CoCo-C

by Infinitum Technology

CoCo-C

An RSDOS C Compiler Development System
for the
Tandy Color Computer

User's Manual

License Statement

Table of Contents

CoCo-C Overview

Ultra Editor (CoCo 3)

Line Editor (CoCo 2)

CoCo-C Compiler

CoCo-C Library Functions

CoCo-ASM Assembler

CoCo-C Library Linker

CoCo-C Examples

1. Introduction:

CoCo-C - An RSDOS C Compiler for the TRS-80 Color Computer

CoCo-C is a complete programming environment designed to be used on a CoCo 1, 2, or 3 with a minimum of 64K of memory and at least 1 disk drive. The programs contained within CoCo-C are: an Editor, a C Compiler, an Assembler, and a Library Linker. All these programs run under the CoCo-C's Command Coordinator for speed and simplicity. The compiler is capable of producing position independent (re-locatable) code, or absolute (ROM-able) code. All programs created with CoCo-C are in machine language and do not require a "run-time" program for execution. LOADM and EXEC is all that is needed to run a user-created CoCo-C program. The CoCo-C Function Library supports many of the standard C library functions along with several "special" functions which are unique to the CoCo. CoCo-C also supports mixed programming, so that you can combine C, Assembler and BASIC commands into one program.

The disk that comes with CoCo-C is a "flippy" disk containing the necessary programs and files for either the CoCo 1 or 2, or the CoCo 3. The 'A' side of the disk is for the CoCo 3 and the 'B' side of the disk is for the CoCo 1 or 2. The main difference between the CoCo 3 vs. the CoCo 1 or 2 versions, is the support for the 80 column screen. This is primarily found in the text editor provided on the disk. Also, the CoCo-3's library contains more functions which take advantage of the CoCo 3's high resolution screen formats .

The files that are contained on the supplied disk are as follows:

CC.BAS	- CoCo-C's Command Coordinator
EDITOR.BAS	- Full Screen Editor (CoCo 3) or Line Editor (CoCo 1 or 2)
COMPILER.BIN	- CoCo-C Compiler
ASSEMBLR.BIN	- CoCo-ASM 6809 Assembler
LINKER.BIN	- CoCo-C Library Linker
CLIB.BIN	- re-locatable function library
CSTART.C	- Start-up / initialization routines for CoCo 1, 2 or 3
CLIB.INC	- function library entry table
STDIO.H	- header file for CoCo's Standard I/O
BASIC.H	- header file for BASIC library functions
CHARIO.ASM	- source code for CoCo's character I/O
STDLIB.C	- source code for Standard C library functions
FILELST.C	- source code for example C application

Both the 'A' side and the 'B' side of the disk contain the same file names with different versions for either the CoCo 3, or CoCo 1 or 2.

The CoCo-C disk is not copy-protected and may be backed up using standard backup procedures.

2. Scope and References:

The purpose of this manual is to describe the specifications and operation of CoCo-C as a compiler for the Color Computer, as opposed to being a tutorial for the C programming language. Therefore, it is recommended if you have a limited knowledge of C that you read the book "The C Programming Language" by Kernighan and Ritchie.

It is also recommended, though not essential, that you have a general understanding of 6809 assembly language programming. Two excellent books on this topic are "6809 Assembly Language Programming" by Lance Leventhal and "TRS-80 Color Computer Assembly Language Programming" by William Barden Jr.

The Editors that are contained in this package are Bob van der Poel's Ultra Editor (CoCo 3) and Line Editor (CoCo 1 or 2). These editors were chosen because they are easy to use, and are specifically tailored for programming environments. The Ultra Editor features on-screen help menus and can be custom configured for either 40 or 80 columns with choice of foreground and background colors.

All programs within this package runs under RSDOS or equivalent. The OS-9 Operating System is not required.

3. Hardisks and Ramdisks:

Since the programs within CoCo-C do not make any undocumented DOS calls, it is possible for CoCo-C to run on either a hard disk or RAM disk providing that the following is true:

1. The hard disk or RAM disk must exactly emulate a standard CoCo disk. (35 tracks, single sided, 18 sectors/track, 256 bytes/sector).

2. The DOS being used is 100% BASIC compatible with RSDOS disk commands.
   (ie. OPEN O,"TEST.DAT:2"AS 1)

3. The DOS being used must accept a DEFAULT DRIVE designator

This is also true for any CoCo-C user programs needing to be created on a hard disk or RAM disk. If all the above is true, the entire contents of CoCo-C may be copied to the desired hard disk or RAM disk. The programs will operate normally as long as the drive containing CoCo-C is assigned as the default drive.

CoCo-C has been tested to work successfully with Burke & Burke's RGBDOS along with several tested 35 track RAM disks.

4. Start-up / Operation:

CoCo-C is a collection of programming tools designed for the development of C language programs. All of these programs are in machine language and are executed from CoCo-C's Command Coordinator. Four steps are required to create a C program, they are as follows:

1. Edit - for creating or editing an existing C program.
2. Compile - to convert the C code into assembly code.
3. Assemble - to create a binary image of the assembled code.
4. Link - to "merge-in" the library, making the final ML program.

After you made a backup copy of the appropriate version of CoCo-C, insert the backup disk into the default drive and type: **RUN "CC <ENTER>**. This will cause the Command Coordinator's main menu to appear on the screen. You are now ready to begin by typing the first letter of the desired option.

Please note that the CoCo-C disk must remain in the default drive during the entire development phase of your C program.

Note:

If you are anxious to try out your new CoCo-C package, but you're not ready to read the entire manual at this time, you may jump ahead to the Examples section of this manual to familiarize yourself with the development phase of CoCo-C.

5. The Command Coordinator:

CoCo-C uses a command "shell" type program to coordinate the loading and executing of the individual programs within CoCo-C. This command "shell" is called the "Command Coordinator" and is named "CC.BAS" on your distribution disk. It is written in BASIC and has a small machine language loader appended to it which is used to load the compiler.

The purpose of the Command Coordinator is to act as the "main menu-ing" master program, which controls all the other programs within CoCo-C. The only program you need to run for CoCo-C is "CC.BAS". All programs within CoCo-C not only execute from the coordinator, but return to it as well.

To run the Command Coordinator, simply type RUN "CC <ENTER>. This will cause the Command Coordinator's main menu to appear on the screen. From that point on, all you have to do is enter the first letter of your selected option.

5.1 Command Coordinator options.

The available options within the Command Coordinator are as follows:

5.1.1 Edit

The Edit option loads and executes the included text editor for CoCo-C. For the CoCo 1 or 2, this is Bob van der Poel's Line Editor. For the CoCo 3, it's Bob van der Poel's Ultra Editor. Use Edit for creating new or editing existing C or assembly language source files. Edit may also be used for examining the output text files created by the compiler or assembler.

5.1.2 Compile

This is CoCo-C's C compiler. Selecting Compile will load and execute the compiler. Use this option when you are ready to compile your C source file.

5.1.3 Assemble

This is the CoCo-ASM assembler. Selecting Assemble will load and execute the assembler. Since the compiler produces assembly language files as output, this option will assemble the compiler-created file. The Assemble option may also be used to assemble user-created source files

5.1.4 Link

The Link option loads and executes CoCo-C's Library Linker. The purpose of Link is to "merge-in" CoCo-C's 90+ function library along with your compiled and assembled program. This will produce a stand-alone relocatable (by default) binary file ready for LOADM and EXEC.

5.1.4 Quit

The Quit option simply exits the Command Coordinator and returns you to BASIC. This option should only be used as a temporary exit, with the intent to return back to the Command Coordinator. You may use this option to do some housekeeping (ie. checking the directory, killing files, setting the default drive #, etc.). Just don't do anything that may destroy the Command Coordinator's contents (ie. LOAD(M), CLEAR, etc.). To return to back to the Command Coordinator, simply type RUN.

5.1.5 Re-Boot

This is the best way to exit the Command Coordinator. This will perform a cold start on the CoCo. Use this option when you are ready to test your compiled and assembled program.

5.2 Customizing the Command Coordinator:

The BASIC portion of the Command Coordinator may be modified to suit your needs. The CoCo 3 version of the Command Coordinator assumes an RBG monitor and defaults to an 80 column white on blue text screen when loading the Compiler, Assembler, or Linker.

This information is contained in line #360 in the "CC.BAS" file, and may be modified if desired.

For modifications available to the Editor, refer to the Ultra Editor section of this manual.

1. Introduction:

ULTRA EDITOR is a machine language program appended to a short BASIC loader. The loader takes care of simple housekeeping chores like opening and closing files. By using BASIC for these tasks compatibility between different DOSes and DOS versions is maintained. The loader also permits user modifications and makes the making of backup copies much easier.

ULTRA EDITOR's main purpose is for the editing of source code for compilers and assemblers. But of course, it can be put to many other uses. You may find the macro abilities of ULTRA EDITOR to be useful in editing various text files, etc. as well as BASIC programs -- just remember that ULTRA EDITOR does not like lines longer than 128 characters.

2. Loading and Running the Editor:

ULTRA EDITOR is first invoked by running CoCo-C's command coordinator. Type **RUN "CC <ENTER>**. After the menu appears, select **E** (for edit). This will load and execute the editor.

The first thing you'll see is a copyright notice and a prompt for a filename. This prompt is for the initial file you wish to edit. If you press <ENTER> without a filename you'll advance to the main menu; if you enter a filename, the file will be loaded and you'll advance to the edit mode. Note: if the file does not exist or if an IO error is encountered, you will also enter the edit mode with no data being read or files created.

To put some order to this section, we'll assume that you didn't enter a filename and are now in ULTRA EDITOR's main menu.

3. Main Menu Commands:

In the main menu you have eight options. Each one can be accessed by pressing the key corresponding to the first letter of the choice.

Note that the top of the screen shows the current free memory in the current buffer. In addition, if the alternate buffer has been selected, a reverse video message 'secondary buffer' will be displayed. All the commands in the main menu act on the current buffer. More on buffers later.

3.1 Editor:

This should be pretty obvious. It takes you into edit mode (lots more on this later!).

3.2 Read file:

This option permits the reading of a new file. Careful, the existing contents of the buffer will be overwritten. For this reason a check is made to see if a save has been done since last leaving the editor. If no save has been done you'll be informed and asked to confirm your choice.

3.3 Append file:

This is the same as the 'read' option, only in this case the file will be appended to the end of the file in memory.

3.4 Save buffer:

This option saves the buffer to disk. Also, see the comments on 'partial save/print' below.

3.5 Print buffer:

This option prints the buffer. If the printer is off line a message will be printed. You must press <ENTER> or <BREAK> to return to the editor's main menu. Also, see the comments on 'partial save/print' below.

3.6 Macro save:

This option allows you to save the macros you defined. This is done by re-saving ULTRA EDITOR to disk. Since the buffers used for the macro definitions are within the program, they get saved along with the program. See section 18 for macro description.

3.7 CoCo-C Main Menu:

This option exists ULTRA EDITOR and returns you to CoCo-C's Command Coordinator (Main Menu). The 'files not saved' warning will be displayed if no saves have been done.

3.8 @FILE:

The default filename will be displayed after this prompt. The default is the last filename entered for a load or save command. It can be forced to a different name by pressing <@> and entering a filename.

4. Main Menu Options:

4.1 Filename Entry:

Whenever a filename is required, the default filename (if defined) will be displayed at the top of the screen. If you press <ENTER> by itself the default name will be used. This can be a great time saver when you make frequent saves of a file.

By default, the /TXT extension and the drive number set by the DRIVE command will be appended to your filenames unless these items are already present in your input. In addition, the filename DIR is reserved. Entering DIR at a filename prompt will cause a disk directory to be displayed. If you wish to have a directory of a drive other than the default drive, use the format DIR:2 etc. Note the colon!

4.2 Partial Print and Save Options:

If a block has been defined in the current buffer, you will be given an additional option when you select the 'print' and 'save' options. You'll be asked if you wish the entire file to be saved/printed or only the defined block. Respond accordingly.

5. Editor Commands:

Now for the good stuff!! But first a few conventions used in this documentation.

5.1 Whenever you see a key refered to in braces (ie. {A}), we are referring to a control key combination. These keys are generated by holding down the key marked <CTRL> and pressing the key at the same time. All valid control key combinations result in editor commands.

5.2 Keys referred to in pointed braces (ie. <ENTER>) mean normal keys. In some cases a different value will be generated when the <SHIFT> key is pressed with the key. Sometimes (see the alternate key definitions, below) the <ALT> key will modify the keypress. If the particular <ALT> combination is not defined, the <ALT> key will have no effect.

5.3 Every command can be aborted by pressing {BREAK}, Remember, this means you hold <CTRL> down and press <BREAK>. This keypress combination will generate a 'keyboard abort' error message at the top of the screen.

5.4 Whenever you are in the editor, you will see a line at the top of the screen highlighted in reverse video. This line is called the 'status line'. It advises you of the current column position of the cursor, the current line number, the amount of free memory available, and the current editor modes. This

line is also used for the entry of data required by some commands. In this case it is referred to as the 'command line'.

6. Entering Text:

In many ways ULTRA EDITOR is much like a word processor editor, but in many ways it is much different. The basic differences lie with the type of text ULTRA EDITOR and a word processor handle. ULTRA EDITOR is a line editor. This means that it presumes all the text is composed of lines -- unlike most word processors which handle text as a continuous stream of characters (the printing portion of word processors handle mundane chores like determining the end of lines, etc.).

To enter text in ULTRA EDITOR, just move the cursor to where you want the character to appear using the various movement commands below. Each time a character is typed, ULTRA EDITOR first checks to see if it is command. If it is, the command is executed. Otherwise, the character is placed into the buffer and the cursor is moved one position to the right.

ULTRA EDITOR permits lines to a maximum length of 128 characters. A virtual screen of 24 lines by 128 characters is maintained in memory. The horizontal scroll built into the CoCo 3 has been utilized to permit instantaneous windowing into any portion of this screen using either the 40 or 80 column mode. As the cursor moves off the screen (either left or right) the window moves to display the additional text.

When inserting characters on a line, the characters at the right of the line will be lost. This will only occur of course the character at column 127 is a non-space character. Normally you will be editing lines much shorter than 128 characters, so the problem will not occur.

7. Help:

Help on all the various commands is available to you whenever you are in the editor. To access the help screens just press the <F1> key. Note that this key is not active during the definition of the macro. Also, <F1> will abort commands like a 'change' and a 'jump'. Help consists of a number of screens, press <ENTER> to advance from screen to screen and finally back to the editor.

8. Alternate Key Definitions:

By using the <ALT> key combination, the following characters can be generated:

<ALT> key:	character:	ASCII value:	comments:
<ALT/1>	|	$7C (124)	
<ALT/2>	~	$7E (126)	
<ALT/8>	[	$5B (91)	remember as the '(' key
<ALT/9>	]	$5D (93)	remember as the ')' key
<ALT/,>	{	$7B (123)	remember as the '<' key
<ALT/.>	}	$7D (125)	remember as the '>' key
<ALT/UP ARROW>		$5E (94)	this is displayed as the up arrow on the screen, but it is really a '^', and this is how it appears on most printers.
<ALT/LEFT ARROW>		$5F (95)	this is displayed as a left arrow on the screen, but it is really an 'underline', and this is how it appears on most printers.
<ALT/DOWN ARROW>		$60 (96)	this is displayed as a '^' on the screen. It's correct ASCII representation is really '`'. Note that this character has special uses, detailed later.

