VERIFY		[FN] /
153	99	SAVE		[FN] *
154	9A	OLD		[FN] −
155	9B	LIST		[FN] +
156	9C	CALL		[FN] .
157	9D	ELSE		[FN] ,
158	9E	CHR$(		[FN] ;
159	9F	GOTO		[FN] =
160	A0			[FN] [CLR]
161	A1	ASN(	｡	[FN] A
162	A2	PAUSE	｢	[FN] B
163	A3	GRAD	｣	[FN] C
164	A4	ATN(	､	[FN] D
165	A5	TAN(	･	[FN] E
166	A6	LN(	ｦ	[FN] F
167	A7	LOG(	ｧ	[FN] G
168	A8	LINPUT	ｨ	[FN] H
169	A9	NEXT	ｩ	[FN] I
170	AA	INPUT	ｪ	[FN] J
171	AB	PRINT	ｫ	[FN] K
172	AC	USING	ｬ	[FN] L
173	AD	THEN	ｭ	[FN] M
174	AE	IF	ｮ	[FN] N
175	AF	GOSUB	ｯ	[FN] O
176	B0	RETURN	ｰ	[FN] P
177	B1	SIN(	ｱ	[FN] Q
178	B2	PI	ｲ	[FN] R

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
179	B3	ACS(	ｳ	[FN] S
180	B4	SQR(	ｴ	[FN] T
181	B5	TO	ｵ	[FN] U
182	B6	EXP(	ｶ	[FN] V
183	B7	COS(	ｷ	[FN] W
184	B8	RAD	ｸ	[FN] X
185	B9	FOR	ｹ	[FN] Y
186	BA	DEG	ｺ	[FN] Z
187	BB	BREAK	ｻ	[FN] [BREAK]
188	BC		ｼ	[SHIFT] [RUN]
189	BD		ｽ	[CTL] [RUN]
190	BE	CONTINUE	ｾ	[FN] [RUN]
191	BF	RUN	ｿ	[RUN]
192	C0		ﾀ	[SHIFT] [FN] 0
193	C1		ﾁ	[SHIFT] [FN] 1
194	C2		ﾂ	[SHIFT] [FN] 2
195	C3		ﾃ	[SHIFT] [FN] 3
196	C4		ﾄ	[SHIFT] [FN] 4
197	C5		ﾅ	[SHIFT] [FN] 5
198	C6		ﾆ	[SHIFT] [FN] 6
199	C7		ﾇ	[SHIFT] [FN] 7
200	C8		ﾈ	[SHIFT] [FN] 8
201	C9		ﾉ	[SHIFT] [FN] 9
202	CA		ﾊ	
203	CB		ﾋ	
204	CC		ﾌ	
205	CD		ﾍ	
206	CE		ﾎ	
207	CF		ﾏ	
208	D0		ﾐ	
209	D1		ﾑ	
210	D2		ﾒ	
211	D3		ﾓ	
212	D4		ﾔ	
213	D5		ﾕ	
214	D6		ﾖ	
215	D7		ﾗ	
216	D8		ﾘ	
217	D9		ﾙ	

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	**HEX**	**Character**	**Using CHR$**	**Sequence**
218	DA		レ	
219	DB		ロ	
220	DC		ワ	
221	DD		ン	
222	DE		゛	
223	DF		゜	
224	E0		α	
225	E1		ä	
226	E2		β	
227	E3		ε	
228	E4		μ	
229	E5	PB	σ	[SHIFT] ↑
230	E6	OFF	ρ	[OFF]
231	E7	BREAK	g	[BREAK]
232	E8	UP	√	↑
233	E9	DOWN	⁻¹	↓
234	EA	SHIFT	j	[SHIFT] [ENTER]
235	EB		ˣ	
236	EC		¢	
237	ED		£	
238	EE		n̄	
239	EF		ö	
240	F0		p	
241	F1		q	
242	F2		θ	
243	F3		∞	
244	F4		Ω	
245	F5		ü	
246	F6	DEL	Σ	[SHIFT] ←
247	F7	INS	π	[SHIFT] →
248	F8	HOME	x̄	[CTL] ↑
249	F9	SKIP	y	[CTL] ↓
250	FA	CLR	千	[CLR]
251	FB	BTAB	万	[CTL] ←
252	FC	←	円	←
253	FD	FTAB	÷	[CTL] →
254	FE	→		→
255	FF			

APPENDIX E
TRIGONOMETRIC CALCULATIONS & RESTRICTIONS

Trigonometric Calculations and Restrictions

The following information provides restrictions for trigonometric functions, a list of trigonometric identities, and a table of trigonometric conversions.

Restrictions for SIN, COS, TAN

The approximate valid range for the arguments of SIN, COS, and TAN is given below for radians, degrees, and grads.

$|X| < PI/2*10^{10}$ radians
$|X| < 90*10^{10}$ degrees
$|X| < 10^{12}$ grads

Restrictions For Inverse Trigonometric Functions

The largest angle resulting from an arc function is 180°, π radians, or 200 grads. Because each resultant value has many angle equivalents (for example, ASN(.5) = 30°, 150°, 390°, ...), angles calculated by inverse trigonometric functions are restricted as follows.

	Range of calculated angles		
ARC Function	**Degrees**	**Radians**	**Grads**
Arcsine (x)	−90 to 90	$-\pi/2$ to $\pi/2$	−100 to 100
Arccos (x)	0 to 180	0 to π	0 to 200
Arctan (x)	−90 to 90	$-\pi/2$ to $\pi/2$	−100 to 100

APPENDIX E
TRIGONOMETRIC CALCULATIONS & RESTRICTIONS

Trigonometric Identities

The following trigonometric functions are not part of CC-40 BASIC, but may be calculated using the BASIC expressions described below. The expressions for functions that are frequently used can be assigned to any of the ten user-assigned keys as described in chapters 1 and 2.

Function	Symbol	BASIC Expression Equivalent
Secant	SEC(X)	1/COS(X)
Cosecant	CSC(X)	1/SIN(X)
Cotangent	COT(X)	1/TAN(X)
Inverse Secant	ARCSEC(X)	SGN(X)*ACS(1/X)
Inverse Cosecant	ARCCSC(X)	SGN(X)*ASN(1/X)+(SGN(X)−1)*PI/2
Inverse Cotangent	ARCCOT(X)	PI/2−ATN(X) or PI/2+ATN(−X)
Hyperbolic Sine	SINH(X)	(EXP(X)−EXP(−X))/2
Hyperbolic Cosine	COSH(X)	(EXP(X)+EXP(−X))/2
Hyperbolic Tangent	COTH(X)	−2*EXP(−X)/(EXP(X)+EXP(−X))+1
Hyperbolic Secant	SECH(X)	2/(EXP(X)+EXP(−X))
Hyperbolic Cosecant	CSCH(X)	2/(EXP(X)−EXP(−X))
Hyperbolic Cotangent	COTH(X)	2*EXP(−X)/(EXP(X)−EXP(−X))+1
Inverse Hyperbolic Sine	ARCSINH(X)	LN(X+SQR(X*X+1))
Inverse Hyperbolic Cosine	ARCCOSH(X)	LN(X+SQR(X*X−1))
Inverse Hyperbolic Tangent	ARCTANH(X)	LN((1+X)/(1−X))/2
Inverse Hyperbolic Secant	ARCSECH(X)	LN((1+SQR(1−X*X))/X)
Inverse Hyperbolic Cosecant	ARCCSCH(X)	LN((SGN(X)*SQR(X*X+1)+1)/X)
Inverse Hyperbolic Cotangent	ARCCOTH(X)	LN((X+1)/(X−1))/2

Radian, Degree, and Grad Conversions

It may be necessary to convert angular values from one unit of angle measurement to another. The following table provides the factors needed to make these conversions.

From/To	Degrees	Radians	Grads
Degrees		× PI/180	÷ 0.9
Radians	× 180/PI		× 200/PI
Grads	× 0.9	× PI/200	

Because these conversions are independent of the computer's angle setting, use care when using the results for further calculations. Before you use the result in subsequent trigonometric calculations, make certain that the appropriate angle setting has been selected.

APPENDIX F
ACCURACY INFORMATION

Accuracy Information

Calculation Accuracy

The CC-40, like all computers, operates under a fixed set of rules within preset limits. The mathematical tolerance of the computer is controlled by the number of digits it uses for calculations.

The CC-40 uses a minimum of 13 digits to perform calculations. The results are rounded to 10 digits when displayed in the default display format. The computer's 5/4 rounding technique adds 1 to the least significant digit of the display if the next nondisplayed digit is five or more. If this digit is less than five, no rounding occurs. Without these extra digits, inaccurate results such as the following would frequently be displayed.

```
1/3*3=.9999999999
```

This result occurs because 1/3 is maintained as .3333333333 in the finite internal representation of a number. However, when 1/3 x 3 is rounded to 10 digits, the answer `1.` is displayed.

The more complex mathematical functions are calculated using iterative and polynomial methods. The cumulative rounding error is usually kept beyond the tenth digit so that displayed values are accurate. Normally there is no need to consider the undisplayed digits. However, certain calculations may cause the unexpected appearance of these extra digits as shown below.

2/3 = .66666666666667 and 1/3 = .33333333333333

2/3 − 1/3 − 1/3 = .00000000000001 (displayed 1.E − 14)

Such possible discrepancies in the least significant digits of a calculated result are important when testing if a calculated result is equal to another value. In testing for equality, precautions should be taken to prevent improper evaluation.

A useful technique is to test whether two values are sufficiently close together rather than absolutely equal as shown below.

Instead of

```
IF X=Y THEN ...
```

use

```
IF ABS(X-Y) < 1E-11 THEN ...
```

Internal Numeric Representation

The CC-40 uses radix-100 format for internal calculations. A single radix-100 digit ranges in value from 0 to 99 in base 10. The computer uses a 7-digit mantissa which results in 13 to 14 digits of decimal precision. A radix-100 exponent ranges in value from −64 to +63 which yield decimal exponents from 10^{-128} to 10^{+126}. The exponent and the 7-digit mantissa combine to provide a decimal range from −9.9999999999999E+127 through −1.E−128; zero; and then +1.E−128 through +9.9999999999999E+127.

The internal representation of the radix-100 format requires eight bytes. The first byte contains the exponent, and the algebraic sign of the entire floating-point number. The exponent is a 7-bit hexadecimal value offset or biased by 40_{16} (the 16 subscript indicates hexadecimal values in this appendix). The correspondence between exponent values is shown below.

Biased hexadecimal value	00_{16}	to	40_{16}	to	$7F_{16}$
Radix-100 value	−64	to	0	to	+63
Decimal value	−128	to	0	to	+126

If the floating-point number is negative, the first byte (the exponent value) is inverted (1's complement). Each byte of the mantissa contains a radix-100 digit from 0 to 99 represented in binary coded decimal (BCD) form. In other words, the most significant four bits of each byte represent a decimal digit from 0 to 9 and the least significant four bits represent a decimal digit from 0 to 9. The first byte of the mantissa contains the most significant digit of the radix-100 number. The number is normalized so that the decimal point immediately follows the most significant radix-100 digit.

The following examples show some decimal values and their internal representations.

Decimal Number	**Internal Value**							
1	40_{16}	01_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}
10	40_{16}	10_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}
100	41_{16}	01_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}
1234	41_{16}	12_{16}	34_{16}	00_{16}	00_{16}	00_{16}	00_{16}	00_{16}
PI	40_{16}	03_{16}	14_{16}	15_{16}	92_{16}	65_{16}	35_{16}	90_{16}
−PI	BF_{16}	03_{16}	14_{16}	15_{16}	92_{16}	65_{16}	35_{16}	90_{16}

APPENDIX G
SYSTEM POWER UP & DOWN PROCEDURE

System Power Up & Down Procedure

This appendix describes the actions taken when the system is powered up and down.

System Power Up

When the CC-40 is turned off, the power continues to be supplied to the CMOS RAM chips as part of the *Constant Memory*™ feature. This power supply gives the computer the capability to retain information in memory even when the computer is turned off. Pressing the **[ON]** key turns on the full power supply and causes the TMS70C20 microprocessor to execute the power up code. The power up code resets all hardware with power up defaults and performs several operations to initialize the system.

Next the power up code checks to see if the expected values are stored in RAM locations 0802_{16} and 0803_{16} (the 16 subscript indicates hexadecimal values). One of these locations must be an $A5_{16}$ and the other a $5A_{16}$ or the system is coldstarted as described below. If the expected values are correct, an exclusive-OR checksum is calculated for all of the RAM in the system. This checksum is compared to the checksum value that was stored when the computer was turned off. If the checksums are identical, the system is warmstarted as described below. If the checksums are different and a CALL ADDMEM is in effect, the system is coldstarted. If the checksums are different and a CALL ADDMEM is not in effect, the message `Memory contents may be lost` is displayed and only essential parts of the system are initialized. This latter operation leaves the contents of program memory intact and is described below under "Partial Initialization".

Warmstart

When the CC-40 is warmstarted, a bus reset command is sent over the I/O bus. If a CALL ADDMEM is not in effect, the cartridge port is checked for a cartridge. If one is installed, pointers to the program/subprogram header list and BASIC extension information are copied into the system reserved area and the speed of the system is matched with the cartridge speed. The cartridge is then checked for a program that is to be executed at power up. If one exists, it is executed; otherwise, the system enters the system command level.

Coldstart

A coldstart of the CC-40 initializes the system by:

- Setting the language flag to English
- Initializing the expected values used at power up
- Initializing the BASIC program space
- Initializing the user-defined strings
- Initializing all important registers, RAM based trap vectors, etc.

The I/O bus is then reset and the cartridge is checked as in warmstarting the system. **Note:** Entering the command **NEW ALL** is the same as coldstarting the system without checking the cartridge port.

Partial Initialization

When the expected values are correct, but the checksum of the RAM is incorrect and CALL ADDMEM is not in effect, the message `Memory contents may be lost` is displayed. The system is powered up essentially 'as is', except that registers necessary for it to run are initialized and the cartridge port is checked.

System Power Down

When the **[OFF]** key is pressed while in system command level, the power down code is entered. This code closes all open files, resets the I/O bus, and calculates the exclusive-OR checksum of memory. This value is stored in memory for the next power up.

APPENDIX H
LOGICAL OPERATIONS ON NUMBERS

Logical Operations on Numbers

The logical operators AND, OR, NOT, and XOR can be used on integer numbers in the range −32768 to 32767. This appendix briefly describes the binary number system, conversion of decimal numbers to their binary equivalents, and the operation of the logical operators.

Binary Notation

Binary (base 2) notation is another way to express the value of a number. Our usual system, decimal (base 10) notation, uses combinations of the ten digits zero through nine. Numbers written in binary notation use only the two digits zero and one. Each position occupied by a binary digit (a 0 or 1) is called a bit.

In decimal notation, each digit in a number represents a power of 10. For example, the number 2408 in decimal notation can be written in expanded form as follows.

$$(2\times 10^3)+(4\times 10^2)+(0\times 10^1)+(8\times 10^0)$$

This is equal to 2408 as shown below.

$$\begin{array}{l}
2\times 10^3=2\times 1000=2000\\
4\times 10^2=4\times 100=400\\
0\times 10^1=0\times 10=0\\
8\times 10^0=8\times 1=\underline{8}\\
\phantom{8\times 10^0=8\times 1=}2408
\end{array}$$

In binary notation, each digit represents a power of two. For example, the binary number 101101 can be written as

$$(1\times 2^5)+(0\times 2^4)+(1\times 2^3)+(1\times 2^2)+(0\times 2^1)+(1\times 2^0)$$

For reference purposes, the powers of two and their decimal values are as follows.

$$\begin{array}{lcccccccc}
\ldots & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0\\
\ldots & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1
\end{array}$$

The decimal equivalent of 101101 can be calculated as shown below.

$$\begin{array}{l}
1\times 2^5=1\times 32=32\\
0\times 2^4=0\times 16=0\\
1\times 2^3=1\times 8=8\\
1\times 2^2=1\times 4=4\\
0\times 2^1=0\times 2=0\\
1\times 2^0=1\times 1=\underline{1}\\
\phantom{1\times 2^0=1\times 1=}45
\end{array}$$

To convert a number from decimal notation to binary notation, repeatedly reduce the decimal number by the greatest power of 2 not larger than the number until there is no remainder.

For example, the decimal number 77 can be converted to binary notation using the following technique.

The largest power of 2 contained in the number 77 is 64 (2^6). A 1 is placed in that position of the binary number as shown below.