Other changes from the standard Color Computer key definitions:

<SHIFT/@>	\	$5C (92)

All the other keys follow the standard Color Computer convention. The differences in the true ASCII representations of the ALT/ARROW combinations are due to the hardware of the Color Computer, not ULTRA EDITOR.

9. Key Repeat:

When in the editor all keys will repeat if they are held down for more than 1/2 second. Characters generated with the <CTRL> key will NOT repeat. When typing in filenames etc. from ULTRA EDITOR's main menu, the above keyboard redefinitions are not active.

10. Cursor Movement Commands:

Once text has been loaded into the buffer (or typed in), there are many powerful cursor movement commands to help you get to where you want to be.

<LEFT ARROW>	moves the cursor one position to the left. Of course you can't move past column 0.
<RIGHT ARROW>	moves the cursor one position to the right. Again you can't move past column 127.
<SHIFT/LEFT ARROW>	moves the cursor to the start of the current line. The start of the line is the first non-space character on the line. On a blank line the cursor is moved to column 0.
{LEFT ARROW}	this moves the cursor to column 0, the absolute left of the screen.
{RIGHT ARROW}	moves the cursor to column 127, the absolute left of the screen.
<CLEAR>	moves the cursor to the next word (actually the next space character).
<SHIFT/CLEAR>	moves the cursor to the start of the previous word.
<UP ARROW>	moves the cursor up one line.
<DOWN ARROW>	moves the cursor down one line.
<SHIFT/UP ARROW>	moves the cursor up one line and to the start of this line.
<SHIFT/DOWN ARROW>	moves the cursor down one line and to the start of this line.
{UP ARROW}	moves the cursor up 22 lines in the file (a page command).
{DOWN ARROW}	moves the cursor down 22 lines in a file.
{:}	moves the cursor to the start of the file. (This is easier to remember as a {*}, for START.)
{-}	moves the cursor to the end of the file. (The logic here is that '-' is beside ':'.)

11. Find Commands:

ULTRA EDITOR has a number of commands to help you find text in a file. Please note these comments on the ^ character. They apply to CHANGE and JUMP commands as well.

The ^ character (generated by <ALT> <DOWN ARROW>) has three special uses when entering commands, depending upon where it is in the entered text.

11.1 If a ^ is the first character on the line, upper and lower case characters will be identical. (ie. In a FIND command both '^text' and '^TEXT' would match on 'text', 'TEXT' and 'TeXt').

11.2 If a ^ is in the middle of the entered text (ie. not the first position or the last non-space character) it is treated as a wild card character. This means that TE^T would match TEST and TEXT.

11.3 A ^ as the last character on a line is used to include trailing blanks in the input. Normally trailing blanks are dropped into the bit bucket, but this option is included in case you wish to match something like 'TEXT '. If you input 'TEXT <ENTER>' the last space will not be included. By using 'TEXT ^<ENTER>' the space will be part of the input; the ^ will be eaten by the bit bucket monster. Note that if you wish to have a wild card character as the last character on a line, then you'll have to use two ^'s. The last one gets eaten, the second last is retained as a wild card. A similar sequence is required if you wish to have a wildcard character as the first character in your input. Since the first ^ is interpreted as an ignore case directive, two ^'s are required -- the first one ignores case, the second is a wild card. Due to this limitation, you cannot use the wildcard in the first position and have the search case-sensitive.

{F} Find a match. You will be prompted for a target to search for on the command line. The buffer will be scanned backwards from the current cursor position.

{B} This is the same as FIND, except that the text is scanned backwards from the current cursor position.

Note that text entered with {F} and {B} is saved in a buffer for the next two commands. This buffer is cleared when you access ULTRA EDITOR's main menu.

{N} Find the next occurrence of the pattern entered with {F} or {B} (forward from the current cursor position).

{L} Find the previous occurrence of the pattern entered with {F} or {B} (backward from the current cursor position).

12. Change Commands:

{C} Change a pattern with a replacement pattern. You'll be prompted first with a 'Change:' then a 'To:'. The first match from the current cursor position will be changed. If no match is found, an error message will be displayed on the status line.

{A} Change again. Just like in Find, the input for Change is saved in a buffer. This is a different buffer than the ones used by Find and Jump; it is also cleared by exiting to ULTRA EDITOR's main menu. The {C} command is the same as {C}, it just avoids the initial input. By combining {C}/{A} and {F}/{N} you can have a prompted change command -- first find the occurrence, then change it if you want.

{G} Global change. This command accepts input just like {C}, it then changes all occurrences from the start of the file. Essentially it is the same as going to the start of a file, doing a {C} and then repeated {A}'s. Note that a 'no match' error will be generated when the last change is made. This should be a 'no more matches' error, but in the interest of memory conservation the error generated by {A} has been used.

13. Jump Commands:

{J} The JUMP command is the quickest way for you to go from one part of your program to another. The JUMP command has five different forms -- each one is designed for specific applications, so please read carefully. Also, JUMP never checks for case (ie. uppercase=lowercase), but for the sake of consistency, the ^ character can be included as the first character of a JUMP command -- it is ignored.

When the JUMP command is initiated, you will see a prompt on the command line. Here are the various ways you can answer:

13.1 A number between 0 to 65535. This will move the cursor to the line matching the number input. If the highest line number is less than the number input, an error will be displayed. Note that an entry like '123TEXT' is interpreted as line number 123.

13.2 A string of characters corresponding to a label in an assembly language program. For example, 'LABEL' will advance the cursor to a line starting (in column 0) with the text 'LABEL'. The match must terminate with a space, a '(' or an end-of-line. This means that the following lines would match:

LABEL

LABEL JMP LABEL

LABEL (X,Y)

Note that LABEL1 and LABELS will not match. The search for the label is done from the start of the buffer. If you have 2 identical labels, the second will never be found by JUMP. If no match is found, an error message will be displayed on the command line.

13.3 If the input is a '<', the cursor will be moved to the 'start of block' character, if it exists.

13.4 If the input is a '>', the cursor will be moved to the 'end of block' character, if it exists.

14. Editing Commands:

ULTRA EDITOR contains many powerful editing commands. Remember, these commands effect the line on which the cursor is positioned. Also, all the changes are made in a buffer. The changes are not actually inserted into the main buffer until a non-editing command is given (ie. move cursor to next line, <ENTER>, {J} or {C}, etc.). This means that any changes made by mistake can be cancelled with {BREAK}.

{K} Kill or delete the current line. When a line is killed, it is stored in a buffer and can be restored with the {U} command -- it can't be undone with {BREAK}.

{U} Unkill or restore a line which has been deleted with {K}. Note that a line killed once can be unkilled any number of times. This may be useful in duplicating a line - Kill it, Unkill it at the same position and finally, Unkill it again at the new position.

{H} Hack line. All non-space characters from the cursor position to the end of the line are deleted.

{Y} Yank. Delete until the next space -- essentially a word delete.

<BREAK> A true backspace. The cursor is moved one position left and the character at the new cursor position is deleted.

<SHIFT/BREAK> Delete the character at the cursor.

{SPACE} Insert a space at the current cursor position.

15. Block Commands:

A number of ULTRA EDITOR commands work not in the entire text buffer, but rather on a defined block within the buffer. Before any of the block commands will function, a block must be defined. If no block is defined, an appropriate error message will be displayed.

{<} Insert a begin block marker before the current line. Once the marker has been inserted, a line will be inserted in the text. ULTRA EDITOR will not permit the insertion of a begin marker and an end marker.

{>} Insert an end block marker before the current line. An end block marker cannot be inserted before a begin block marker.

{D} Delete block. The cursor will be placed at the start of the defined block, and the block will be deleted. Careful with this command -- the block is not saved in a buffer. As a reminder a warning message is displayed. You must enter a word beginning with the letter 'Y' (either 'Y' or 'y' or 'yes' are okay) and press <ENTER> for the delete to occur. Any other entry will abort the delete command.

{S} Report the block size. The size of the defined block will be displayed on the command line (a size of 0 indicates that a block has not been defined). In addition, the current size of the text buffer will also be displayed.

NOTE: The following block commands rely on a buffer for data transfer. The maximum size of any of these transfers is approximately 8000 characters. If the defined block is too large, an error message will be displayed and the command will be aborted. Use {S} to determine how much to shorten the block before attempting the command again.

{M} Move block. The defined block will be transferred to the current cursor position and the old block will be deleted. The block markers will also be deleted.

{R} Copy (Replicate) block. The defined block will be copied to the current cursor position. The block markers will be deleted, but the old block will remain.

{X} Xtransfer. This command will transfer a marked block in the alternate buffer (more on buffers later) to the current cursor position in the current buffer. No changes are made to the data in the alternate buffer.

{Z} Delete all block markers in the current buffer.

16. Buffers:

As mentioned earlier, ULTRA EDITOR maintains two separate text buffers. The main buffer is approximate 50,000 bytes long, the secondary buffer is approximately 16,000 bytes long. With the {X} command it is possible to transfer data between the two buffers.

The buffers share macro commands and the FIND and CHANGE buffers -- so editing commands in one buffer can easily be duplicated in both buffers.

So, why would you use a secondary buffer? As in so many things with computers we're sure that you'll find lots of uses -- uses we never thought of. But as a start, you can edit two separate documents at the same time. Or, you could have a program in the main buffer, read a library file into the secondary buffer and then transfer parts of the library to different parts of the main buffer.

{@} Toggle to alternate buffer. The command line will have a '*' in position 40 if the secondary buffer is active. Also, in the main menu the text 'secondary buffer' will appear if it is active.

17. Editing Modes:

Editing can be done in either insert or overstrike mode. Insert means that a character is inserted at the current cursor position and the text to the right of the cursor is moved to the right. Overstrike means that the character at the cursor is overlaid with a new character. In both cases, the cursor is moved one position right after the character is inserted.

<F2> puts ULTRA EDITOR into insert mode. An 'I' will be displayed in the status line indicating 'Insert'.

<SHIFT/F2> puts ULTRA EDITOR into overstrike mode. An 'O' will be displayed in the status line indicating 'Overstrike'.

Just like BASIC, ULTRA EDITOR will read the keyboard as upper and lowercase or just uppercase. The command line will show an 'L' if lowercase is enabled, or a 'U' if only uppercase is accepted.

<SHIFT/O> will toggle the upper/lower case modes.

For people writing structured code, an auto-indent feature has been included in the ULTRA EDITOR. When enabled, this feature permits a new line to be indented with the same number of spaces as the line before it. The feature only functions when <ENTER> is used to start a new line.

{I} toggles the auto-indent feature on and off. When enabled a '->' will be displayed an the status line.

18. Macro Commands:

One of the features which makes ULTRA EDITOR such a powerful editor is the ability to define your own sequence of keystrokes as a one-keystroke macro. These macro definitions can include not only standard text characters, they can also include cursor commands, Change or Jump commands and even call other macros. In sort, anything you can do from the keyboard can be duplicated in a macro.

{V} Define a macro. You will be asked which macro you wish to define. Answer with a number from 1 to 9. If you wish to edit an already defined macro, precede the number with an 'E' (ie. e9). If the 'E' option has been used, the existing macro will be displayed on line 2, otherwise a blank field will be displayed.

When typing in the macro definition a few changes to the normal editing methods have been made. First, macro definitions are always made in overstrike mode. This is since commands like delete, etc. no longer function (they can't function in define and still be permitted in the definition itself). Therefore the only editing commands available are the left and right arrow keys which move the cursor. If you make a mistake, move the cursor to the mistake and type over it. Sorry, but you can't insert or delete characters in the definition. Second, to exit the define mode you must press {V} again. This makes sense since you can't define a macro from within another macro -- and since {V} can't be used and some kind of exit key is required; what better key to use to signal the end of the macro. Note that the cursor can be at any position in the macro when the {V} is pressed and the entire line will be accepted. If you are editing an existing macro and decide that you don't want to make any changes after all, just use {BREAK} to exit -- no changes will be made to the existing definition in this case.

With the exception of {BREAK}, {V}, <LEFT ARROW> and <RIGHT ARROW> any other keystroke may be included as part of a macro definition. Unfortunately (due to hardware limitations in the CoCo 3) control keys will appear the same as normal keys, but they will function differently when called. For example, if you press either <J> or {J} a 'J' will appear. But if you call the macro, the first definition will cause a 'J' to appear in the text; the second will initiate a Jump command. Pay attention to the keys you press when defining the macro. Other keys (<ENTER>, <ARROWS>, etc.) will be displayed as "strange" characters.

A macro can only be 49 characters long. If you typed more than 49 characters in your definition, the macro will be truncated to the first 49 characters -- no error messages will be displayed.

Once you have defined your macros, you may save them for later use. Just use the <Save Macros> option in the main menu to save your macro definitions.

{1}...{9} Invokes a macro. Once you have defined your macro, pressing the <CTRL> key and the number key corresponding to the macro number invokes the defined macro.

Macros can be very useful -- but a few examples will get you started in finding uses of your own. Note: in these examples keystrokes generated by the <CTRL> key will be shown in the { } format, but don't include either { or } in your definition.

18.1 If you get tired of typing a particular word in your text, just set one of the macros to the word. For example, if you find yourself typing 'printf' a lot, just define macro '1' to this text. Now every time you type a {1} the word 'printf' will be typed.

18.2 Now for a more complex macro. Let's say you are using ULTRA EDITOR for writing assembly language source code and you like all your subroutines to start with a block which looks like this:

```
**************************************
*
* Subroutine name goes here
*
```

and you are tired of holding down the '*' key. You can define a macro to do the job for you:

```
**************************************<ENTER>*<ENTER>*
<ENTER>*<UP ARROW><SHIFT/RIGHT ARROW><SPACE>
```

In this macro, the first bunch of '*'s print the first line, the <ENTER> advances to the next line, another '*' is printed, the next <ENTER> advances to line 3, another '*' is printed, the last <ENTER> advances to line 4, and the last '*' is printed. Finally, the <UP ARROW> moves the cursor to line 3, <SHIFT/RIGHT ARROW> moves the cursor to the space after the '*' and the <SPACE> moves the cursor to the position where we want to start our text.

18.3 This one was used when preparing the help file for ULTRA EDITOR. The help text was typed in without the 'FCC's necessary for the assembler. A macro was defined in the following manner:

```
<F2>{LEFT ARROW}FCC<SPACE>"<SHIFT/RIGHT ARROW>",0<DOWN ARROW>
```

Each time this macro was invoked, the edit mode was switched to insert (with <F2>), the cursor was moved to the start of the line, the text 'FCC "' was typed, the cursor was moved to the end of the line, '",0"' was typed and the cursor was advanced to the next line.

18.4 If you are Jumping to a label a lot, you might want to define a macro as a Jump command.

{J} LABEL<ENTER>

will save lots of keystrokes. Now all you need to do is hit the appropriate control key.