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	0

Reducing 77 by 64 leaves a remainder of 13. The largest power of 2 contained in 13 is 8 (2^3) and a 1 is placed there. Reducing 13 by 8 leaves a remainder of 5. The largest power of 2 contained in 5 is 4 (2^2) and a 1 is placed there. Reducing 5 by 4 leaves a remainder of 1. Place a 1 in the 2^0 position.

The decimal number 77 in binary notation is shown below.

128	64	32	16	8	4	2	1
0	1	0	0	1	1	0	1

You can check the accuracy of the conversion as follows.

$$\begin{array}{r}
1 \times 2^6 = 1 \times 64 = 64 \\
0 \times 2^5 = 0 \times 32 = 0 \\
0 \times 2^4 = 0 \times 16 = 0 \\
1 \times 2^3 = 1 \times 8 = 8 \\
1 \times 2^2 = 1 \times 4 = 4 \\
0 \times 2^1 = 0 \times 2 = 0 \\
1 \times 2^0 = 1 \times 1 = \underline{1} \\
77
\end{array}$$

Logical Operations

When logical operations are performed on numbers within the valid range, the CC-40 first converts the values to their 16-bit binary equivalents. The logical operations are performed on a bit-by-bit basis, and the resulting binary number is converted back to decimal notation.

The left-most bit is reserved to indicate the sign (0 = positive; 1 = negative). Therefore, the largest number that can be represented by the remaining 15 bits is 32,767.

If a decimal number with a fractional part is used with a logical operator, the number is rounded before any logical operation is performed.

APPENDIX H
LOGICAL OPERATIONS ON NUMBERS

The following are the rules for the four logical operators.

Operator	**Rule**
AND	If both bits are 1s, the result is 1. If either bit is 0, the result is 0.
OR	If either bit is a 1, the result is 1. If both bits are zero, the result is 0.
XOR	If either bit, but not both, is 1, the result is 1. If both bits are the same, the result is 0.
NOT	If the bit is 0, the result is 1. If the bit is 1, the result is 0.

The following table shows the results of the four logical operations on all the possible combinations of bits.

AND	First bit	0	0	1	1
	Second bit	0	1	0	1
	Results	0	0	0	1
OR	First bit	0	0	1	1
	Second bit	0	1	0	1
	Results	0	1	1	1
XOR	First bit	0	0	1	1
	Second bit	0	1	0	1
	Results	0	1	1	0
NOT	Bit	0	1		
	Results	1	0		

For example, when the logical operations are performed on the numbers 77 and 67, the numbers are first converted to binary notation. The number 77 is represented in 16 bits as 0000000001001101 and the number 67 is represented in 16 bits as 0000000001000011. The results of performing an AND, an OR, and an XOR on the two values are shown below.

	AND		OR
(77)	0000000001001101	(77)	0000000001001101
(67)	0000000001000011	(67)	0000000001000011
(65)	0000000001000001	(79)	0000000001001111

	XOR
(77)	0000000001001101
(67)	0000000001000011
(14)	0000000000001110

The results of performing an AND, OR, and an XOR on 77 and 67 can be obtained on your CC-40 by entering the following.

```
PRINT 77 AND 67; 77 OR 67; 77 XOR 67
```

Using the logical operator NOT on 77 and 67 is shown below.

```
        NOT 77                      NOT 67
(77)    0000000001001101    (67)    0000000001000011
(-78)   1111111110110010    (-68)   1111111110111100
```

To display the results of NOT 77 and NOT 67, enter the following.

```
PRINT NOT 77; NOT 67
```

Note that the results of NOT 77 and NOT 67 have a 1 in the leftmost bit which denotes that they represent negative numbers. In the CC-40 a negative binary number is represented as the two's complement of the absolute value of the number.

To obtain the two's complement of a binary number, change each 0 bit to 1 and each 1 bit to 0. Then add 1 to this changed number. For example, the two's complement of 77 is obtained as shown below.

```
77 in binary                   0000000001001101
Change each bit                1111111110110010
add 1                                         1
-77 in two's-complement form   1111111110110011
```

A more detailed description of binary arithmetic is beyond the scope of this appendix. Refer to a standard reference book on this subject for more information.

DEBUG Monitor

The DEBUG subprogram is used to access the debug monitor. The debug monitor is designed to be used with the Editor/Assembler to read and modify memory and to run and debug assembly language programs and subprograms. More detailed information on the debug monitor is available in the Editor/Assembler manual. Indiscriminate use of the debug monitor may result in loss of data in memory.

Running the DEBUG Monitor

To execute the debug monitor, type **CALL DEBUG** and press **[ENTER]**. The prompt `MONITOR:` is displayed, followed by the flashing cursor, to indicate that the debug subprogram is active. The prompt changes to : after the first command is entered. CALL DEBUG can be used as a statement in a BASIC program to allow debugging of assembly language subprograms called from a BASIC program. The following notational conventions are used in this appendix.

- The characters that are **bold** in examples must be typed by the user and if the characters are to be entered, the **[ENTER]** key must be pressed.
- The space bar and the **[ENTER]** key are used to execute most commands.
- The **[CLR]** and **[BREAK]** keys are used to cancel commands.
- The ← key can be used to erase the previous character typed when entering an address or data.
- Memory addresses can be entered in either hexadecimal or decimal notation. A number is assumed to be in hexadecimal notation unless it is preceded with a decimal point, in which case it is assumed to be in decimal notation.

Displaying Memory—The D Command

The display command is D. It displays the contents of memory eight bytes at a time in hexadecimal notation. Execute the display command by entering

D nnnn

where **nnnn** is the address of the start of the first eight-byte block of memory to be displayed.

The monitor responds by displaying

```
nnnn b0 b1 b2 b3 b4 b5 b6 b7
```

where b0 represents the first byte and b7 represents the eighth byte.

The ↑ or − key can be used to display the values in the next lower addresses in multiples of eight. The ↓ or + key or the space bar can be used to display the values in the next higher addresses in multiples of eight.

To leave the D command, press the **[CLR]**, **[ENTER]**, or **[BREAK]** key.

Examining and Modifying Memory—The M Command

The examine and modify command is M. It can be used to read and modify individual bytes of memory. Execute the command by entering

M nnnn

where **nnnn** is the address of the first byte to be examined or modified.

The monitor responds by displaying

```
nnnn=xx
```

where nnnn is the address of the byte and xx is the hexadecimal value stored in that byte. A new value can be stored in that byte by entering the value.

The ↑ or − key can be used to display the value in the next lower address. The ↓ or + key or the space bar can be used to display the value in the next higher address.

To leave the M command, press the **[CLR]**, **[ENTER]**, or **[BREAK]** key.

Copying Memory—The C Command

The copy command is C. It can be used to copy a block of memory to a specified location. Execute the copy command by entering

C ssss dddd llll

where **ssss** specifies the lowest address of the block to be copied, **dddd** specifies the first memory location to be copied to, and **llll** specifies the number of bytes to copy. **llll** bytes are copied one at a time from **ssss** to **dddd**.

Modifying Processor Information—The P Command

The modify program information command is P. It can be used to modify the microprocessor's program counter (PC), status register (SR), and stack pointer (SP). Execute the command by typing

```
P
```

The monitor responds by displaying

```
PC=nnnn
```

where nnnn is the current hexadecimal value of the program counter. A new value can be entered for the program counter. If the program counter is not to be modified, press the space bar.

The monitor responds by displaying

```
ST=xx
```

where xx is the current hexadecimal value of the status register. A new value can be entered for the status register. If the status register is not to be modified, press the space bar.

The monitor responds by displaying

```
SP=yy
```

where yy is the current value of the stack pointer. A new value can be entered for the stack pointer. If the stack pointer is not to be modified, press the space bar to exit from the command.

Note: Indiscriminate modification of the program counter or stack pointer followed by the E commmand may cause undesirable results.

Setting Break Points—The B Command

The breakpoint command is B. It can be used to set up to two breakpoints. A breakpoint is set by entering an address for either of the breakpoints. Entering an address for a breakpoint causes a break to occur when that location is executed. To set a breakpoint, type the following.

```
B
```

The monitor responds by displaying

```
B nnnn
```

where nnnn is the current value of the first breakpoint. (A 0000 value means no breakpoint has been set.) To set only one breakpoint, type the address, press **[ENTER]**, and the monitor prompts for another command.

To set a second breakpoint, press the space bar instead of **[ENTER]** after the first address has been typed. The monitor responds by displaying

```
nnnn
```

where nnnn is the address of the first breakpoint set. The address for the second breakpoint can then be entered.

When a break occurs, the monitor displays the prompt

```
nnnn xx yy:
```

where nnnn is the hexadecimal address where the breakpoint occurred, xx is the hexadecimal value of the status register at the time the breakpoint occurred, and yy is the hexadecimal value of the stack pointer.

Executing a breakpoint automatically clears any breakpoint(s) set.

Single Stepping—The S Command

The single step command is S. It can be used to execute the instruction at the address in the program counter. Execute the command by typing the following.

S

This instruction has the same effect as executing a breakpoint at the instruction following the current one. (See the B command.)

Executing—The E Command

The execute command is E. It can be used to start execution at the address given in the program counter. Execute the command by entering the following.

E

Paging—The R Command

The page command is R. It can be used to change the page on which code is executing. The page can be either the system ROM page or a cartridge page. Execute the command by typing the following.

R

The monitor responds by displaying

```
CARTRIDGE PAGE=x
```

where x is the current page that is selected for the cartridge. If the cartridge page is not to be modified, press the space bar to display the system ROM page. Otherwise, type the new cartridge page number and then press the **[ENTER]** key to exit from the command or the space bar to modify the system page. When the space bar is pressed, the monitor responds by displaying

```
SYSTEM PAGE=n
```

where n is the current system ROM page that is selected while the machine language program is running. The new system ROM page can then be entered.

Help—The ? Command

The help command is ?. It can be used to display a list of the commands used in the debug monitor. Execute the command by typing the following.

```
?
```

The monitor responds by displaying

```
COMMANDS=Q,B,E,M,C,S,D,P,R
```

Press the space bar or **[ENTER]** to leave the command.

Exiting—The Q Command

The exit command is Q. It is used to leave the debug monitor by typing the following.

```
Q
```

The monitor responds by displaying

```
:Q
```

B can be typed to continue program execution at the next BASIC statement or **I** can be typed to return to system command level.

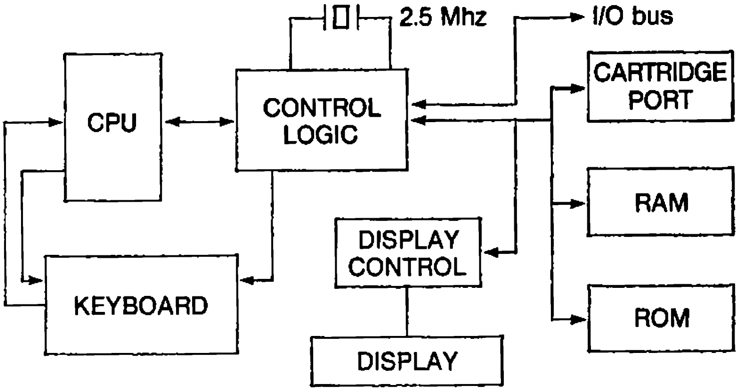
Technical Information

This appendix provides technical information on the Texas Instruments Compact Computer Model CC-40 and presumes some knowledge of digital circuits and assembly language programming. The CC-40 hardware, memory organization, memory expansion, system command level, and the *HEX-BUS™* interface are described in this appendix. More detailed system information is given in the Editor/Assembler manual.

CC-40 Hardware

The CC-40 is built around the TMS70C20 CMOS microprocessor. The 70C20 is an 8-bit microprocessor with 2K bytes of internal ROM and 128 bytes of RAM (called the register file). A 256-byte block, starting at 0100_{16} (the subscript 16 indicates a hexadecimal number) is used for memory-mapped peripheral ports.

Composing the rest of the system is a 32K-byte ROM, up to 18K bytes of RAM, the display controller subsystem; Liquid Crystal Display (LCD), keyboard, power supply, and control logic. A block diagram is shown below.



CC-40 Memory Organization

The TMS70C20 microprocessor can access a total of 64K bytes of memory. This memory is mapped into several distinct sections.

- A 128-byte register file
- A peripheral file
- System RAM
- The cartridge port
- System ROM
- Processor ROM

Each of these sections is addressed at a specific area in the memory map as shown in the following table.

Address:		
Decimal	**Hex**	**Description**
0	0000	Register
127	007F	File (128 bytes)
128	0080	unused
255	00FF	(128 bytes)
256	0100	Peripheral
511	01FF	File (256 bytes)
512	0200	unused
2047	07FF	(1.5K bytes)
2048	0800	System
20479	4FFF	RAM (up to 18K bytes)
20480	5000	Cartridge
53247	CFFF	port (32K bytes)
53248	D000	System
61439	EFFF	ROM (8K bytes)
61440	F000	unused
63487	F7FF	(2K bytes)
63488	F800	Processor
65535	FFFF	ROM (2K bytes)

System Memory Map

Note: When a RAM cartridge is added to less than 18K of built-in RAM, the cartridge overlays the memory starting at 1000_{16}.

The Register File

The register file contains the following groups of registers used in BASIC.

1. The A register
2. The B register
3. The assembly language subroutine stack
4. BASIC reserved (program pointer, current program character, etc.)
5. General purpose temporary registers (floating-point operations, I/O temporaries, etc.)

The general layout of the register file is shown below.

Address: Decimal	Hex	Description
0	0000	A Register
1	0001	B Register
2 57	0002 0039	Assembly language subroutine stack
58 74	003A 004A	BASIC statement temporaries
75 87	004B 0057	BASIC reserved area
88 126	0058 007E	General purpose temporaries
127	007F	Floating-point status

Register File

The Peripheral File

The TMS70C20 contains special instructions for performing I/O. These instructions access a particular section of the memory map called the peripheral file. This area contains several built-in peripheral registers such as the I/O control register, timer control registers, A and B ports, and the peripheral file expansion area. The general layout of the peripheral file is shown below.

Address:		
Decimal	**Hex**	**Description**
256	0100	I/O control register
257	0101	Reserved
258	0102	Timer data register
259	0103	Timer control register
260	0104	A port input data
261	0105	Reserved
262	0106	B port output data
263 271	0107 010F	unused
272	0110	Address control register
273	0111	Power on hold latch
274	0112	I/O bus-data
275	0113	I/O bus-bus available
276	0114	I/O bus-handshake ctl
277	0115	Piezo control

(continued)

(continued)

Address: Decimal	Hex	Description
278	0116	Low battery sense line
279 280	0117 0118	unused
281	0119	Page control register
282	011A	Clock control register
283 511	011B 01FF	unused

Peripheral File

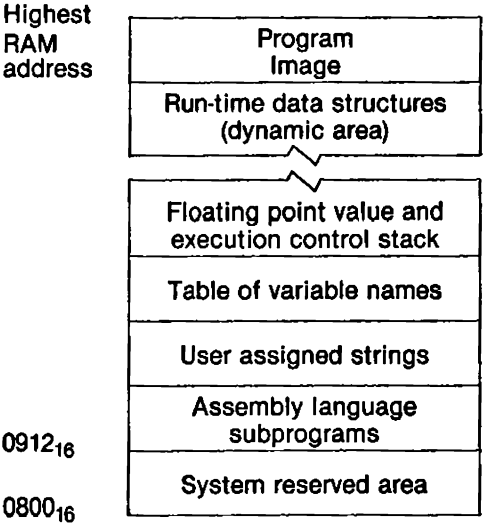
System RAM

RAM starts at 0800_{16} in the CC-40. The RAM can be increased by using a *Memory Expansion* cartridge. A minimum of 402 bytes of the memory is reserved by BASIC for the following.

- RAM-based trap vectors
- List pointers
- Random number seeds
- Permanent buffers (such as the keyboard input buffer)
- Other necessary information

The rest of memory is used to store the floating point stack, the dynamic area, any program in memory, and any user loaded assembly language subprograms. Each user-assigned string requires 1 byte plus the length of the string.

The use of RAM is outlined in the table below.



RAM Usage

BASIC Program Image

A BASIC program requires the following quantities of memory.

- Eleven bytes for overhead information.
- Four bytes of overhead for each program line.
- Two bytes of overhead plus the length of the variable name for each variable name. Each additional use of the same variable name requires only one byte.
- One byte of memory for each use of the following BASIC program elements.