18.5 Even though the keys repeat and you have many cursor keys, you may want to scroll through some text very quickly.

If you define {9} as '<UP ARROW>{9}' you'll initiate a rapid scan up through the text. Pressing {BREAK} will stop the cursor.

Macros can call each other or themselves. For example, if you have defined {1} as 'HELLO' and {2} as 'GOODBYE' you could define {3} as {1}{2}. Now when you press {3} 'HELLOGOODBYE' will be displayed. Another method is to have a macro called itself (called recursion). For example, you could define {1} as 'HELLO<ENTER>{1}'. Now when you press {1} the text 'HELLO' will be printed, the cursor will advance to the next line and macro 1 will be called and 'HELLO' will be printed again. This procedure will continue until an error is generated (probably memory full) or {BREAK} is pressed (which causes its own error).

Macros can be nested up to 20 levels deep. An attempt to nest deeper than this will cause an error.

19. Miscellaneous Commands:

By no means are any of these commands "miscellaneous" -- they just didn't seem to fit in any of the other categories in this section of the manual.

{Q} Quits the editor to ULTRA EDITOR's main menu. This is the only way out of ULTRA EDITOR, so don't forget about it.

{W} Toggles the display Width between 40 and 80 column modes.

<ENTER> This important command inserts a blank line below the current line. This corresponds to the normal method of entering text in most editors. <ENTER> has no effect on the data in the current line, this means that the cursor may be positioned at any place in the line (it doesn't have to be at the end).

{ENTER} This command will insert a carriage return at the current cursor position effectively splitting the line into two lines.

{;} The opposite of {ENTER}. This command is used to append the next line to the current line. This is the command which can be used to join lines together.

{P} Print a page. 55 lines of text, starting at the top of the screen (not the current cursor position) will be printed on the printer.

20. Errors:

ULTRA EDITOR can produce two types of errors. First, in the main menu, errors will be reported in the standard BASIC format. The only errors encountered here should be of the IO variety.

In the editor mode a number of different error messages may be encountered. In all cases the message is displayed on the command/status line. Pressing any key will cause the normal status line to reappear. Following are the possible errors:

20.1 BLOCK NOT DEFINED: You have attempted to execute a block command without first defining a block. Use the {<} and {>} commands to define a block.

20.2 BLOCK TOO LARGE FOR COPY OR MOVE: The defined block exceeds the transfer buffer's capacity. Use {S} to find the size of the block, then reduce the size of the block.

20.3 ATTEMPT TO COPY OVER DEFINED BLOCK: During a Move or Copy the cursor is within the defined block. You cannot copy/move a block onto itself.

20.4 OPERATION TOO LARGE FOR AVAILABLE MEMORY: You have run out of memory. This can occur when you have finished editing a line and attempted to move to another line, or during a block move or copy.

20.5 MARKER ALREADY EXISTS: You have attempted to insert a second block start or block end marker. Use the Jump command to find the first one and delete it with {K}.

20.6 CAN'T HAVE END BEFORE START: An attempt has been made to put a start block marker after an end block marker, or an end block marker before a start block marker.

20.7 FAIL, NO MATCH FOUND: The text searched for by a Jump, Change or Find command was not found.

20.8 LINE NUMBER TOO HIGH: The number entered for a Jump command is higher than the last line in the buffer.

20.9 UNABLE TO RESTORE LINE: In response to an Unkill command when there is no data in the buffer.

20.10 KEYBOARD ABORT: Whenever {BREAK} is pressed this error is generated.

20.11 MACRO NESTING TOO DEEP: Macros which call themselves have done so more than 20 times.

20.12 LONG LINE SPLIT: A line longer than 128 characters was split into two shorter lines after editing.

21. Making Modifications to the BASIC Driver:

The BASIC portion of ULTRA EDITOR can be modified, if you wish. However, you must be very careful since the majority of the program has been compressed to save memory. Four lines have been left in an uncompressed mode especially for your modifications:

Line 10 -- this contains the PALETTE commands to set the foreground and background colors for the editor. It is currently set to white on blue.

Line 20 -- this contains the PALETTE commands to set the foreground and background colors for the status lines in the editor. The choice of blue on white was used since it contrasts nicely with the edit screen.

Line 30 -- this line is currently just a remark. It can be used to set the printer baud rate, etc.

It is not recommended that other modifications to ULTRA EDITOR be attempted.

22. Some Additional Notes:

22.1 Entering text at the start of the file: Since you can't use the cursor keys to move the cursor in front of the first line in the buffer, how do you insert text in front of line 0? Position the cursor to the start of the buffer and insert a carriage return at the start of line 0 with {ENTER}. Now edit this blank line in the normal manner.

22.2 Entering text at the end of the file: When the editor is first entered, the cursor will be positioned at the 'END OF FILE' marker. You must insert a blank line to edit by pressing <ENTER> before any text can be entered.

22.3 Line numbers: ULTRA EDITOR does not number the lines in its buffers. It maintains a line count for display on the status line and for use by Jump, but the line numbers are not saved in the text files.

22.4 Files: All files generated by ULTRA EDITOR are in standard ASCII format. When saving, file block markers are skipped. When loading, characters below an ASCII SPACE (except for carriage returns) and greater than ASCII 127's are skipped.

22.5 Long lines: ULTRA EDITOR permits the entry of lines with a maximum length of 128 characters. It is, however possible to have lines in the buffer which are longer. This could be because the file read into the buffer contained long lines, or a Change command could alter a line's contents so that it becomes longer than 128 characters. These long lines will be saved during Saves, the first 128 characters will be displayed when viewing the text, and changes made to surrounding lines will NOT affect these long lines. When you edit a line which is longer than 128 characters, only the first 128 characters will be transferred to the edit buffer. Your editing commands will affect only these characters. When you insert a line into the buffer (<ENTER>, <UP ARROW>, <DOWN ARROW>, {J}, etc.) a 'long line split' message will appear on the status line. The screen will also reformat and the rest of the line will be displayed. This newly displayed line could still be longer than 128 characters.

1. Loading and Running the Editor:

The editor is first invoked by running CoCo-C's command coordinator. Type **RUN "CC <ENTER>**. After the menu appears, select **E** (for edit). This will load and execute the editor.

Once the editor is running you will see the copyright message and a prompt for a filename. If you wish to edit an existing file, input the name and it will be loaded. If you press <ENTER> you will go to a hi-res screen containing the text 'COMMAND' and some numbers at the top of it.

2. Options in the Command Mode:

When you first enter the editor you will be in the command mode. This is indicated by the status line at the top of the screen. The numbers at the right are the numbers of free bytes in the buffer and the current line number.

In this mode you have the following options available:

2.1 Cursor movement commands.

<UP ARROW> moves the cursor up a line.

<DOWN ARROW> moves the cursor down a line.

<P> Page up. This moves the cursor forward by 22 lines.

<O> Page Down. This moves the cursor backward by 22 lines (the key beside the Page key).

<*> STARt. This moves the cursor to the start of the text.

<=> End. This moves the cursor to the end of the text (the key beside STARt).

<(> Begin Block. This moves the cursor to the 'begin block' marker (the key resembles '<').

<)> End Block. This moves the cursor to the 'end block' marker (the key resembles '>').

<J> Jump. This powerful command first prompts you for a pattern to jump to. You may input either a word or a line number. If a number is input, then the cursor will move to that line number. If text is entered the program will search for a line number starting with that word(s). An exact word match must be found. If the search fails for a label or line number, the cursor will go to the end of the file. Note that a search for LABEL X will only find a line starting

with that exact text, ie: LABEL X XYZ. A line beginning with LABEL XX will not be matched.

<F> Find. When you select this option, you will be prompted for a search pattern (Find). The text will be searched from the current cursor position for a match. If no match is found the cursor will be positioned at the end of the text.

<B> Find Backwards. This option is the same as <F> except that the search will be made from the current position backwards. If no match is found the cursor will be positioned at the beginning of the text.

<N> Find Next. This will find the next occurrence of the pattern last used by <F> or <B>.

<L> Find Last. This will find the last (like <B> occurrence of the pattern last used by <F> or <B>).

2.2 Text editing commands:

<I> Insert. This option puts you in the insert mode. You can now insert a line at the current cursor position. When you have finished your line, press <ENTER>. To leave this mode and return to the command mode, press <BREAK>.

<E> Edit. This option will permit editing of the existing line at the cursor. After the edit is completed, press <ENTER>. If you decide that the changes you have made, shouldn't be made, press <BREAK> (instead of <ENTER>) and the line will be restored.

<K> Kill. This command will delete (Kill) the line at the cursor. For safety's sake, don't use the auto-repeat-key feature with this command. If you want to delete a number of lines, use the <D> command (see below).

<C> Global Change. This command will allow global changes. To use, first input the pattern to be changed and press <ENTER>. Now input the new pattern beside the 'TO:' prompt. The text will be searched from the current position until the pattern is found. It will then be changed and the cursor will be set to the start of the line changed.

<A> Change Again. This command (Again) will repeat the last change <C> command input. If you have a number of changes to make, just hold the key down. A combination of Find/Next and Change/Again make a prompted replace.

2.3 Block Commands:

'<' Begin Block. This inserts a 'begin block' marker at the current line position. Only one of these markers are allowed at one time.

'>' End Block. This inserts an 'end block' marker at the current line position. Only one of these markers are allowed at one time.

<M> Move Text. This will move the text marked with a 'begin' and 'end' marker to the current cursor position. This command will not work if both markers are not present or if an attempt is made to copy the block onto itself. When this command is used, you will notice some garbage on the screen -- this is normal. If an attempt is made to move more than 6000 bytes, an error message will be shown and the command will be aborted. The block markers will not be deleted when this command is used.

<R> Replicate. This command will replicate the text marked with a 'begin' and 'end' marker to the current cursor position. This command will will not work if both markers are not present or if the 'start' marker occurs after the 'end' marker or if an attempt is made to copy the block onto itself. The block markers will not be deleted when this command is used.

<D> Delete. This command will delete the text marked with a 'begin' and 'end' marker to the current cursor position. This command will will not work if both markers are not present or if the 'start' marker occurs after the 'end' marker or if attempt is made to copy the block onto itself. When this command is called, the cursor will first be moved to the start of the block and the size of the block will be shown at the top of the screen. To delete the block you must press the <Y> key. Pressing any other key will abort the delete operation. This command will also delete the block markers.

<Z> Kill Markers. This command will delete any block markers present in the text (there's no reason for the selection of 'Z' for this -- you'll just have to remember it).

2.4 Miscellaneous commands:

<S> Split. This is used to split a line into two. Position the cursor to the spot where you want an end-of-line marker placed and press <ENTER>.

<H> Help. In case you misplaced the manual, this command will give a summary of the commands. Press any key to show the next page and then to return back to the editor.

<Q> Quit. This will quit the editor and return you to the I/O menu selections. If you read a file when you first started, the first option will be to Save the file with the last filename. The next two options will let you either append a file or write the current buffer out with a new filename. After the file is saved, you will return to CoCo-C's command coordinator.

3. Text Editing Conventions:

Whenever text is being entered -- whether it's for inserting or editing a line or entering a search pattern -- you will be in special input mode. Some keys have been redefined from the normal BASIC arrangement.

<Left arrow>	moves the cursor left.
<Right arrow>	moves the cursor right.
<Up arrow>	produces the '^' character (ASCII caret).
<Down arrow>	produces the ']' character.
<Shift left arrow>	deletes the character at the cursor.
<Shift right arrow>	inserts a space at the cursor.
<Shift up arrow>	produces the '|' character.
<Shift down arrow>	produces the '[' character.
<Shift @>	produces the '\' character.
<Clear>	produces the '}' character.
<Shift clear>	produces the '{' character.

Whenever you are typing, you are in overstrike mode. This means that the character at the cursor will be changed to the key you press.

The <Shift 0> combination still turns the lower case on and off. If lower case is on, the cursor will flash slowly, when it's off the cursor flashes quickly.

4. Long Lines:

This editor is designed for a special purpose. In order to make it as fast and easy to use, certain trade offs have been made -- the most obvious will be the line length. It is not possible to input a line which extends past the right side of the screen. This means that when you are inputting lines, they can only be 51 characters long. This should be more than ample for most purposes -- any lines longer than this will be truncated in the program listing anyway.

It is possible to end up with lines longer than 51 characters -- you could use the Glue or Change command, or perhaps you are reading a file created by a different editor. In this case, only the first 51 characters will be shown. If you never edit the line, it will be saved in its long version when the text is saved. If you attempt to edit the line, you will only be allowed to edit the first 51 characters. After the line is edited an end-of-line marker will be placed in the 51th position and the line

will be split. This means that no characters will be lost. You'll be made aware of the split by the screen refresh which occurs.

5. File Formats:

All I/O is done in standard ASCII format. If you have an existing file which contains control characters (less than ASCII 32) or graphics characters (greater than ASCII 127), then the file should first be filtered. Use the following BASIC program:

```
10  OPEN  "I",1,"OLDFILE.DAT"
20  OPEN  "O",2,"NEWFILE.TXT"
30  IF  EOF(1)  THEN  100
40  LINE  INPUT  #1,A$
50  PRINT  #2,A$
60  GOTO  30
100  CLOSE:END
```

6. Wildcards:

On all searches, changes and jumps, a wildcard character is available. This is the '\', generated with the <Shift @> key.

Since all trailing spaces are truncated by the input routines, there has to be a way to enter search and change data with trailing blanks -- if you end a string input for FIND or CHANGE with the '\' the '\' will be changed to an end-of-line. For example, lets say you wish to change all occurrences of '\\' (two spaces) to '\' (one space). At the FROM: prompt, enter '\\\'. At the TO: prompt, enter '\\'.

7. I/O Conventions:

When a disk system is connected, the default filename extension will be '.TXT'. Whenever you are inputting a filename, you may type the word "DIR' for a directory of drive 0. Adding a drive number with a colon (ie. DIR:2) will give a directory of the specified drive. This does mean that you are not allowed a file called 'DIR.TXT'.

After a directory, just enter the filename you wish to read or write.

You will notice that the I/O portion of the program is in BASIC. This permits some modification on your part as well as ensuring compatibility with other ROMs. This program works with both cassette and disk. It works with standard Radio Shack DOSes 1.0 and 1.1 as well as JDOS and ADOS and EDOS. It should work with any other DOSes which are compatible with BASIC.

If you wish to return to the editor, press any other key, except <BREAK> -- this will return you to BASIC.

8. Errors:

Since the files are opened from BASIC, some errors of the IO and NE type may occur. If the do just type GOTO 9 <ENTER> to return to the command mode.

1. Compiler Specifications:

CoCo-C is a single pass optimizing compiler which compiles a subset of the C language as defined by Kernighan and Ritchie. The output it creates is a 6809 assembly language file designed to be assembled with the CoCo-ASM 6809 Assembler. The resulting output from the assembler is a binary file which then may be linked with the CoCo-C Library Linker. The Linker merges in the CoCo-C Function Library with the compiled program to create a stand-alone ML program.