  ABS, ACCEPT, ACS, ALL, AND, APPEND, ASC, ASN, AT, ATN, ATTACH, BEEP, CALL, CHR$, CLOSE, COS, DATA, DIM, DISPLAY, END, EOF, ERASE, EXP, FOR, FRE, GOSUB, GOTO, IF, IMAGE, INPUT, INT, INTERNAL, INTRND, KEY$, LEN, LET, LINPUT, LN, LOG, NEXT, NOT, NULL, NUMERIC, ON, OPEN, OR, OUTPUT, PAUSE, PI, POS, PRINT, RANDOMIZE, READ, REC, RELATIVE, RELEASE, REM, RESTORE, RETURN, RND, RPT$, SEG$, SGN, SIN, SIZE, SQR, STEP, STOP, STR$, SUB, SUBEND, SUBEXIT, TAB, TAN, THEN, TO, UPDATE, USING, VAL, VALIDATE, VARIABLE, XOR, statement separator :, comma ,, semicolon ;, left parenthesis (, right parenthesis), not-equal < >, less-than-or-equal < =, greater-than-or-equal > =, equal =, less-than < , greater-than > , concatenation &, addition or unary plus + , subtraction or unary minus −, multiplication *, division /, exponentiation ^, file number # .
- Two bytes of memory for each use of the following BASIC program elements.

  ALPHA, ALPHANUM, BREAK, CONTINUE (CON), DEG, DELETE (DEL), DIGIT, ELSE, ERROR, FORMAT, GRAD, LIST, NEW, NUMBER (NUM), OLD, PROTECTED, RAD, RENUMBER (REN), RUN, SAVE, UALPHA, UALPHANUM, UNBREAK, VERIFY, WARNING, tail remark ! .
- The number of bytes required for the following BASIC program elements are shown below.

  a. Three bytes for each line reference which appears in control transfer statements such as GOTO, GOSUB, ON GOTO, ON GOSUB, and IF THEN ELSE. Line references can also be in statements and commands such as ON ERROR, RESTORE, RUN, and DELETE.

  b. One byte of overhead plus two to eight bytes for each numeric constant. The number of bytes depends upon the number of significant digits in the floating-point representation of the constant. Trailing zeros are truncated from the normal representation to generate the program representation.

c. Two bytes of overhead plus the length of the string characters contained between the quotation marks for each quoted string constant. The length does not include the quotation marks. Within the quoted string two consecutive quotation marks count as a single quotation mark.

d. Two bytes of overhead plus the length of the string for each unquoted string constant. Leading and trailing spaces are ignored. Subprogram names in SUB, CALL, ATTACH, and RELEASE statements are unquoted strings. Unquoted strings also appear in REM, IMAGE, and DATA statements.

Run-time Data Structures

The following memory requirements are necessary to run a BASIC program. These structures are allocated dynamically during program execution.

- Four bytes of overhead for each variable in the main program. In addition the following memory is required for those variables.
  a. Eight bytes for each simple numeric variable.
  b. Two bytes of overhead for each dimension of each numeric array plus 8 bytes for each element value.
  c. Four bytes of overhead for the value of each simple string variable plus the length of the value. (**Exception:** If the variable is assigned a simple constant value in the program, the overhead for the value is reduced to 2 bytes. For example, A$ = "HELLO" requires 4 bytes of overhead for the variable A$ and 2 bytes of overhead for the value. A$ = "HELLO"&B$ requires 4 bytes of overhead for A$, plus 4 bytes of overhead for the value, plus the length of the value.)
  d. Two bytes of overhead for each dimension of a string array plus 4 bytes of overhead for each element value plus the length of each element value (see exception above).
- Eleven bytes of overhead for each BASIC subprogram plus 2 bytes for each variable (including arrays), plus 2 bytes for each dimension of each array. In addition each active and each attached subprogram has two bytes of overhead plus memory space for variables as described previously. If the subprogram is attached, the additional memory space remains allocated until the subprogram is released. Otherwise, the memory space is released when the subprogram terminates. See the CALL, ATTACH, and RELEASE statements.

- Twenty-one bytes of overhead for each open file or device plus the maximum record length specified in the OPEN statement. If record length is not specified in the OPEN statement, it is specified by the device when the OPEN statement is executed. This memory is released when the file or device is closed.
- Twenty-four bytes of the execution control stack during execution of each FOR NEXT loop.
- Eight bytes of the execution control stack during execution of the subroutine for each GOSUB or ON GOSUB.
- Sixteen bytes of the execution control stack during execution of the error handling subroutine for ON ERROR line-number.
- Twenty-four bytes of the execution control stack during execution of the subprogram for each BASIC subprogram CALL.
- Sixteen bytes of the execution control stack for an occurrence of a breakpoint until the program is continued or the capability to continue is destroyed.

Memory Expansion

The amount of memory added by a *Memory Expansion* cartridge depends upon the amount of resident memory and the size of the *Memory Expansion* cartridge. The table below shows the memory capacities resulting from adding a particular *Memory Expansion* cartridge to a specific amount of resident memory.

		Cartridge Memory Size (K bytes)		
		2	8	16
Resident Memory (K bytes)	2	4	10	18
	4	4	10	18
	6	4	10	18
	10	4	10	18
	18	20	26	34

System Command Level

The system command level of the BASIC interpreter is a loop which repetitively performs three phases of operation.

1. An input line is accepted from the keyboard and echoed in the display.
2. The input line is translated into an internal representation which can be processed by the execution level of the BASIC interpreter.
3. Based on the content of the input line and the key used to terminate the input, the command level determines how to use the input and processes it accordingly. After processing the current input line, the command level loops back to the input phase to accept another line from the user.

The key to the proper functioning of the command level is the decision concerning how to use the input line. This process begins in the line compression routine which translates the input line into its corresponding internal representation. This routine decides whether or not the input is a BASIC program line. If the input line begins with a valid line number (an integer from 1 to 32766), followed by one or more spaces, followed by an alphabetic character, the at sign, the underline, or tail remark symbol (!), it is translated as a program line. Otherwise, the input is translated as a statement (or command) for immediate execution or an equation for immediate calculation.

BASIC program lines are edited into the current BASIC program in memory. If the current program is not a BASIC or a protected program, an error occurs. If the input is not a program line, the command level must decide whether it is a statement/command or an equation. A statement/command is executed immediately as if it were a one line program. An equation is evaluated and the result is displayed left justified in the display. Other calculations may be appended to the result using the +, −, *, /, and ^ operators. However, if a new equation or statement/command is input, the result is automatically cleared before the input is accepted.

Error Messages

The following lists describe the cause of each error message generated by the CC-40. The first list, arranged alphabetically by message, provides detailed information about the probable cause of each error. The second list, arranged in ascending order by error code, serves as a cross reference to locate the message associated with a particular error code.

When an error message is displayed, the →, ←, ↑, ↓, and **[SHIFT] [PB]** keys can be used to display additional system error information and to edit an erroneous line.

[SHIFT] [PB] is used when an error occurs after a line is entered. **[SHIFT] [PB]** displays the erroneous entry which can then be edited and entered again

→ is used when an error occurs during program execution. → displays the error code and the line number of the line being executed (when the error occurred) in an `Enn Lmmmmm` format where `nn` is the error code and `mmmmm` is the line number. (This line is not necessarily the one that is the source of the problem since an error may occur because of values generated or actions taken elsewhere in the program.)

When an I/O error occurs, → displays either the error code, file number, and line number in an `E0, xxx #yyy mmmmm` format or the error code, device number, and line number in an `E0, xxx "yyy" mmmmm` format. `xxx` is the I/O error code, `#yyy` is the file number or `"yyy"` is the device number, and `mmmmm` is the line number of the line that was executing when the error occurred.

← can be used to redisplay the error message immediately after the → key has been pressed.

↑ or ↓ is used when an error occurs during program execution to display the program line that was executing when the error occurred.

Errors can be handled in a program using ON ERROR and CALL ERR. Refer to chapters 4 and 5 for more information.

Messages Listed Alphabetically

Code Message/Cause

29 `Bad argument`

- Invalid argument provided for one of the built-in numeric, string, or file functions such as LOG, CHR$, and EOF.
- Invalid argument provided for one of the option clauses in an input/output statement such as AT, SIZE, VALIDATE, and TAB.
- Arguments in a CALL statement did not match the requirements for the subprogram called.

07 `Bad INPUT data`

- Entered more than one value at a time in an INPUT or ACCEPT statement.
- Invalid data from a file in an INPUT or LINPUT statement.

17 `Bad line number`

- Line number specified in a statement or command was less than 1 or greater than 32766.
- RENUMBER command generated a line number greater than 32766.