CoCo-C supports the following:

1.1 Symbolic Names:

Symbolic names may consist of letters and digits and the underscore character ('_'). The first character in the symbol name must be a letter or '_'. The symbol name may be any length, but only the first eight characters are used. All lower-case letters are converted to upper-case as output to the assembler.

1.2 Constants:

The following constants are recognized by the compiler:

type:	*value:*	*example:*
decimal number	(0 - 65535)	`123`
hexadecimal number	(0x0 - 0xffff)	`0x1e7f`
quoted string	(string address)	`"sample string"`
character constant	(1 or 2 chars)	`'a' or 'Z' or 'ab'`

In addition to the above, the compiler recognizes the following "special" character constants:

type:	*value:*	*example:*
newline character	0x0a	`\n`
tab character	0x09	`\t`
backspace character	0x08	`\b`
formfeed character	0x0c	`\f`
octal value	(0 - 777)	`\123`

1.3 Data Type Declarations (variables):

The compiler supports two types of variables: integers and characters. Integers occupy a word in memory and characters, a byte. Both integers and characters are of the signed type.

examples:

```
char  c;              /* 8 bit character, signed */
int  i;               /* 16 bit integer, signed */
```

1.3.1 Global variables:

Variables declared outside a function are defined to be global or "static" because memory is permanently reserved for them. All global variables reside in the lower 32K of memory and each must have a unique symbolic name. Also, global variables must be initialized during the run-time of the executed C code.

1.3.2 Local variables:

Variables declared inside a function are defined to be local or "automatic", because they exist only during the execution of a function in which they are declared. When control leaves the function, they disappear. Local variables reside in the stack frame area of memory. The amount of local variable space is only limited to the size of available stack. Symbolic names for local variables need not be unique for each function. As with global variables, local variables must be initialized during run-time.

1.4 Arrays:

Single dimension (vector) arrays are supported and can be of type "char" or "int".

examples:

```
char  c[n];           /* character array */
int  i[n];            /* integer array */
```

1.5 Pointers:

Local and static pointers can contain the address of "char" or "int" data elements.

examples:

```
char  *c;             /* pointer to character */
int  *i;              /* pointer to integer */
```

1.6 Initializers:

Initializers are accepted for global declarations only. They consist of constants declaring arrays, strings or pointers. Initializers may be of type int or char. Initializers have the form:

type name = {init_list};

examples:

```
int  ia[3]  =  {1,2,3};        /* 3 element integer array   */
                               /* containing 1, 2, 3        */

int  ia[3]  =  1;              /* 3 element integer array   */
                               /* containing 1, 0, 0        */

char  ca[]  =  {'a',0};        /* 2 element character array */
                               /* containing ASCII 'a' & 0  */

char  ca[4]  =  "abc";         /* 4 element character array */
                               /* containing ASCII abc & 0  */

char  ca[]  =  "abc";          /* 4 element character array */
                               /* containing ASCII abc & 0  */

char  *cp  =  "name";          /* character ptr containing  */
                               /* address of "name"         */
```

1.7 Expressions:

CoCo-C supports the following expression operators:

1.7.1 unary operators:

		examples:
-	minus	`-i`
*	indirection	`*i`
&	address of	`&name`
++	increment, either prefix or postfix	`++i or i++`
--	decrement, either prefix of postfix	`--i or i--`

1.7.2 logical operators:

!	logical 'not'	`!a`
&&	logical 'and'	`a && b`
||	logical 'or'	`a || b`

1.7.3 binary operators:

		examples:
~	one's complement	`~a`
+	addition	`a + b`
-	subtraction	`a - b`
*	multiplication	`a * b`
/	division	`a / b`
%	modulo, i.e. remainder from division	`a % b`
|	inclusive 'or'	`a | b`
^	exclusive 'or'	`a ^ b`
&	logical 'and'	`a & b`
==	test for equal	`a == b`
!=	test for not equal	`a != b`
<	test for less than	`a < b`
<=	test for less than or equal to	`a <= b`
>	test for greater than	`a > b`
>=	test for greater than or equal to	`a >= b`
<<	arithmetic left shift	`a << b`
>>	arithmetic right shift	`a >> b`

1.7.4 assignment operators:

|=	logical 'or' and assign	`a |= b`
^=	exclusive 'or' and assign	`a ^= b`
&=	logical 'and' and assign	`a &= b`
+=	add and assign	`a += b`
-=	subtract and assign	`a -= b`
*=	multiply and assign	`a *= b`
/=	divide and assign	`a /= b`
%=	modulo and assign	`a %= b`
<<=	shift left and assign	`a <<= b`
>>=	shift right and assign	`a >>= b`

1.7.5 miscellaneous operators:

/* */	comment delimiter	`/* I'm a comment */`
" "	character string delimiter	`"Hello Everybody"`
' '	character constant delimiter	`'A'`
;	end of a statement	`a = 5;`
{ }	statement block delimiter	`{a = 5; b = 6;}`
( )	indicates function calls and sets priority in expressions	`func1()` `a = (3 - a) * b`
[ ]	indexes arrays	`a[5]`
:	label delimiter, CASE terminator, and conditional separator.	`case 1:`
?	conditional expression	`c = a < b ? a : b`

1.8 Functions:

Functions in CoCo-C are basically subroutines, containing optional arguments as input, which returns an integer value as output upon its exit. The value returned may be used in the subsequent evaluation of an expression. Furthermore, a function not requiring a returned value, may just simply "return".

CoCo-C functions have the form:

```
name (argument list, if any)
argument declaration, if any
{
    object-declarations and statements, if any
}
```

When a function is called, arguments are allocated on the stack in right to left order. While in function, local variables are also allocated on the stack. The declared arguments and variables become de-allocated when control leaves the function block.

function examples:

```
dummy()  {  }                   /* smallest function, does nothing */

getstat()                       /* function returning int value */
{                               /* (assume stat is global var)  */
   int a;                       /* int a is a local variable */
   a = stat & 0x7f;
    return(a);
 }

area(w,  l,  h)                 /* function w/3 args returning int */
int w,  l,  h;                  /* w, l & h are function arguments */
{
    return(w*l*h);
}
```

function call examples:

```
init_var();                     /* call w/no args or return value */
x  =  getstat();                /* call with return int in x */
i  =  area(7,  x,  y);          /* call w/3 args & return int in i */
```

1.9 Program Control:

CoCo-C supports the following control statements:

1.9.1 If - else statement

syntax:

```
if (expression)
    statement-1;
else
    statement-2;
```

example:

```
if (n > 0) {
  if (a > b)
    z = a;
  }
else
   z = b;
```

1.9.2 Else - if statement

syntax:

```
if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement
```

example:

```
if (n == 1)
  z = a;
else if (n == 2)
  z = b;
else if (n == 3)
  z = c;
else
  z = 0;
```

1.9.3 Switch statement

syntax:

```
switch (expression) {
    case const-expr:
      statements
    case const-expr:
      statements
    default:
      statements
}
```

example:

```
switch (ch) {
  case 'Y':
  case 'y':
    cptr = "yes";
    break;
  case 'N':
  case 'n':
    cptr = "no";
    break;
  default:
    cptr = "error";
}
```

1.9.4 While statement

syntax:

```
while (expression)
    statement;
```

example:

```
while((c=getch())  !=  '\n')
  putch(c);
```

1.9.5 Do - while statement

syntax:

```
do statement
    while (expression);
```

example:

```
do {
   c = getch();
   putch(c);
}
while(c != '\n');
```

1.9.6 For statement

syntax:

```
for (expr1 ; expr2 ; expr3)
  statement;
```

example:

```
for (n = 0; n < k; n++)
{
   a[n] = 0;
}
```

1.9.7 Break statement

syntax:

```
break;
```

example:

```
for (n = 0; n < 10; n++)
{
   if (n == 5)
      break;
   a[n] = 0;
}
```

1.9.8 Continue statement

syntax:

```
continue;
```

example:

```
for (n = 0; n < 10; n++)
{
   if (n != 5)
      continue;
   a[n] = 0;
}
```

1.9.9 Goto statement

syntax:

goto label;

label: statement

example:

```
for (j = 0; j < 10; j++)
{
  for (k = 0; k < c; k++)
  {
    if (k > c)
       goto report_err;
    count++;
  }
}

report_err:
  printf("Error  Found\n");
  exit();
```

1.9.10 Return statement

syntax:

return expression-list;

example:

```
if (error)
  return (-1);
```

1.9.11 Semicolon statement

syntax:

;

example:

```
/* infinite loop */
for (;;) ;
```

1.10 Preprocessor Commands:

CoCo-C has a built in preprocessor which allows simple macro substitutions, conditional compilation, and inclusion of text from other files. The following preprocessor commands are supported:

1.10.1 #define <name> <string>

Preprocessor will replace name by string throughout text.

1.10.2 #include <filename>

Allows program to include other files within its compilation. Optional quotes (" ") surround filename.

note: #includes may not be nested

1.10.3 #ifdef <name>

Allows the following lines (up to #else or #endif) to be processed, providing that name is defined.

1.10.4 #ifndef <name>

Allows the following lines (up to #else or #endif) to be processed, providing that name is NOT defined.

note: #ifdef's and #ifndefs may be nested

1.10.5 #else

Allows the following lines (up to #endif) to be processed, providing that the preceding #ifdef or #ifndef was false.

1.10.6 #endif

Terminates #ifdef and #ifndef.

1.10.7 #asm (not supported by standard C)

Allows all code to be passed unchanged directly to the target assembler, enabling "in-line" assembly language within the C program.

1.11 Miscellaneous:

Constants are evaluated by the compiler. As example, the line of code:

```
x = 1+2;
```

would be evaluated as:

```
x = 3;
```

by the compiler.

During the evaluation of an expression, any undeclared name is assumed to be a function.

Function calls are defined as any primary followed by an open parenthesis, so legal forms include:

```
variable();
array[expression]();
constant();
function()();
```

Pointer arithmetic takes into account the data type of the destination (ie. pointer++ will increment by two if pointer was declared "int *pointer").

Pointer compares generate unsigned compares (since addresses are not signed numbers).

By default, the compiler generates relocatable code. Code, literals and global variables are in one contiguous section of memory.

Generated code can be selected as absolute (i.e. the code may be placed in Read Only Memory). Code, literals, and variables are kept in separate sections of memory.

The generated code is re-entrant. Every time a function is called, its local variables refer to a new stack frame. By way of example, the compiler uses recursive-descent for most of its parsing, which relies heavily on re-entrant (recursive) functions.

2. Assembly Language Interface:

The CoCo-C Compiler allows the possibility of mixing assembly language with your C program. This gives total flexibility within the user program, allowing you to custom tailor sections of code requiring maximum speed or efficiency. Advanced users may want to create a custom interrupt service routine, which normally would be impossible without assembly language.

Interfacing to assembly language is relatively straight forward. The "#asm ... #endasm" construct allows the user to place assembly language code directly into the control context. Since it is considered by the compiler to be a single statement, it may appear in such forms as:

```
while (1) #asm ... #endasm

or
if (expression) #asm
...
...
#endasm

else statement;
```

Note a semicolon is not required after the #endasm since the end of context is obvious to the compiler. Assembly language code within the "#asm... #endasm" context has access to all global symbols and functions by name. It is up to the programmer to know the data type of the symbol (whether "char" or "int" implies a byte access or a word access).

Local variables and arguments are passed from C to assembler by placing them onto the 6809's machine stack. The compiler places the arguments in right to left order before a branch to subroutine is made.

As an example the function:

```
setclock(h,  m,  s);
```

would set up variable 's' as the first argument and variable 'h' as the third argument. To retrieve the arguments you must index into the stack to get the desired variable:

```
#asm
?SETCLOCK
 LDD  2,S    * get first argument  (s), A = hi byte, B = lo byte
 LDX  4,S    * get second argument (m)
 LDU  6,S    * get third argument  (h)
      .
      .
 RTS
#endasm
```

To pass variables from assembler to C, simply return the value in register pair A,B:

```
c = inchar();      /* get byte from CoCo Keyboard */

#asm
?INCHAR
 JSR  [$A000]      * check keyboard for character
 BEQ  ?INCHAR      * keep trying until available
 TFR  A,B          * move char to B
 CLRA              * clear out hi byte
 RTS
#endasm
```

It is also important to note that a "?" must precede the subroutine name that is being called from a C function. This is because the compiler automatically appends a "?" to all defined functions .

Since everything between #asm to #endasm is passed straight through to the assembler, it is also possible to control the assembler's directives within C. As an example, you could insert the name of the C file into the .LST output file from the assembler by doing the following:

```
#asm
NAME  TEST.C
#endasm
```

This would pass to the assembler as a directive, and the assembler would place the name "TEST.C" at the top of each page in the listing.

3. Stack Frame:

CoCo-C uses the 6809's machine's stack to pass function arguments and also to allocate space for local variables. This "stack frame" resides in CoCo's lower 32K of RAM and grows downward as variables use its space. Function arguments are pushed onto the stack as they are encountered between parentheses. During the function call, the sender's return address is also pushed on the stack.

Local variables allocate as much stack space as is needed, and are then assigned the current value of the stack pointer (after the allocation) as their address.

```
int  x;
```

will produce

```
LEAS  -2,S
```

which merely allocates room on the stack for 2 bytes (not initialized to any value). References to the local variable 'x' will now be made to the stack pointer + 0. If another declaration is made:

```
char  array[3];
```

the code would be

```
LEAS  -3,S
```

Array [0] would be at SP + 0, array [1] would be at SP + 1, array [2] would be at SP + 2, and 'x' would now be at SP + 3.

Thus, assembly language code using "#asm...#endasm" cannot access local variables by name, but must know how many intervening bytes have been allocated between the declaration of the variable and its use.

It is worth pointing out local declarations allocate only as much stack space as is required, including an odd number of bytes, whereas function arguments always consist of two bytes each. In the event the argument was type "char" (8 bits), the most significant byte of the 2-byte value is a sign-extension of the lower byte.

4. Function Library:

CoCo-C uses a relocatable function library for all its utility, I/O, and RSDOS functions. This library is in a separate binary file called "CLIB.BIN". This library is intended to get linked in with the compiled C program. In order for the C program to access any library function, the function name along with its relative entry point must be included in the C file. The file that contains the run-time library entry table is called "CLIB.INC". By default, this file is automatically "included" at the end of the .ASM output during the compilation of a C file.

CoCo-C has a special function within its library called "bascmd". This function allows you to mix BASIC commands within your C program. Many of the commands for the CoCo 1, 2 and 3 are supported!

For a detailed description of all the library functions contained within CoCo-C, refer to the section: CoCo-C Library Functions.

5. Start-Up / Initialization (CSTART. C):

Initialization must take place before for program enters main(). This is done by including "CSTART.C" at the beginning of your C file.

CSTART.C is a user configurable C and assembly language file that contains (at minimum) the beginning address for code, the beginning address for data, and the top address of the stack frame. Also, CSTART.C contains subroutine calls to initialize the interface between RSDOS and BASIC as well as setting up error trapping parameters. Conditional #defines are used within the start-up file to enable special user options during the compile. As example, it is not necessary to modify CSTART.C to enable error trapping; a simple #define ERRTRAP is all that is necessary.