18 `Bad program type`

- Entered a BASIC program line with an assembly language or other non-BASIC program in memory.
- Entered a SAVE, VERIFY, BREAK *line-list*, UNBREAK *line-list*, NUMBER, RENUMBER, LIST, CONTINUE *line-number*, RUN *line-number*, or DELETE *line-group* command with an assembly language or other non-BASIC program in memory.
- Attempted to CALL a main program or RUN a subprogram.
- Attempted to ATTACH a main program or an assembly language subprogram.
- File specified for LOAD subprogram did not contain a relocatable, assembly language subprogram.

32 `Bad subscript`

- Subscript value too large.
- Missing comma between subscripts or missing parentheses around subscripts.
- Incorrect number of subscripts.

(continued)

(continued)

Code	Message/Cause
04	Bad value • Index value in ON GOTO or ON GOSUB statement was zero or greater than the number of line number entries. • Raised a negative value to a non-integer power. • Invalid value provided for one of the option clauses in an input/output statement such as AT, SIZE, REC, and VARIABLE. • Attempted a logical operation (AND, OR, XOR, or NOT) with a value less than −32768 or greater than 32767.
31	BASIC extension missing • Attempted to execute an extended BASIC statement or function without the extension in the system. • May also occur when the contents of memory have been improperly modified (see System error).
37	Break • A breakpoint occurred or the break key was pressed.
10	Can't do that • Attempted to perform a string operation as an immediate calculation. • Entered CONTINUE command when not stopped at a breakpoint. • A SUBEXIT or SUBEND statement was encountered when no subprogram was called. For example, CONTINUE *line-number* specified a line in a subprogram after the main program stopped at a breakpoint.
43	DATA error • Out of data in the current program or subprogram. • Improper data list in a DATA statement. For example, items not separated by commas. • During an attempt to read a numeric item, the data read was not a valid representation of a numeric constant.
34	Division by zero • Evaluation of a numeric expression includes division by zero; result is replaced by 9.9999999999999E+127 with the appropriate algebraic sign.

(continued)

(continued)

Code Message/Cause

23 Error in image

- Null string provided as image string.
- Numeric format field specified more than 14 significant digits.
- *Print-list* included a print-item but image string had only literal characters.

02 Expression too complex

- Too many functions, operators, or levels of parentheses pending evaluation; expression must be simplified or must be performed in two or more steps in separate statements.

24 File error

- *File-number* specified in an OPEN statement refers to a file already opened.
- *File-number* in an input/output statement, other than OPEN, did not refer to an open file.
- *File-number* or *device* number in an input/output statement was greater than 255.
- Attempted to INPUT or LINPUT from a file opened in OUTPUT or APPEND mode.
- Attempted to LINPUT from an internal-type file.
- Attempted to PRINT to a file opened in INPUT mode.
- Used REC clause in an input/output statement which accessed a sequential file.
- Missing period or comma after device number in *device* or *filename* specification.

30 FOR without NEXT

- More FOR statements than NEXT statements in a program or subprogram. **Note:** the line number reported is the last line of the current program or subprogram, not the line containing the unmatched FOR statement.

11 Illegal after SUBEND

- Statement other than REM, !, END, or SUB used after a SUBEND statement.

(continued)

(continued)

Code Message/Cause

13 Illegal FOR-NEXT nesting

- Too many levels of nested FOR NEXT loops.
- Same control variable used in nested FOR NEXT loops.

19 Illegal in program

- Used CALL ADDMEM, CALL CLEANUP, CONTINUE, DELETE *line-group*, LIST, NEW, NUMBER, OLD, RENUMBER, SAVE, or VERIFY in a program.

01 Illegal syntax

- Missing parentheses or quotation mark(s).
- Missing statement separator (:) or tail remark symbol (!).
- Missing or extra comma(s). For example:
  - –between arguments in *argument-list*
  - –between line numbers in *line-number-list*
  - –between variables in *variable-list*
  - –after *file-number* in input/output statements
- Missing hyphen in line sequence.
- Missing argument or clause. For example:
  - –no limit value after TO or increment value after STEP
  - –no line number or statement after THEN or ELSE
  - –no *string-constant* following IMAGE
  - –no *line-number* or *string-expression* after USING
  - –no value before or no value after a binary operator such as *, /, ^ , or &
  - –no input variable following INPUT, LINPUT, ACCEPT, or READ
- Invalid argument or clause. For example:
  - –a string variable is used as *control-variable* in FOR
  - –a numeric variable is used as input variable in LINPUT
  - –VALIDATE or NULL is used in a DISPLAY statement
  - –USING or TAB is used with an internal-type file
  - –the size of print item exceeds record size for an internal-type file

(continued)

(continued)

Code Message/Cause

- Missing keyword. For example:
  - –no TO after FOR
  - –no THEN after IF
  - –no GOTO or GOSUB after ON *numeric-expression*
  - –no STOP, NEXT, or ERROR after ON BREAK
  - –no PRINT, NEXT, or ERROR after ON WARNING
- Improperly placed keyword. For example:
  - –DIM or SUBEND is used after a DIM statement in a multiple statement line
  - –a statement begins with a non-statement keyword such as TO, ERROR, VARIABLE, SIZE
  - –a misspelled variable results in a keyword or a misspelled keyword in a variable
  - –a keyword is used as a variable, such as ON VAL GOTO or IF STOP=1 THEN
- Duplicated option in input/output statement. For example:
  - –more than one AT, SIZE, ERASE ALL is in ACCEPT or DISPLAY
  - –more than one string expression is in VALIDATE
  - –more than one *open-mode*, *file-type*, *file-organization* is in OPEN
- Missing or invalid *filename* in OLD, SAVE, VERIFY, or DELETE file command.
- Invalid character in statement. For example “%”, “?”, “,”, “O”, “C”, etc., are valid only within quoted strings or in an IMAGE or REM statement.
- Invalid character within a numeric constant.

08 `Invalid dimension`

- Specified array dimension was negative or was not a numeric constant.
- Too many elements specified for an array.
- More than three dimensions specified for an array
- Missing comma between dimensions or missing parentheses around dimensions of an array.

(continued)

(continued)

Code Message/Cause

00 I/O error

- An error was returned by a peripheral device during an input/output (I/O) statement or command, or while using the EOF function. A special I/O code is returned by the device and is displayed after the message. Common I/O error codes are described in the I/O ERROR CODES section of this appendix.

  The error code is followed by the *file-number* or the *device* number, whichever is appropriate to the statement or command being executed. A number sign indicates a *file-number* and quotation marks indicate a *device* number. Both the common codes and other device-dependent I/O error codes are described in the peripheral manuals.

16 Line not found

- Could not find a line number specified in BREAK, CONTINUE, DELETE, GOSUB, GOTO, ON ERROR, USING, RESTORE, RUN, or BREAK.
- RENUMBER could not find a referenced line. The command replaced the reference by 32767, which is not a valid line number.

12 Line reference out of range

- BASIC statement referred to a line number which was lower than the first (or higher than the last) line number of the current program or subprogram.

27 Line too long

- The internal representation of a program line or immediate statement(s) was too long.
- The LIST representation of a program line exceeded 80 characters.

35 Memory contents may be lost

- When the power was turned on, the computer determined that the contents of the constant memory were not the same as when the power was turned off. However, some system data was correct, so the loss may or may not be serious. This message often appears when the reset key is pressed while the power is on.

(continued)

(continued)

Code Message/Cause

127 Memory full

- Insufficient space to add, insert, or edit a program line.
- Insufficient space to allocate variables for a program or subprogram.
- Insufficient memory to allocate space for a string value.
- Insufficient space to load a program or subprogram into memory.
- Insufficient space to OPEN a file or device.
- Insufficient space to assign a user-assigned string.
- Attempted to allocate more than the largest available block of memory using the GETMEM subprogram.

14 Missing RETURN from error

- An error processing subroutine terminated with a SUBEXIT or SUBEND statement instead of a RETURN statement.

42 Missing SUBEND

- SUBEND missing in a subprogram.
- Encountered a SUB statement within a subprogram; a subprogram cannot contain another subprogram.

44 Must be in program

- ACCEPT, CALL with BASIC language subprograms, GOSUB, GOTO, INPUT, LINPUT, ON ERROR *line-number*, ON GOSUB, ON GOTO, READ, RESTORE *line-number*, SUB, SUBEXIT, and SUBEND statements can be executed only in a program.

39 Must be in subprogram

- SUBEXIT or SUBEND statement encountered in a main program.

25 Name table full

- Defined more than 95 variable names. The CLEANUP subprogram can be used to delete all variable names not used in the current program in memory.

28 Name too long

- More than 15 characters in a variable or subprogram name.

(continued)

APPENDIX K
ERROR MESSAGES

(continued)

Code Message/Cause

06 `NEXT without FOR`

- More NEXT statements than FOR statements in a program or subprogram.
- *Control-variable* in NEXT statement did not match *control-variable* in corresponding FOR statement.
- Executed a NEXT statement without previously executing the corresponding FOR statement.

40 `No RAM in cartridge`

- Called ADDMEM subprogram with no cartridge installed or with a cartridge which did not contain RAM memory.

33 `Overflow`

- A numeric value was entered or a numeric expression was evaluated which resulted in a number whose absolute value was greater than 9.9999999999999E+127; the value is replaced by 9.9999999999999E+127 with the appropriate algebraic sign.

15 `Program not found`

- RUN statement did not find the specified program.
- CALL statement did not find the specified subprogram.

20 `Protection violation`

- Attempted to insert, delete, or edit a line with a protected program in memory.
- Attempted to LIST, SAVE, NUMBER, or RENUMBER a protected program.

45 `RETURN without GOSUB`

- Executed a RETURN statement without previously executing the corresponding GOSUB statement.

05 `Stack underflow`

- Attempted to remove a value from the execution control stack when it was empty. This error only occurs when the contents of memory have been improperly modified (see System error).

41 `Statement must be first on line`

- SUB statement used after the first statement in a multiple statement line.

(continued)

(continued)

Code Message/Cause

03 `String-number mismatch`

- Used a string argument where a numeric argument was expected or a numeric argument where a string argument was expected.
- Assigned a string value to a numeric variable or a numeric value to string variable.
- A numeric variable or expression was provided as a prompt in an INPUT or LINPUT statement.

36 `String truncation`

- String operation (concatenation or RPT$) resulted in a string with more than 255 characters; the extra characters are discarded.

21 `Subprogram in use`

- Called an active subprogram; subprograms may not call themselves, directly or indirectly.

126 `System error`

- This error generally occurs when the contents of memory have been lost or improperly modified. For example, memory may be modified by a loss of power or by improper use of the POKE, RELMEM, EXEC, or DEBUG subprogram(s).

38 `System initialized`

- Displayed when circumstances force the complete initialization of the system. The system is initialized when the power is turned on and one of the following occurs.
  - –The computer determines that the contents of memory have been destroyed (may occur after changing the batteries).
  - –The computer determines that previously appended expansion RAM (through ADDMEM subprogram) is no longer in the system.
- The message may also appear when the reset button is pressed because much of the same memory checking is performed,. (The system initialization procedure is described in appendix G.)

(continued)

(continued)

Code	Message/Cause
26	Unmatched parenthesis • A statement or expression did not contain the same number of left and right parentheses. • Left and right parentheses in a statement or expression did not match up. For example, SIN(1+)PI/2)(where SIN(1+(PI/2)) was intended.
22	Variable not defined • Attempted to perform a calculation with a variable which has not been defined. • Encountered an undefined variable in a program or subprogram. This error can occur when CONTINUE *line-number* specifies a line which is not in the same program or subprogram where the breakpoint occurred.
09	Variable previously defined • Variable in a DIM statement appeared previously in the current program or subprogram. • Variable referenced using the wrong number of dimensions. For example, a variable was first used as a simple variable and later used as an array in the same program or subprogram.

Error Codes List in Ascending Order

Code	Message
00	I/O error
01	Illegal syntax
02	Expression too complex
03	String-number mismatch
04	Bad value
05	Stack underflow
06	NEXT without FOR
07	Bad INPUT data
08	Invalid dimension
09	Variable previously defined
10	Can't do that
11	Illegal after SUBEND
12	Line reference out of range
13	Illegal FOR-NEXT nesting
14	Missing RETURN from error
15	Program not found

(continued)

(continued)

Code	Message
16	Line not found
17	Bad line number
18	Bad program type
19	Illegal in program
20	Protection violation
21	Subprogram in use
22	Variable not defined
23	Error in image
24	File error
25	Name table full
26	Unmatched parenthesis
27	Line too long
28	Name too long
29	Bad argument
30	FOR without NEXT
31	BASIC extension missing
32	Bad subscript
33	Overflow
34	Division by zero
35	Memory contents may be lost
36	String truncation
37	Break
38	System initialized
39	Must be in subprogram
40	No RAM in cartridge
41	Statement must be first on line
42	Missing SUBEND
43	DATA error
44	Must be in program
45	RETURN without GOSUB
126	System error
127	Memory full

I/O ERROR CODES

The following list details the standard input/output (I/O) error codes. Some peripherals may have additional error codes; if so, they are explained in the peripheral manual.

I/O errors are displayed in one of the following forms.

- `I/O error ccc #fff`
- `I/O error ccc "ddd"`

where `ccc` is the I/O error code listed below or in the peripheral manual, `fff` is the file number assigned in an OPEN statement, and `ddd` is the device code associated with the peripheral device.

Code Definition

1 DEVICE/FILE OPTIONS ERROR

- Incorrect or invalid option specified in "*device.filename*".
- *Filename* too long or missing in "*device.filename*".

2 ERROR IN ATTRIBUTES

- In an OPEN statement, incorrect attributes (*file-type*, *file-organization*, *open-mode*, *record-length*) were specified for an existing file.

3 FILE NOT FOUND

- The file specified in one of the following operations does not exist.
  –OPEN statement using the INPUT attribute
  –OLD "*device.filename*"
  –RUN "*device.filename*"
  –DELETE "*device.filename*"
  –CALL LOAD("*device.filename*")

4 DEVICE/FILE NOT OPEN

- Attempted to access a closed file with a INPUT, LINPUT, PRINT, or CLOSE operation.
- File specified in EOF function is closed.

5 DEVICE/FILE ALREADY OPEN

- Attempted to OPEN or DELETE an open file.
- Attempted to FORMAT storage medium on a device which has a file open.

(continued)

(continued)

Code Definition

6 DEVICE ERROR

- A failure has occurred in the peripheral. This error can occur when directory information on a tape was lost, the peripheral detected a transmission error or a medium failure, etc.

7 END OF FILE

- Attempted to read past the end of the file.

8 DATA/FILE TOO LONG

- Attempted to output a record which was longer than the capacity of the device.
- A file exceeded the maximum file length for a device.

9 WRITE PROTECT ERROR

- Attempted to FORMAT a write-protected storage medium.
- Attempted to OPEN a write-protected file in OUTPUT or UPDATE mode.
- Attempted to DELETE a file from a write-protected medium.

10 NOT REQUESTING SERVICE

- Response to a service request poll when the specified device did not request service. (This code is used in special applications and should not be encountered during normal execution of BASIC programs.)

11 DIRECTORY FULL

- Attempted to OPEN a new file on a device whose directory is full.

12 BUFFER SIZE ERROR

- When an existing file was opened for input or update, the specified record length (VARIABLE XXX) was less than the length of the largest record in the existing file.
- The VERIFY command found the program in memory was smaller than the program on the storage medium.

13 UNSUPPORTED COMMAND

- Attempted an operation not supported by the peripheral.

(continued)

(continued)

Code Definition

14 DEVICE/FILE NOT OPENED FOR OUTPUT

- Attempted to write to a file or device opened for input.

15 DEVICE/FILE NOT OPENED FOR INPUT

- Attempted to read from a file or device opened for output or append.

16 CHECKSUM ERROR

- The checksum calculated on the input record was incorrect.

17 RELATIVE FILES NOT SUPPORTED

- Device specified in OPEN does not support relative record file organization.

19 APPEND MODE NOT SUPPORTED

- Device specified in OPEN statement does not support append mode.

20 OUTPUT MODE NOT SUPPORTED

- Device specified in OPEN statement does not support output mode.

21 INPUT MODE NOT SUPPORTED

- Device specified in OPEN statement does not support input mode.

22 UPDATE MODE NOT SUPPORTED

- Device specified in OPEN statement does not support update mode.

23 FILE TYPE ERROR

- File type specified in OPEN statement is not supported by the specified device.
- File type specified in OPEN statement does not match file type of existing file or device.

24 VERIFY ERROR

- Program or data in memory does not match specified program or storage medium.