CSTART.C also contains CoCo-C's relocatable run-time library. Unlike the function library, the run-time library contains the low-level logical and math routines necessary for the generated output code. The run-time library is in ASCII hex and the compiler's generated code makes calls to these routines by address. Therefore, the run-time code should NOT be modified for any reason.

6. Register Usage:

The Compiler uses the following registers in the 6809:

U	- argument count	
B	- primary register	(lo byte)
A	- primary register	(hi byte)
X	- secondary register	
Y	- temporary register	

These registers may be "borrowed" by a user's assembly language application, but they must be restored to their original values before re-entering C. This is a must for any interrupt service routines.

7. Special Functions:

There are several special functions within CoCo-C which do not contain a fixed number of arguments. They ones that apply are:

printf(), fprintf(), sprintf(), scanf(), fscanf(), and sscanf().

These functions rely on the compiler to count the number of arguments before control is passed to that function. Each time such function is called, the number of arguments is loaded in register 'U' and then sent to that function where it can be evaluated.

The function names listed above are reserved by the compiler and should not be re-defined.

8. Special Defines:

CoCo-C contains several special defines which are used as compiler option selectors. These defines direct the compiler to either conditionally compile certain sections of the start-up code and/or change its default generated output. The defines that apply are:

```
#define  ABSOLUTE     /* create absolute code for ROM-PAK */
#define  ERRTRAP      /* enable RSDOS error trapping */
#define  NOCLIB       /* do not use function library */
```

Due to the nature of these defines, it is necessary to declare these at the very beginning of your C file (before the #includes).

9. Using the Compiler:

The compiler is first invoked by running CoCo-C's command coordinator. Type **RUN "CC <ENTER>**. After the main menu appears, select **C** (for compile). The compiler will automatically load and execute, and request the following:

```
Options ? _
```

One or more compiler options may be selected at this time. Each option contains one character. Spaces or commas may be used as delimiters. After selecting the desired option(s) hit **<ENTER>**. If no options are needed, just hit **<ENTER>**.

Compiler Options:

```
I  = Include Source in Output
M  = Monitor Source on Screen
A  = Alarm on Error
P  = Pause on Error
```

Entering a **? <ENTER>** will display the options if you forget what they are.

Next the compiler will ask you for the output file:

```
Output File ? _
```

Enter any valid file name as your assembly language output file. Examples:

TEST.ASM	*create output file "TEST.ASM" on default drive*
MYFILE.ASM:2	*create output file "MYFILE.ASM" on drive 2*

If a drive designator is not specified, the output file will be created on the default drive. If no output file is required, just hit **<ENTER>**. This will cause the output to default to the screen.

Next the compiler will ask you for the input file:

```
Input File ? _
```

Enter the name of your C source file. Examples:

TEST.C	*read input file "TEST.C" from default drive*
MYFILE.C:2	*read input file "MYFILE.C" from drive 2*

After the <ENTER> key is pressed, the compiler will begin compiling your code. Any errors will be displayed on the screen. The compiler will pause on error if the P option was selected. Hitting <ENTER> resumes the compile, while the <BREAK> key terminates the compile.

After the compilation of your C file, the compiler will request you for another input file. This feature gives you an additional way to *include* other source files. If no other source files are to be included, just hit **<ENTER>**.

When the compile has completed, and assuming there were no errors in your C program, the compiler will respond with:

```
No Errors Found
Hit any Key to Return to Menu
```

At this time, hitting any key will return you to CoCo-C's command coordinator.

The next step would be to assemble the ASM file the compiler created. This is explained in the CoCo-ASM section of this manual.

Note: Hitting the <BREAK> key at any time while the compiler is running will terminate the compile and return you to the command coordinator.

10. Error Trapping:

Many functions within Co-Co-C's library (CLIB.BIN) support disk I/O (ie. fopen(), fputc(), fgetc(), kill(), etc. If an error occurs during the call of such function, the program will exit and return to BASIC with an error message (ie. NE ERROR for file not found). The same is true for any error during a bascmd() function call. Like BASIC, this is the default standard for CoCo-C.

As an option, the compiler provides a method of "trapping" errors so program control does not go to BASIC during an error. By defining the following:

```
#define  ERRTRAP
```

at the beginning of your C program (before the #includes), will instruct the compiler to enable error trapping within your program. Since most of the disk I/O functions return their own error codes, your program can now decipher whether or not an error has occured.

It is advisable to use error trapping only on fully debugged programs, failure to do so may cause unpredictable results.

11. Creating ROM-able Code:

With CoCo-C it is possible to create programs suitable for EPROMs. Not only is this useful for applications using the CoCo's ROM-PAK, but also for embedded applications using a stand-alone 6809 single board computer.

By default, CoCo-C produces complete position independent (relocatable) code. This means that the code, literals, and global variables are all "inline" within one contiguous section of memory. This enables your program to load and execute in any allowable space within the lower 32K of memory. Programs for EPROMS, on the other hand, have different requirements. Since a program for a ROM ends up in Read Only Memory, the code and variable areas must be placed in separate locations or *segments*. By defining the following:

```
#define  ABSOLUTE
```

at the beginning of your C program (before the #includes), will instruct the compiler to produce absolute code rather than relocatable code. Separate origins will be created for both the code and the variables. During the compile, the code will be placed in the code segment and all global variables will be placed in the data segment. After assembling and linking, the resulting code could then be programmed into an EPROM for whatever application.

The file CSTART.C contains the starting locations for both the code and data segments. This file may be modified to suit your application. The label "CSEG" declares the code segment and the label "DSEG" declares the data segment. As example, for the CoCo ROM-PAK you may want to change the file to this:

```
CSEG    EQU  $C000          * ROM-PAK address
DSEG    EQU  $1000          * variables begin
```

Your compiled program would then have an absolute starting address at $C000, and the variables would begin at location $1000. The resulting binary file could not be relocated.

For testing purposes, you may want to "CSEG" your program at a location within the lower 32K of memory. Once your program is fully debugged, all you have to do is change CSEG to the starting address of the EPROM and re-compile your program.

12. Error Messages:

When CoCo-C detects an error, it displays as output the line that caused the error. An arrow consisting the character "↑" is displayed beneath the line, containing a descriptive error message. If the P option was selected at the time of compile, the program will pause and resume upon entry of the <RETURN> key.

CoCo-C produces the following error messages:

12.1 bad label

A goto statement has an improperly formed label. Either it does not conform to the C naming conventions, or its missing altogether.

12.2 can't subscript

A subscript is associated with something which is neither a pointer nor an array.

12.3 cannot assign to pointer

An initializer consisting of a constant or a constant expression is associated with a pointer. Integer pointers do not take initializers, and character pointers take only expression-list or string-constant initializers.

12.4 global symbol table overflow

The global-symbol table has overflowed due to exceeding the number of global symbols allowable by the compiler.

12.5 illegal address

The address operator is applied to something which is neither a variable, a pointer (subscripted or unsubscripted), nor a subscripted array name.

12.6 illegal argument name

A name in the format argument list of a function declarator does not conform to the C naming conventions.

12.7 illegal function or declaration

After preprocessing a line, the compiler found something at the global level which is not a function or object declaration.

12.8 illegal symbol

The compiler found a symbol which does not conform to the C naming convention.

12.9 invalid expression

An expression contains a primary which is neither a constant, a string constant, nor a valid C name.

12.10 line too long

A source line after preprocessing is more than 128 characters long. This can be corrected by breaking the line into two or more parts.

12.11 literal queue overflow

A string constant has overflowed the compiler's literal queue. The literal queue is a buffer where the compiler stashes away string constants until the end of a function is reached.

12.12 local symbol table overflow

The local-symbol table has overflowed due to exceeding the number of local symbols allowable by the compiler. This error may be caused by a single function, since that the local symbol table is cleared after use of each function.

12.13 macro name table full

Too many #define commands has overflowed the macro name table.

12.14 macro string queue full

A #define command has overflowed the macro string queue. The macro string queue is a buffer for the replacement strings associated with macro names.

12.15 missing token

The syntax requires a particular token which is missing.

12.16 multiple defaults

A switch contains more than one default prefix.

12.17 must assign to char pointer or array

A string-constant initializer is applied to something other than a character pointer or character array.

12.18 must be constant expression

Something other than a constant expression was found where the syntax requires a constant expression.

12.19 must be lvalue

Something other than an lvalue is used as a receiving field in an expression. Attempting to assign a value to a constant or an unsubscripted array name will cause this error.

12.20 must declare first in block

A local declaration occurs after the first statement in a block.

12.21 negative size illegal

An array is dimensioned to a negative value.

12.22 no apostrophe

A character constant lacks its terminating apostrophe.

12.23 no closing bracket

The end of the input was reached at a point within the body of a function.

12.24 no comma

An argument list or declaration list lacks a separating comma.

12.25 no final }

The end of the input occured while inside of a compound statement.

12.26 no matching #if . . .

A #else or #endif is not preceded by a corresponding #ifdef or #ifndef command.

12.27 no open paren

An apparent function declarator lacks the left parenthesis which introduces the format argument list.

12.28 no quote

A string constant lacks its terminating double quote. Note that string constants cannot be continued from one line to the next, so the terminating quotation mark must be on the same line as the initial quotation mark.

12.29 no semicolon

A semicolon does not appear where the syntax requires one.

12.30 not a label

The name following the keyword goto is defined, but not as a label.

12.31 not allowed with block-locals

A goto statement occurs within a function which has local declarations at a level lower than the function header. CoCo-C does not handle this situation.

12.32 not allowed with goto

A local declaration occurs at a level below the body of a function which contains goto statements. CoCo-C does not handle this situation.

12.33 not allowed in switch

A local declaration occurs within the body of a switch statement. CoCo-C does not allow this.

12.34 not an argument

The names in the formal-argument list of a function header do not match the corresponding type declarations.

12.35 not in switch

A case or default prefix occurs outside of a switch statement.

12.36 open error on filename

The output file or an input file cannot be opened.

12.37 open failure on include file

A file named in a #include command cannot be opened.

12.38 out of context

A break statement is not located within a do, for, while, or switch statement, or a continue is not within a do, for, or while statement.

12.39 output error

An error occured while writing to the output file. This would indicate an I/O error, a write-protected disk, or insufficient space on the disk.

12.40 staging buffer overflow

The code generated by an expression exceeds the size of the staging buffer. The staging buffer temporarily holds all of the code generated by an expression. This situation can be corrected by breaking the expression into several intermediate expressions.

12.41 too many active loops

The level of nesting of any combination of do, for, while, and switch statements exceeds the capacity of the while queue to track loop-back and terminal labels.

12.42 too many cases

The number of case prefixes in a switch exceeds the capacity of the switch table.

12.43 wrong number of arguments

One or more of the formal arguments in a function header was not typed in before entering the body of a function.

abs

Format:

abs (nbr) int nbr;

Description:

The abs function returns the absolute value of 'nbr'. If 'nbr' is a positive value, it is returned unchanged. If negative, the negated value is returned.

Returns:

The absolute value of 'nbr'.

Examples:

```
offset  =  abs(voltage1  -  voltage2);
```

atoi

Format:

atoi (str) char *str;

Description:

The atoi function converts the decimal number represented by the string at 'str' to an integer and returns its value. Leading white space is skipped, and an optional sign (+ or -) may precede the leftmost digit. The first nonnumeric character terminates the conversion.

Returns:

Integer equivalent of 'str'.

Examples:

```
number  =  atoi("1234");
```

atoib

Format:

atoib (str, base) char *str; int base;

Description:

The atoib function converts the unsigned integer of base 'base' represented by the string at 'str' to an integer and returns its value. Leading white space is skipped. The first nonnumeric character terminates the conversion.

Returns:

Integer equivalent of 'str' within 'base'.

Examples:

```
number = atoib("17FF",16);
```

bascmd

Format:

```
#include "BASIC.H"
bascmd (cmd, str) int cmd; char *str;
```

Description:

The bascmd function allows a selected set of BASIC commands to be sent to the CoCo's BASIC interpreter. This includes control, I/O, disk, and graphics commands. The compiler and function library "pre-tokenizes" the command before sending it to BASIC. This virtually operates the same way when a command is entered at the BASIC command prompt (ie. DIR).

The bascmd function *always* requires two arguments; the command 'cmd' and the parameter string pointed to by 'str'. If no parameters are required, a NULL parameter must be used. The file "BASIC.H" contains the selected list of commands and the parameter string is whatever is necessary to follow the command (ie. AUDIO, "ON").

Special consideration must be taken when using graphic commands. Since CoCo graphics require reserved memory at specific locations, memory and/or stack conflicts can arise if the C startup file (CSTART.C) and your C program are not set up properly. Also the proper PCLEAR command may be required before loading your program (ie. PCLEAR 8).

bascmd

The following is the list of BASIC commands acceptable by bascmd:

All CoCo's:		CoCo 3 Only:
RUN	SCREEN	WIDTH
SET	COLOR	PALETTE
RESET	CIRCLE	HSCREEN
CLS	PAINT	HCLS
MOTOR	DRAW	HCOLOR
SOUND	PCOPY	HPAINT
AUDIO	PMODE	HCIRCLE
EXEC	PLAY	HLINE
LINE	DIR	HPRINT
PCLS	DRIVE	HSET
PSET	VERIFY	HRESET
PRESET		HDRAW
		ATTR
		HPOINT

Since nested quotes are not allowed in CoCo-C, a parameter string that requires quotes may substitute the ASCII octal equivalent (\042).

Octal substitutions are also required for any special BASIC tokens that may be needed in the parameter string. Use the following substitutions for these tokens:

BASIC token:	Octal Substitute:
+	\253
-	\254
*	\255
/	\256
TO	\245
PSET	\275
PRESET	\276

Returns:

ERR if BASIC error.

Examples:

```
bascmd(RUN,   "GAME");
bascmd(CLS,   "2");
bascmd(WIDTH,  "32");
bascmd(PCOPY,  "4  \245  3");          /* PCOPY 4 TO 3 */
bascmd(LINE,   "\254(191,0),\275"); /* LINE -(191,0), PSET */
bascmd(DRAW,   "\042BM128,96;U25;R25;D25;L25\042");
```

cls

Format:

```
cls ( ) ;
```

Description:

The cls function clears the CoCo's text screen. This function operates the same as BASIC's CLS command. No parameters are passed to this function.

Returns:

Nothing

Examples:

```
cls();
```

cmp

Format:

```
cmp ( );
```

Description:

The cmp function puts the CoCo 3 in composite video mode. This function operates the same as BASIC's CMP command.

Note: CoCo 3 ONLY.
See also rgb.

Returns:

Nothing

Examples:

```
cmp();                          /* put CoCo 3 in composite mode */
```

coco2

Format:

coco2 ( );

Description:

The coco2 function puts the CoCo 3 in a pseudo CoCo 2 mode (a CoCo 3 can never *exactly* emulate a CoCo2). All CoCo 3 BASIC commands are disabled. This function should only be called when the CoCo is in 32 column mode.

Note: CoCo 3 ONLY.

Returns:

Nothing

Examples:

```
bascmd(WIDTH,  "32");      /* put CoCo 3 in 32 column */
coco2();                   /*   and in CoCo 2 mode */
```

coco3

Format:

coco3 ( );

Description:

The coco3 function puts the CoCo 3 back in CoCo 3 mode. This function should not be called if the Super Extended BASIC memory has been overwritten.

Note: CoCo 3 ONLY.

Returns:

Nothing

Examples:

```
coco3();                   /* Ensure CoCo 3 mode */
```

cursor

Format:

cursor (pos) int pos;

Description:

The cursor function positions the CoCo's text screen cursor by the value of 'pos'. The CoCo must be in 32 column mode and 'pos' must be in the range from 0 - 511.

Returns:

ERR if CoCo is not in 32 column mode or if 'pos' is out of range.

Examples:

```
cursor(72);             /* set cursor on 2nd line, 8th char */
cursor(32*4+15);        /* set cursor on 4th line, 15th char */
```

dtoi

Format:

dtoi (str, nbr) char *str; int *nbr;

Description:

The dtoi function converts the signed decimal number in the character string at 'str' to an integer at 'nbr' and returns the length of the numeric field found. Conversion stops at the end of the string or upon any illegal numeric character. With 16-bit integers, dtoi will use a leading sign and at most five digits.

Returns:

Field length of 'nbr'.
ERR on error.

Examples:

```
length  =  dtoi("5678",&n);
```

exit

Format:

exit ( );

Description:

The exit function provides a way to terminate a program. This function may be placed anywhere within the program. When the function is called, program control returns to BASIC.

Returns:

Does not return.

Examples:

```
if((c=getchar())  ==  BREAK)     /* get next character */
   exit();                       /*   and exit if BREAK key */
```

fast

Format:

fast ( );

Description:

The fast function puts the CoCo 3 in high speed mode. This function has the same effect as POKE &HFFD9, 0 in BASIC.

Note:

CoCo 3 ONLY; See also slow.

Returns:

Nothing

Examples:

```
fast();                          /* switch to hi-speed mode */
```

fclose

Format:

```
#include "STDIO.H"
fclose (fd)  int fd;
```

Description:

The fclose function closes the file associated with the file descriptor 'fd'. Any data in the buffer of an output file is written to before the file is closed.

Returns:

ERR if unsuccessful.

Examples:

```
fclose(fd);                          /* close file fd */
```

fclosall

Format:

```
#include "STDIO.H"
fclosall ( );
```

Description:

The fclosall function closes all open files. All open file buffers are written to their associated files before closing.

Returns:

ERR if unsuccessful.

Note:

All open files are automatically closed when a program terminates normally.

Examples:

```
fclosall();                          /* close all open files */
```

fgetc

Format:

```
#include "STDIO.H"
fgetc (fd);
```

Description:

The fgetc function reads the next character from the file associated with file descriptor 'fd'. The character returned is a positive integer in the range of 0 to 255. A returned value of EOF indicates that either the end-of-file has been reached or that an error has occurred. This function is equivalent to the getc function.

Returns:

Integer (0 - 255).
EOF if end of file or error.

Examples:

```
while((c=fgetc(fd))  !=  EOF)   /* print all chars in file */
   putchar(c);
```

fgets

Format:

```
#include "STDIO.H"
fgets (buffer, nbr, fd);  char *buffer;  int nbr, fd;
```

Description:

The fgets function reads a string of characters from the file associated with file descriptor 'fd' and places them into the specified character buffer until a newline character is encountered. The 'nbr' argument specifies the maximum number of characters to be read (nbr-1). The newline character is kept with an appending NULL character.

Returns:

Pointer to 'buffer' if successful.
NULL if end-of-file occurs.

fgets

Examples:

```
while(fgets(string, 80, infile) != NULL)
    fputs(string, stdout);      /* print all lines in file */
```

fopen

Format:

```
#include "STDIO.H"
fopen (fname, mode)  char *fname, *mode;
```

Description:

The fopen function opens a file with the name specified by 'fname' and the type of file access specified by 'mode'. If the attempt to open a file is successful, fopen returns a file descripter value (integer) for the open file; otherwise it returns a NULL.

The 'fname' argument is a string containing the name of the file with an optional drive number. The 'mode' argument is a string having one of the following values:

```
"r"   - open text for reading
"w"   - open text for writing
```

The file must already exist if the 'mode' string contains an "r". Otherwise fopen returns a NULL. The file need not exist if the 'mode' string contains a "w". If the file does not exist, a new file is created. If the file does exist, and 'mode' contains a "w", the files contents are erased.

Returns:

File pointer associated with 'fname'.
NULL if 'fname' could not be opened.

Examples:

```
fd=fopen("OUTFILE.DAT",  "w");

if((fd=fopen("INFILE.DAT:2",  "r"))  ==  NULL)
    printf("could  not  open  INFILE.DAT\n");
```

fprintf

Format:

```
#include "STDIO.H"
fprintf (fd, str, arg1, arg2,...)  int fd; char *str;
```

Description:

The fprintf function performs a formatted print to the file associated with the file descriptor 'fd'. The format of the output is controlled by the 'str' string. Following the 'str' string is an optional list of values to be written to the file. With the exception of the additional 'fd' argument, the fprintf function is identical to the printf function. The function call printf (str, arg1, arg2,...) is equivalent to fprintf (stdout, str, arg1, arg2,...).

Returns:

Count of total characters written if successful.
ERR if an error occurs.

Note:

Also see printf for a description of the format string.

Examples:

```
i = fprintf(fd,"Name: %s, Age: %d\n", "Mr Magoo", 97);
if(i == ERR)
   fprintf(stdout, "Error writing to file\n");
else
   fprintf(stdout, "%d characters written\n", i);
```

fputc

Format:

```
#include "STDIO.H"
fputc (c, fd)  char c; int fd;
```

Description:

The fputc function writes the character 'c' to the file associated with the file descriptor 'fd'. The fputc function is similar to the putc function. The only difference is that the fputc function is a function rather than a macro.

fputc

Returns:

c if successful.
EOF if an error occurs.

Examples:

```
if(fputc('A',  fd)  ==  EOF)
   printf("Error  writing  to  file\n");
```

fputs

Format:

#include "STDIO.H"
fputs (str, fd) char *str; int fd;

Description:

The fputs function writes a string of characters 'str' to the file associated with the file descriptor 'fd'. Each successive character is written to the file until a null character is found. The null character is *not* written, and a newline character is *not* appended.

Returns:

EOF if an error occurs.

Examples:

```
char  customer[]="Joe  Schmoe";
if(fputs(customer,  fd)  ==  EOF)
   printf("Error  writing  to  file\n");
```

fscanf

Format:

```
#include "STDIO.H"
fscanf (fd, str, arg1, arg2,...)  int fd; char *str;
```

Description:

The fscanf function reads formatted input from the file associated with the file descriptor 'fd'. The format of the input is controlled by the 'str' string. Following the 'str' string is an optional list of variable addresses where the input data is stored. With the exception of the additional 'fd' argument, the fscanf function is identical to the scanf function. The function call scanf (str, arg1, arg2,...) is equivalent to fscanf (stdin, str, arg1, arg2,...).

Returns:

Number of fields read if successful.
EOF if end of file or an error occurs.

Note:

Also see scanf for a description of the format string.

Examples:

```
if(fscanf(fd, "%c", &c) == EOF)
   printf("Error reading file");

if(fscanf(fd, "%80s", s) == EOF)
   printf("Error reading file");

count=(fscanf(fd, "%d %d", &x, &y));
if(count == EOF)
   printf("Error reading file");
else
   printf("%d fields read successfully\n", count);
```

getc

Format:

```
#include "STDIO.H"
getc (fd);
```

Description:

The getc function reads the next character from the file associated with file descriptor 'fd'. The character returned is a positive integer in the range of 0 to 255. A returned value of EOF indicates that either the end-of-file has been reached or that an error has occurred. This function is equivalent to the fgetc function.

Returns:

Integer (0 - 255)
EOF if end of file or error.

Examples:

```
while((c=getc(fd))  !=  EOF)    /* print all chars in file */
    putchar(c);
```

getch

Format:

```
#include "STDIO.H"
getch ( );
```

Description:

The getch function gets a single character from the standard input and returns it as a positive integer in the range of 0 to 255. The character returned is as exactly as it is read (no filtering), and it is *not* echoed to the screen.

Returns:

Integer (0 - 255)

Examples:

```
while(getch()  !=  '\r');        /* Wait for return key */
```

getchar

Format:

```
#include "STDIO.H"
getchar ( );
```

Description:

The getchar function gets a single character from the standard input and returns it as a positive integer in the range of 0 to 127. If the character read is greater than 127, the MSB is set to zero to maintain ASCII compatibility. This function echoes the character to the screen.

Returns:

Integer (0 - 127)

Examples:

```
i=getchar();
printf("You entered the %c character", i);
```

getcurs

Format:

```
getcurs ( );
```

Description:

The getcurs function returns the current cursor position of the CoCo's text screen. It will work with all standard CoCo text screen formats (32, 40, or 80 column). The upper left corner of the screen is position 0.

Returns:

Cursor position if using standard text screen.
ERR if not using standard text screen.

Examples:

```
pos=getcurs();
printf("The end of line is at position %d\n", pos);
```

getftyp

Format:

```
#include "STDIO.H"
getftyp ( );
```

Description:

The getftyp function returns the filetype of the *last* opened file.

Returns:

0 if the file is BASIC
1 if the file is ASCII
2 if the file is BINARY

Examples:

```
if(getftyp()  !=  ASCII)
   printf("Not  an  ASCII  file\n");
```

getkey

Format:

```
getkey ( );
```

Description:

The getkey function checks the CoCo's keyboard for a keypress. If a key is currently pressed, a positive integer in the range of 0 to 255 is returned. If no key is pressed, EOF is returned. The getkey function does not wait for a key to be pressed.

Returns:

Integer (0 - 255) if key pressed.
EOF if no key pressed.

Examples:

```
while(getkey()  ==  EOF);          /* Wait for any key */
```

gets

Format:

```
#include "STDIO.H"
gets (str)  char *str;
```

Description:

The gets function reads a string of characters from the standard input and places them into the specified character buffer until a newline character is encountered. The newline character is replaced with a NULL character in the output buffer.

Returns:

Pointer to 'str'.

Examples:

```
gets(keybuff);
```

getwidth

Format:

```
getwidth ( );
```

Description:

The getwidth function returns the actual screen width of CoCo 3's standard text screen. This function works only with standard screen widths (32, 40, or 80 column).

Note: CoCo 3 ONLY.

Returns:

Screen width (32, 40, or 80).
ERR if not using standard text screen or not CoCo 3.

Examples:

```
w=getwidth();
printf("This screen is set to %d columns\n", w);
```

iniacia

Format:

iniacia (baud) int baud;

Description:

The iniacia function initializes the ACIA-PAK (RS232) and sets up the baud rate to the value of 'baud'. The ACIA-PAK is initialized in normal mode, with 8 data bits, 1 stop bit, and no parity. The accepted values for baud are: 110, 300, 600, 1200, 2400, 4800, 9600, and 19200. See also uain and uaout.

Returns:

ERR if baud rates other than above are given.

Examples:

```
iniacia(9600);              /* set ACIA-PAK for 9600 baud */
```

iniser

Format:

iniser (baud) int baud;

Description:

The iniser function initializes the CoCo's internal serial port (printer) and sets up the baud rate to the value of 'baud'. The accepted values for baud are: 110, 300, 600, 1200, 2400, 4800, and 9600. See also serout.

Note:

9600 baud may be marginal on some CoCo's.

Returns:

ERR if baud rates other than above are given.

Examples:

```
iniser(1200);               /* set printer port for 1200 baud */
```

is2coco

Format:

is2coco ();

Description:

The is2coco function returns TRUE if the computer hardware is a CoCo 1 or 2.

Returns:

TRUE if computer is CoCo 1 or 2.
FALSE if computer is not CoCo 1 or 2.

Examples:

```
if(is2coco())  {
   printf("I'm sorry this program is for CoCo 3 only\n");
   exit();
   }
```

is3coco

Format:

is3coco ();

Description:

The is3coco function returns TRUE if the computer hardware is a CoCo 3.

Returns:

TRUE if computer is CoCo 3.
FALSE if computer is not CoCo 3.

Examples:

```
if(is3coco())
   printf("Welcome to the CoCo 3 XYZ program\n");
```

isalnum

Format:

isalnum (c) char c;

Description:

The isalnum function returns TRUE if the character 'c' is alphanumeric ('A'-'Z', 'a'-'z', or '0'-'9').

Returns:

TRUE if 'c' is alphanumeric.
FALSE if 'c' is not alphanumeric.

Examples:

```
while(isalnum(*str))              /* print title in string */
   putchar(*str++);
```

isalpha

Format:

isalpha (c) char c;

Description:

The isalpha function returns TRUE if the character 'c' is alphabetic ('A'-'Z' or 'a'-'z').

Returns:

TRUE if 'c' is alphabetic.
FALSE if 'c' is not alphabetic.

Examples:

```
test  =  isalpha(getchar());
```

isascii

Format:

isascii (c) char c;

Description:

The isascii function returns TRUE if the character 'c' is an ASCII character (decimal values 0-127).

Returns:

TRUE if 'c' is an ASCII character.
FALSE if 'c' is not an ASCII character.

Examples:

```
if(isascii(inchar))          /* print only if ASCII */
   putchar(inchar);
```

iscntrl

Format:

iscntrl (c) char c;

Description:

The iscntrl function returns TRUE if the character 'c' is a control character (ASCII codes 0-31 or 127).

Returns:

TRUE if 'c' is a control character.
FALSE if 'c' is not a control character.

Examples:

```
if(iscntrl(inchar))          /* print '.' if cntrl char */
   putchar('.');
```

isdigit

Format:

isdigit (c) char c;

Description:

The isdigit function returns TRUE if the character 'c' is an ASCII digit ('0'-'9').

Returns:

TRUE if 'c' is a digit ('0'-'9').
FALSE if 'c' is not a digit.

Examples:

```
if(isdigit(inchar))        /* convert digit to binary */
   binnum = inchar - 0x30;
```

islower

Format:

islower (c) char c;

Description:

The islower function returns TRUE if the character 'c' is a lower-case letter (ASCII codes 97-122).

Returns:

TRUE if 'c' is lower-case.
FALSE if 'c' is not lower-case.

Examples:

```
caseflg = islower(inchar);
```

isprint

Format:

isprint (c) char c;

Description:

The isprint function returns TRUE if the character 'c' is a printable character (ASCII codes 32-126). Spaces are considered printable.

Returns:

TRUE if 'c' is printable.
FALSE if 'c' is not printable.

Examples:

```
if(isprint(inchar))          /* print char if printable */
   putchar(inchar);
else
   putchar('.');             /* print '.' if not printable */
```

ispunct

Format:

ispunct (c) char c;

Description:

The ispunct function returns TRUE if the character 'c' is a punctuation character (all ASCII codes except control characters and alphanumeric characters).

Returns:

TRUE if 'c' is a punctuation character.
FALSE if 'c' is not a punctuation character.