25 LOW BATTERIES IN PERIPHERAL

- Attempted an I/O operation with a device whose batteries are low.

(continued)

(continued)

Code Definition

26 UNINITIALIZED MEDIUM

- Attempted to open a file on uninitialized storage medium.
- Attempted to open a file on storage medium which has been accidentally erased or destroyed.

32 MEDIUM FULL

- No available space on storage medium.

254 ILLEGAL IN SLAVE MODE

- Attempted a normal (master) I/O bus operation while the computer was in peripheral (slave) mode. (This error occurs during some special applications and should not be encountered during normal execution of a BASIC program.)
- **Note:** Improper modification of memory by the POKE, RELMEM, EXEC, or DEBUG subprograms can result in the computer being placed in peripheral (slave) mode.

255 TIME-OUT ERROR

- Lost communication with the specified device.
- Specified device is not connected to the I/O bus.

APPENDIX L
SERVICE & WARRANTY INFORMATION

In Case of Difficulty

In the event that you have difficulty with your Compact Computer, the following instructions may help you diagnose and remedy the problem. Usually you can correct the problem without returning the unit to a service facility. If the suggested remedies are not successful, contact Texas Instruments' Consumer Relations Department by mail or telephone as described later in the section IF YOU HAVE QUESTIONS OR NEED ASSISTANCE.

Note: All peripherals attached to the CC-40 should be turned on for proper operation.

If one of the following symptoms appears, try the suggested remedy. If you are operating your computer with peripheral devices and the remedy does not correct the problem, remove the peripherals. If the symptom disappears, a peripheral is the most likely source of the difficulty. Refer to the appropriate peripheral or accessory manual for more information on the cause of the problem.

Symptom	Remedy/Cause
No display	Check that power is on. Move the display contrast control to see if the display becomes visible. If there is still no display, replace the batteries with fresh AA alkaline batteries.
No flashing cursor	Check the I/O display indicator to see if any I/O operations are in progress. If the indicator is on, wait for all peripheral activity to cease. If the indicator is still on several minutes later, disconnect the *HEX-BUS* interface cable from the computer. Then press the reset key. If the I/O indicator is not on, the system may be locked up. Press the **[BREAK]** key to try to halt the computer. If the word BREAK appears in the display, enter **CON** to continue executing the program in memory.

(continued)

(continued)

Symptom	Remedy/Cause
No flashing cursor	If the **[BREAK]** key is inoperable, press the reset key. The message `Memory contents may be lost` should be displayed. Press the **[CLR]** key to clear the display. You can check if your program is still in memory by entering **LIST**. If pressing the reset key does not cause the cursor to reappear, the batteries should be removed. Normally, the system is then initialized and any program in memory erased.

Returning Your Computer

When returning your Compact Computer for repair or replacement, also return any software cartridges that were being used when the difficulty occurred. For your protection, the CC-40 should be sent insured; Texas Instruments cannot assume any responsibility for loss of or damage to the CC-40 during shipment. It is recommended that the CC-40 be shipped in its original container to minimize the possibility of shipping damage. Otherwise, the CC-40 should be carefully packaged and adequately protected against shock and rough handling. Send shipments to the appropriate Texas Instruments Service Facility listed in the warranty. Please include information on the difficulty experienced with your computer as well as return address information including name, address, city, state, and zip code.

If the CC-40 is in warranty, it will be repaired or replaced under the terms of the Limited Warranty. Out-of-warranty units in need of service will be repaired or replaced with reconditioned units (at TI's option), and service rates in effect at the time will be charged. Because our Service Facility serves the entire United States, it is not feasible to hold units while providing service estimates. For advance information concerning our flat-rate service charges, please call our toll-free telephone number (800) 858-4565.

Exchange Centers

If your Compact Computer requires service and you do not wish to return the unit to your dealer or to a service facility for repair, you may elect to exchange the computer for a factory-reconditioned computer of the same model (or equivalent model specified by TI) by taking the computer to one of the exchange centers which have been established across the United States.

A handling fee will be charged by the exchange center for in-warranty exchange. Out-of-warranty exchanges will be charged at the rates in effect at the time of the exchange. To determine if there is an exchange center in your area, look for Texas Instruments Incorporated Exchange Center in the white pages of your telephone directory or look under the Calculating and Adding Machines and Supplies heading in the yellow pages. Please call the exchange center for the availability of your model. You can write or call Texas Instruments Consumer Relations Department for more information.

If You Have Questions or Need Assistance

For General Information

If you have questions concerning Compact Computer repair or peripheral, accessory, or software purchase, please call our Customer Relations Department at (800) 858-4565 (toll free within the contiguous United States). The operators at these numbers cannot provide technical assistance.

For Technical Assistance

For technical questions such as programming, specific computer applications, etc., you can call (806) 741-2663. We regret that this is not a toll-free number, and we cannot accept collect calls. As an alternative, you can write to:

Texas Instruments Consumer Relations
P.O. Box 53
Lubbock, Texas 79408

Because of the number of suggestions which come to Texas Instruments from many sources, containing both new and old ideas, Texas Instruments will consider such suggestions only if they are freely given to Texas Instruments. It is the policy of Texas Instruments to refuse to receive any suggestions in confidence. Therefore, if you wish to share your suggestions with Texas Instruments, or if you wish us to review any computer program which you have developed, please include the following in your letter:

> "All of the information forwarded herewith is presented to Texas Instruments on a nonconfidential, nonobligatory basis; no relationship, confidential or otherwise, expressed or implied, is established with Texas Instruments by this presentation. Texas Instruments may use, copyright, distribute, publish, reproduce, or dispose of the information in any way without compensation to me."

90-Day Limited Warranty

THIS TEXAS INSTRUMENTS COMPACT COMPUTER WARRANTY EXTENDS TO THE ORIGINAL CONSUMER PURCHASER OF THE COMPUTER.

Warranty Duration:

This computer is warranted to the original consumer purchaser for a period of 90 days from the original purchase date.

Warranty Coverage:

This computer is warranted against defective materials or workmanship. **THIS WARRANTY DOES NOT COVER BATTERIES AND IS VOID IF THE PRODUCT HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE OR OTHER CAUSE NOT ARISING OUT OF DEFECTS IN MATERIAL OR WORKMANSHIP.**

Warranty Disclaimers:

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE 90-DAY PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE COMPUTER OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

Legal Remedies:

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

APPENDIX L
SERVICE & WARRANTY INFORMATION

Warranty Performance:

Please contact the retailer from whom you purchased the computer and determine the exchange policies of the retailer.

During the above 90-day warranty period, your TI Compact Computer will be repaired or replaced with a new or reconditioned comparable model (at TI's option) when the computer is returned either in person or by prepaid shipment to a Texas Instruments Service Facility listed below.

Texas Instruments strongly recommends that you insure the computer for value, prior to shipment.

The repaired or replacement computer will be warranted for 90 days from date of repair or replacement. Other than the cost of postage or shipping to Texas Instruments, no charge will be made for the repair or replacement of in-warranty computers.

Texas Instruments Consumer Service Facilities

U.S. Residents:

Texas Instruments Service Facility
2303 North University
Lubbock, Texas 79415

Canadian Customers Only:

Geophysical Services Incorporated
41 Shelley Road
Richmond Hill,
Ontario, Canada L4C5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments Consumer Service
831 South Douglas Street
El Segundo, California 90245
(213) 973-1803

Texas Instruments Consumer Service
6700 Southwest 105th St.
Kristin Square
Suite 110
Beaverton, Oregon 97005
(503) 643-6758

Important Notice Regarding Programs and Book Materials

The following should be read and understood *before* purchasing and/or using TI's Compact Computer.

TI does not warrant that the programs contained in this computer and accompanying book materials will meet the specific requirements of the consumer, or that the programs and book materials will be free from error. The consumer assumes complete responsibility for any decision made or actions taken based on information obtained using these programs and book materials. Any statements made concerning the utility of TI's programs and book materials are not to be construed as express or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THESE PROGRAMS OR BOOK MATERIALS OR ANY PROGRAMS DERIVED THEREFROM AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THESE PROGRAMS OR BOOK MATERIALS, AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THIS COMPACT COMPUTER. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER AGAINST THE USER OF THESE PROGRAMS OR BOOK MATERIALS BY ANY OTHER PARTY.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

INDEX

TEXAS INSTRUMENTS
INCORPORATED
Dallas, Texas

Printed in U.S.A.