Examples:

```
while(ispunct(*ptr))
   ptr++;
```

isspace

Format:

isspace (c) char c;

Description:

The isspace function returns TRUE if the character 'c' is a white-space character (ASCII SP, HT, VT, CR, LF, or FF).

TRUE if 'c' is white-space character.
FALSE if 'c' is not white-space character.

Examples:

```
while(isspace(*ptr))
   ptr++;
```

isupper

Format:

isupper (c) char c;

Description:

The isupper function returns TRUE if the character 'c' is a an upper-case letter (ASCII codes 65-90).

Returns:

TRUE if 'c' is upper-case.
FALSE if 'c' is not upper case.

Examples:

```
caseflg  =  isupper(inchar);
```

isxdigit

Format:

isxdigit (c) char c;

Description:

The isxdigit function returns TRUE if the character 'c' is a hexadecimal digit ('0'-'9', 'A'-'F', or 'a'-'f').

TRUE if 'c' is a hexadecimal digit.
FALSE if 'c' is not a hexadecimal digit.

Examples:

```
hexflg  =  isxdigit(c);
```

itoa

Format:

itoa (nbr, str) int nbr; char *str;

Description:

The itoa function converts the number 'nbr' to its decimal character string representation at 'str'. The result is left-justified at 'str' with a leading minus sign if 'nbr' is negative. A NULL character terminates the string, which must be large enough to hold the result.

Returns:

Nothing

Examples:

```
itoa(1234,  numbuf);
```

itoab

Format:

itoab (nbr, str, base) int nbr; char *str; int base

Description:

The itoab function converts the unsigned integer 'nbr' to its character string representation at 'str' in base 'base'. The result is left-justified at 'str'. A NULL character terminates the string, which must be large enough to hold the result.

Returns:

Nothing

Examples:

```
itoab(0x17ff,  numbuf,  16);
```

itod

Format:

itod (nbr, str, sz) int nbr; char *str; int sz;

Description:

The itod function converts the number 'nbr' to a signed decimal character string at 'str' with a string length of 'sz'. The result is right-justified and padded with blanks in 'str'. If 'sz' is greater than zero, a NULL byte is placed at str[sz - 1]. If 'sz' is zero, conversion terminates upon the first null byte found in 'str'. If 'sz' is less than zero, all 'sz' characters of 'str' are used. Itod returns 'str'.

Returns:

Pointer to 'str'.

Examples:

```
itod(123,  numbuf,  7);      /* print "   123" */
puts(numbuf);
```

itou

Format:

itou (nbr, str, sz) int nbr; char *str; int sz;

Description:

The itou function converts 'nbr' to an unsigned decimal character string at 'str' with a string length of 'sz'. The result is right-justified and padded with blanks in 'str'. If 'sz' is greater than zero, a NULL byte is placed at str[sz - 1]. If 'sz' is zero, conversion terminates upon the first null byte found in 'str'. If 'sz' is less than zero, all 'sz' characters of 'str' are used. Itou returns 'str'.

Returns:

Pointer to 'str'.

Examples:

```
itou(45678,  numbuf,  7);  /* print " 45678" */
puts(numbuf);
```

itox

Format:

itox (nbr, str, sz) int nbr; char *str; int sz;

Description:

The itox function converts 'nbr' to a hexadecimal character string at 'str' with a string length of 'sz'. The result is right-justified and padded with blanks in 'str'. If 'sz' is greater than zero, a NULL byte is placed at str[sz - 1]. If 'sz' is zero, conversion terminates upon the first null byte found in 'str'. If 'sz' is less than zero, all 'sz' characters of 'str' are used. Itox returns 'str'.

Returns:

Pointer to 'str'.

Examples:

```
itox(1234,  numbuf,  5);   /* print " 4D2" */
puts(numbuf);
```

kill

Format:

kill (fname) char *fname;

Description:

The kill function deletes the file associated with the file pointer 'fname'. An optional drive number separated by a colon may be used. This function works like BASIC's KILL command.

Returns:

ERR if unsuccessful delete.

Examples:

```
if(kill("DATA.TXT:2")  ==  ERR)
   printf("Cannot  delete  file\n");
```

left

Format:

left (str) char *str;

Description:

The left function left-adjusts the character string at 'str'. Starting with the first nonblank character and proceeding through the null terminator, the string is moved to the address indicated by 'str'.

Returns:

Nothing

Examples:

```
left(title);
```

loadm

Format:

loadm (fname, offset) char *fname; int offset;

Description:

The loadm function loads a machine language file associated with the file pointer 'fname' into memory. The 'offset' value may be any number from 37267 to -32768. For files that do not require an offset; an offset of 0 *must* be used. Caution should be exercised using this function because loadm can overwrite your program and/or associated library functions.

Returns:

ERR if unsuccessful load.

Examples:

```
if(loadm("DATA.BIN",  0)  ==  ERR)
   printf("Cannot  load  file\n");
```

locate

Format:

locate (x, y) int x, y;

Description:

The locate function positions the CoCo 3's text screen cursor by the x-y coordinate values of 'x' and 'y'. The cursor column is associated with value 'x' and the cursor row is associated with value 'y'. The CoCo 3 must be in either 40 or 80 column text mode. This function works the same as BASIC's LOCATE command.

Note: CoCo 3 ONLY.

Returns:

ERR if not CoCo 3, or if not in 40 or 80 column mode, or if out of range.

Examples:

```
locate(2,8);          /* set cursor on 2nd line, 8th char */
```

pad

Format:

pad (str, ch, nbr) char *str; int ch, nbr;

Description:

The pad function places 'nbr' occurrences of the character 'ch' at the string pointer 'str'. The result is left-justified and the size of the 'str' buffer must be at least as large as 'nbr'.

Returns:

Nothing

Examples:

```
pad(tmpbuf,  0xff,  1024); /* fill temp buffer w/FF's */
```

peek

Format:

peek (addr) int addr;

Description:

The peek function reads a single byte from memory addressed at unsigned 'addr'. The returned value is an integer in the range from 0-255.

Returns:

Byte at address 'addr'.

Examples:

```
value  =  peek(0x2000);
```

peekw

Format:

peekw (addr) int addr;

Description:

The peekw function reads a word from memory addressed at unsigned 'addr'. The word is read in the order high-byte:low-byte. The returned value is an integer in the range from 0-65535.

Returns:

Word at address 'addr'.

Examples:

```
value = peekw(0x2000);
```

poke

Format:

poke (addr, byte) int addr, byte;

Description:

The poke function writes the byte 'byte' to memory addressed at unsigned 'addr'. The integer 'byte' must be in the range from 0-255.

Returns:

Nothing

Examples:

```
poke(0x8000,  0);          /* write 0 to location $8000 */
```

pokew

Format:

pokew (addr, word) int addr, word;

Description:

The pokew function writes the word 'word' to memory addressed at unsigned 'addr'. The word is written in the order high-byte:low-byte. The integer 'word' may be in the range from 0-65535.

Returns:

Nothing

Examples:

```
pokew(0x8000,  0x1234);   /* write $1234 to location $8000 */
```

printf

Format:

```
#include "STDIO.H"
printf (str, arg1, arg2,...) char *str;
```

Description:

The printf function performs a formatted print to the CoCo's standard output. The string 'str' is a null-terminated control string. Printf writes 'str' to the standard output, substituting the arguments in 'str' to special "conversion specifiers". These specifiers are represented by a preceding '%', and may be one of the following:

b	- binary integer
c	- single character
d	- signed decimal integer
o	- octal integer
s	- character string
u	- unsigned decimal integer
x	- hexadecimal number

printf

A numeric "field width" specifier may be inserted between the "%' and the conversion specifier (ie. %4x). This allows the value to be printed with the specified field width. If the field width contains a leading '0', the output will be padded with zeros, otherwise it is padded with spaces. If the field width is a negative number, the output will be left justified in the field, otherwise the output is right justified. If no field width is specified, the output will use only as much space as required.

Returns:

Count of total characters written.

Examples:

```
printf("Beans\n");
printf("There has been %u counts\n", count);
```

putc

Format:

```
#include "STDIO.H"
putc (c, fd)  char c; int fd;
```

Description:

The putc function writes the character 'c' to the file associated with the file descriptor 'fd'. The putc function is similar to the fputc function. The only difference is that the putc function is a macro rather than a function.

Returns:

c if successful.
EOF if an error occurs.

Examples:

```
if(putc(c, fd) == EOF)
   printf("Error writing to file\n");
```

putch

Format:

```
#include "STDIO.H"
putch (c)  char c;
```

Description:

The putch function writes the character 'c' to the standard output. If a newline character is encountered, it is translated to a carriage return, followed by a line-feed. This would advance the printing position to the first column of the next line.

Returns:

Nothing

Examples:

```
putch('z');
putch('\n');
```

putchar

Format:

```
#include "STDIO.H"
putchar (c)  char c;
```

Description:

The putchar function is similar to the putch function. The only difference is that if the character 'c' is greater than 127, the MSB will be set to zero to maintain ASCII compatibility before the character is sent to the standard output.

Returns:

Nothing

Examples:

```
putchar('x');
putchar(0xa0);          /* this would output an ASCII space ($20)*/
```

puts

Format:

```
#include "STDIO.H"
puts (str)  char *str;
```

Description:

The puts function writes a null-terminated string of characters to the standard output indicated by string 'str'. A newline character is appended to the string and the null-terminating byte is not written.

Returns:

Nothing

Examples:

```
puts("I'm  up!\n");
puts(keybuf);
```

rename

Format:

```
rename (source, dest)  char *source, *dest;
```

Description:

The rename function renames the file associated with the file pointer 'source' to the filename associated with the file pointer 'dest'. An optional drive number separated by a colon may be used. This function is similar to BASIC's RENAME command except that the keyword 'TO' is not used.

Returns:

ERR if unsuccessful rename.

Examples:

```
if(rename("DATA.TXT",  "OLD.TXT")  ==  ERR)
   printf("Cannot  rename  file\n");
```

restore

Format:

restore ( );

Description:

The restore functions *undoes* what the settrap function does. This function restores the CoCo's normal error trapping routines back to its default BASIC vectors. It is normally called from "CSTART.C" (providing you are using error trapping) when your program returns to BASIC.

Returns:

Nothing

Note:

See also settrap.

Examples:

See "CSTART.C"

reverse

Format:

reverse (str) char *str;

Description:

The reverse function reverses the order of the characters in the null-terminated string at 'str'.

Returns:

Nothing

Examples:

```
reverse(numbuf);
```

rgb

Format:

rgb ( );

Description:

The rgb function puts the CoCo 3 in RGB video mode. This function operates the same as BASIC's RGB command.

Note: CoCo 3 ONLY
See also cmp.

Returns:

Nothing

Examples:

```
rgb();                          /* put CoCo 3 in RGB mode */
```

rscinit

Format:

rscinit (globram, size) char *globram; int size;

Description:

The rscinit function sets up and initializes a user defined buffer required for CoCo-C's I/O library. The pointer 'globram' points to the user defined buffer which should be at least 128 bytes. The 'size' value contains the exact size of the user buffer. This function *must* be called if any of CoCo-C's DOS functions or non-standard library functions (unique to CoCo) are to be used within a given program. By default, rscinit is called within the "CSTART.C" file before your program enters main(). Also "CSTART.C" contains rscinit's 128 byte buffer.

Returns:

Nothing

Examples:

See "CSTART.C"

savem

Format:

savem (fname, start, end, exec) char *fname; int start, end, exec;

Description:

The savem function saves a machine language program or binary data from memory to the file associated with the file pointer 'fname'. The 'start' value contains the program's starting address, the 'end' value contains the program's ending address, and the 'exec' value contains the program's execution address. If no execution address is required, the starting address may be used. This function is similar to BASIC's SAVEM command.

Returns:

ERR if unsuccessful save.

Examples:

```
if(savem("DATA.BIN",  0x4000,  0x407f,  0x4000)  ==  ERR)
   printf("Cannot  save  file\n");
```

scanf

Format:

#include "STDIO.H"
scanf (str, arg1, arg2,...) char *str;

Description:

The scanf function reads a series of fields from CoCo's standard input. The string 'str' is a null-terminated control string. Scanf reads from the standard input, substituting the arguments in 'str' to special "conversion specifiers". The result is stored at the locations indicated by the arguments. The conversion specifiers are represented by a preceding '%', and may be one of the following:

b	- binary integer
c	- single character
d	- signed decimal integer
o	- octal integer
s	- character string
u	- unsigned decimal integer
x	- hexadecimal number

scanf

A numeric "field width" specifier may be inserted between the "%' and the conversion specifier (ie. %4s). This allows the string to be stored with the maximum specified field width. If no field width is specified, the string will use as much space as required.

The conversion ends when the next white-space character (blank, tab, or newline) is encountered, or when the conversion specifier indicating a maximum field width is reached.

Returns:

Number of fields read.

Examples:

```
scanf("%c",   &c);          /* read non-white-space character */
scanf("%80s",  s);          /* read string in 's' max 80 chrs */
scanf("%d  %d",  &x,  &y);  /* read 2 decimal values in x & y */
```

serout

Format:

serout (c) char c;

Description:

The serout function sends the character 'c' to CoCo's internal serial port (printer). If the printer is not ready after a time-out period, no character is sent and an error status will be returned.

Note:

The iniser function should be called before using this function.

Returns:

'c' if character successfully sent.
ERR if the printer is not ready.

Examples:

```
if(serout(NULL)  ==  ERR)
   printf("Printer  not  ready\n");
```

settrap

Format:

settrap (address) int address;

Description:

The settrap function replaces BASIC's default error trapping routines with a user defined function located at 'address'. The CoCo-C's library error handling routines are replaced as the primary level and the user defined function is replaced as the secondary level. If a user defined function is not required, a null function may be used (ie. errfunc ( ) { }).

Returns:

Nothing

Note:

If the #define ERRTRAP is declared within your C program, the error handling setup will be done automatically by the compiler and "CSTART.C".

See also restore.

Examples:

See "CSTART.C"

sign

Format:

sign (nbr) int nbr;

Description:

The sign function returns minus one, zero, or plus one depending on whether 'nbr' is less than, equal to, or greater than zero.

Returns:

-1 if 'nbr' is less than 0
 0 if 'nbr' equals 0
+1 if 'nbr' is greater than 0

sign

Examples:

```
if(sign(temp)  ==  -1)
   coldflg  =  TRUE;
```

slow

Format:

slow ( );

Description:

The slow function puts the CoCo 3 in normal speed mode. This function has the same effect as POKE &HFFD8, 0 in BASIC.

Note:

CoCo 3 ONLY; See also fast.

Returns:

Nothing

Examples:

```
slow();                              /* back to normal speed */
```

sprintf

Format:

#include "STDIO.H"
sprintf (buff, str, arg1, arg2,...) char *buff; char *str;

Description:

The sprintf function performs a formatted print to the memory location pointed to by buff. The format of the output is controlled by the 'str' string. Following the 'str' string is an optional list of values to be written. The sprintf function is almost identical to the printf function, except that printf writes to stdout and sprintf writes to memory. After the last value has been written, the sprintf function appends a NULL terminating character.

sprintf

Returns:

Count of total characters written.

Note:

Also see printf for a description of the format string.

Examples:

```
char  buffer[80];           /* sprintf storage area */
sprintf(buffer,  "Name:%s,  Age:%d\n",  "John  Doe",  47);
fputs(buffer,  stdout);     /* send output to screen */
fputs(buffer,  printer);    /*   and to printer */
```

sscanf

Format:

```
#include "STDIO.H"
sscanf (buff, str, arg1, arg2,...)  char *buff; char *str;
```

Description:

The sscanf function reads formatted data from memory pointed to by buff. The reading of the input data is controlled by the 'str' string. Following the 'str' string is an optional list of variable addresses where the input data is stored. The sscanf function is almost identical to the scanf function, except that scanf reads from stdin and sscanf reads from memory.

Returns:

Number of fields read.

Note:

Also see scanf for a description of the format string.

Examples:

```
char  c;
int  i;
char  str[20];
char  buffer[]  =  "a  12345  GoodBye";
sscanf(buffer,  "%c  %d  %s",  &c,  &i,  &str);
printf("%c  %d  %s",  c,  i,  str);
```

strcat

Format:

strcat (dest, source) char *dest, *source;

Description:

The strcat function appends the string at 'source' to the end of the string at 'dest', returning 'dest'. The null character at the end of 'dest' is replaced by the leading character of 'source'. A NULL character terminates the new 'dest' string. The space reserved for 'dest' must be large enough to hold the result.

Returns:

Pointer to zero terminated 'dest' string.

Examples:

```
strcat(filename,  ".c");
```

strchr

Format:

strchr (str, c) char *str, c;

Description:

The strchr function returns a pointer to the first occurrence of the character 'c' in the string at 'str'. It returns NULL if the character is not found. Searching ends with the first null character.

Returns:

Pointer to character 'c' in string 'str'.
NULL if character 'c' not found.

Examples:

```
dot  =  strchr(filename,  '.');
```

strcmp

Format:

strcmp (str1, str2) char *str1, *str2;

Description:

The strcmp function returns an integer less than, equal to, or greater than zero, depending upon the comparisons of two strings. Character-by-character comparisons are made starting at the left end of the strings until a difference is found. Comparison is based on the ASCII numeric values of the characters.

Returns:

 0 if 'str1' equals 'str2'
+1 if 'str1' is greater than 'str2'
-1 if 'str2' is greater than 'str1'

Examples:

```
if(!strcmp(inbuff,   "help"))
   hlpstat  =  TRUE;
```

strcpy

Format:

strcpy (dest, source) char *dest, *source;

Description:

The strcpy function copies the string at 'source' to 'dest', returning 'dest'. All data is copied including the null-terminating byte. The space at 'dest' must be large enough to hold the string at 'source'.

Returns:

Pointer to zero terminated 'dest' string.

Examples:

```
strcpy(filename,   "DEMO");
```

strlen

Format:

strlen (str) char *str;

Description:

The strlen function returns a count of the number of characters in the string at 'str'. It does not count the null character that terminates the string.

Returns:

Length of 'str'.

Examples:

```
length = strlen(inbuff);
```

stncat

Format:

stncat (dest, source, n) char *dest, *source; int n;

Description:

The stncat function appends 'n' number of characters from the string at 'source' to the end of the string at 'dest', returning 'dest'. The null character at the end of 'dest' is replaced by the leading character of 'source'. A NULL character terminates the new 'dest' string. The space reserved for 'dest' must be large enough to hold the result.

Returns:

Pointer to zero terminated 'dest' string.

Examples:

```
stncat(filename, extension, 3);
```

stncmp

Format:

stncmp (str1, str2, n) char *str1, *str2; int n;

Description:

The stncmp function returns an integer less than, equal to, or greater than zero, depending upon the comparisons of two strings. Character-by-character comparisons are made starting at the left end of the strings until either a difference is found, or the count of 'n' characters is reached. Comparison is based on the ASCII numeric values of the characters.

Returns:

0 if 'str1' equals 'str2'
+1 if 'str1' is greater than 'str2'
-1 if 'str2' is greater than 'str1'

Examples:

```
if(!stncmp(fnbuff,  filename,  8))
    puts("Please  select  another  name\n");
```

stncpy

Format:

stncpy (dest, source, n) char *dest, *source; int n;

Description:

The stncpy function copies 'n' number of characters from the string at 'source' to 'dest', returning 'dest'. If the source string is too short, null padding occurs. If it is too long, it is truncated in 'dest'. A NULL character is appended to the end of the destination string.

Returns:

Pointer to zero terminated 'dest' string.

Examples:

```
stncpy(outbuff,  inbuff,  16);
```

toascii

Format:

toascii (c) char c;

Description:

The toascii function returns the character 'c' as an ASCII character equivalent in the ASCII character set.

Returns:

ASCII equivalent of 'c'.

Examples:

```
nxtchar = toascii(inchar);
```

tolower

Format:

tolower (c) char c;

Description:

The tolower function returns the lower-case equivalent of 'c' if 'c' is an upper-case letter; otherwise, it returns 'c' unchanged.

Returns:

Lower-case equivalent of 'c'.

Examples:

```
putchar(tolower(outchar));
```

toupper

Format:

toupper (c) char c;

Description:

The toupper function returns the upper-case equivalent of 'c' if 'c' is a lower-case letter; otherwise, it returns 'c' unchanged.

Returns:

Upper-case equivalent of 'c'.

Examples:

```
inchar  =  toupper(getchar());
```

uain

Format:

uain ();

Description:

The uain function gets a single character from the external ACIA-PAK (RS232). This function does *not* wait for the character. If the character is ready, a positive integer in the range from 0-255 is returned, otherwise an ERR is returned. The character returned is as exactly as it is read (no filtering).

Note:

The iniacia function must be called before using this function.

Returns:

Integer (0 - 255)
ERR if character is not ready.

Examples:

```
while((c=uain()) != ERR);   /* wait for char from ACIA-PAK */
```

toascii

Format:

toascii (c) char c;

Description:

The toascii function returns the character 'c' as an ASCII character equivalent in the ASCII character set.

Returns:

ASCII equivalent of 'c'.

Examples:

```
nxtchar  =  toascii(inchar);
```

tolower

Format:

tolower (c) char c;

Description:

The tolower function returns the lower-case equivalent of 'c' if 'c' is an upper-case letter; otherwise, it returns 'c' unchanged.

Returns:

Lower-case equivalent of 'c'.

Examples:

```
putchar(tolower(outchar));
```

toupper

Format:

toupper (c) char c;

Description:

The toupper function returns the upper-case equivalent of 'c' if 'c' is a lower-case letter; otherwise, it returns 'c' unchanged.

Returns:

Upper-case equivalent of 'c'.

Examples:

```
inchar  =  toupper(getchar());
```

uain

Format:

uain ();

Description:

The uain function gets a single character from the external ACIA-PAK (RS232). This function does *not* wait for the character. If the character is ready, a positive integer in the range from 0-255 is returned, otherwise an ERR is returned. The character returned is as exactly as it is read (no filtering).

Note:

The iniacia function must be called before using this function.

Returns:

Integer (0 - 255)
ERR if character is not ready.

Examples:

```
while((c=uain()) != ERR);   /* wait for char from ACIA-PAK */
```

uaout

Format:

uaout (c) char c;

Description:

The uaout function sends the character 'c' to the external ACIA-PAK (RS232). No filtering or translations of char 'c' are performed during this function call.

Note:

The iniacia function must be called before using this function.

Returns:

Nothing

Examples:

```
while(*buf)  uaout(*buf++);   /* send all chars in buf to ACIA */
```

utoi

Format:

utoi (str, nbr) char *str; int *nbr;

Description:

The utoi function converts the unsigned decimal number represented by the character string at 'str' to an integer at 'nbr' and returns the length of the numeric field found. Conversion stops at the end of the string or upon any non-decimal character. With 16-bit integers, utoi will use at most five digits.

Returns:

Field length of 'nbr'.
ERR on error.

Examples:

```
length  =  utoi("56789",&n);
```

xtoi

Format:

xtoi (str, nbr) char *str; int *nbr;

Description:

The xtoi function converts the hexadecimal number in the character string at 'str' to an integer at 'nbr' and returns the length of the hexadecimal field found. Conversion stops when it encounters a non-hexadecimal digit in 'str'. With 16-bit integers, xtoi will use at most four digits.

Returns:

Field length of 'nbr'.
ERR on error.

Examples:

```
length  =  xtoi("7FFF",&n);
```

1. Introduction:

CoCo-ASM is a two-pass symbolic assembler for the CoCo 1, 2 or 3. Although originally designed for CoCo-C's assembly language output, it can also be used as a stand-alone assembler for creating machine language files.

CoCo-ASM conforms to a "Motorolla-Style" syntax for the 6809 microprocessor. The assembler reads as input an assembly language source file (ASCII text) and produces as output a machine language binary file ready for LOADM and EXEC. The assembler also provides options for a formatted list file output and/or a symbol table listing.

2. Assembler Specifications:

2.1 Symbols:

Symbols may consist of an ASCII character or a string of ASCII characters. There are no restrictions of which characters may be used, but to avoid confusion within the assembler, 'operator' characters should be avoided.

Symbol names are limited to seven characters. More than seven characters may be used, but the remaining characters will be ignored.

2.2 Source File Format:

CoCo-ASM expects source input lines to be in the following format:

<label> <instruction> <operands> <comment>

Labels (symbols) must always be in column one. Each field must be separated from its adjacent field. A minimum of one space or one tab must be used as a field delimiter.

If the instruction requires an operand(s), then the operand is required and must be placed in the operand filed.

The comment field is optional and is ignored by the assembler. Any spaces or tabs used in the comment field may be used only as delimiters in a text string, if not it would be recognized as the end of the operand field.

An asterisk '*' in the label field (column one) may be used as a comment. Any text following the asterisk is ignored by the assembler.

2.3 Expressions:

All expressions are evaluated from left to right, using 16 bit values. If an 8 bit value is evaluated within an expression, the lower 8 bits will be used. Spaces or tabs are not allowed within an expression, unless they are contained within a text string.

CoCo-ASM supports the following expression operators:

2.3.1 unary operators:

		examples:
-	negation	`-VALUE`
~	one's complement	`~VALUE`
=	swaps high and low byte of value	`=VALUE`

2.3.2 binary operators:

+	addition	`VALUE1+VALUE2`
-	subtraction	`VALUE1-VALUE2`
*	multiplication	`VALUE1*VALUE2`
/	division	`VALUE1/VALUE2`
\	modulo, ie. remainder from division	`VALUE1\VALUE2`
|	logical inclusive OR	`VALUE1|VALUE2`
^	logical exclusive OR	`VALUE1^VALUE2`
&	logical AND	`VALUE1&VALUE2`

2.3.3 values in expressions:

nnn	decimal number	`123`
$hhh	hexidecimal number	`$1A`
%bbb	binary number	`%11001010`
@ooo	octal number	`@177`
'cc'	ASCII characters	`'A'`
<label>	label value from symbol table	`VALUE`
*	value of current program counter	`EQU *`

3. Addressing Modes:

The CoCo-ASM assembler fully supports the addressing modes which are available to the 6809 microprocessor. These modes are determined by the use of the instructions and/or its operands.

3.1 Immediate Addressing:

Any operand which is preceded by a pound sign ('#'), is determined to be an immediate value. For instructions requiring only 8 bits of immediate data, the lower eight bits of the value will be used. The upper 8 bits of a value can be accessed by preceding it with a '=' (this swaps the high and low bytes).

3.2 Direct/Extended Addressing:

Any operand which is preceded by a left angle bracket ('<'), is determined to be a DIRECT (8 bit) address. This will reference a value in the memory page indicated by the Direct Page (DP) register.

If an operand is preceded by a right angle bracket ('>'), it is determined to be an ABSOLUTE or EXTENDED (16 bit) address.

If no addressing mode is specified, the assembler will use DIRECT addressing if the high byte of the values match the last SETDP, otherwise the ABSOLUTE addressing mode will be used.

3.3 Indirect Addressing:

Indirect addressing is indicated by placing the addressing value in square braces ('[ ]') ie. LDA [$F000].

4. Pseudo-ops:

The CoCo-ASM assembler supports many of the standard 6809 pseudo-ops as well as others which are unique to the assembler. The following is a list of all the pseudo-ops supported by CoCo-ASM:

NAME <text string>

This pseudo-op stores the name of the program and displays it at the top of each of each page in the listing. By default, 'NAME' is set to the file name being assembled. Lines containing this pseudo-op will not appear in the listing.

PAGE

This pseudo-op forces a page eject in the listing. Lines containing this pseudo-op will not appear in the listing.

NOLIST

This pseudo-op suppresses the source listing in the program, preventing any further lines from being displayed.

LIST

This pseudo-op re-enables the output listing, following a NOLIST pseudo-op.

END

The END pseudo-op was included for compatibility with other assemblers. CoCo-ASM simply ignores the END pseudo-op. (Normally, this would indicate the end of assembly and any lines following END would be ignored.)

<label> EQU <expression>

The label of this pseudo-op is assigned to the value of the operand expression.

ORG <expression>

This pseudo-op sets the assembler's internal program counter to the value of the operand expression. All code following ORG will be generated starting at that address. If code is generated without an ORG expression, it will start at address $0000.

REORG

This pseudo-op resets the assembler's internal program counter to the value it had before just before the last ORG expression. It is used to continue assembly which preceded the last ORG.

FCB <expr1>[,<expr2>,<expr3>,...]

Form Constant Byte. This pseudo-op places the values of the operand expressions into memory as single byte constants.

FDB <expr1>[,<expr2>,<expr3>,...]

Form Double Byte. This pseudo-op places the values of the operand expressions into memory as double byte constants.

RMB <expression>

Reserve Memory Bytes. This pseudo-op reserves a number of bytes equal to the value of the operand expression. No code is generated, and the area is not altered when the resulting binary file is loaded.

FCC 'text string'

Form Constant Character. This pseudo-op places the string into memory as ASCII byte values. Any character which is not part of the text string may be used as delimiters. ie. "This is text" or /This is text/.

FCCZ 'text string'

This pseudo-op is almost identical to the FCC pseudo-op, with the exception that FCCZ has a zero byte ($00) appended to it.

SETDP <expression>

The SETDP pseudo-op sets the assembler's default direct page register to the 8 bit value of <expression>. This is used to inform the assembler what value is in the DP register, so that direct addressing may be used. If the value of <expression> is is greater than 255, or less than 0, the default direct page register will be disabled, and all unspecified memory addresses will use extended addressing. If the SETDP pseudo-op is not used, direct page addressing will be disabled.

5. Using the Assembler:

The assembler is first invoked by running CoCo-C's command coordinator. Type **RUN "CC <ENTER>**. After the menu appears, select **A** (for assemble). The assembler will automatically load and execute and request you for the input file:

```
Input Filename ? _
```

Enter the name of your assembly language source file. Examples:

TEST.ASM	*read input file "TEST.ASM" from default drive*
MYFILE.ASM:2	*read input file "MYFILE.ASM" from drive 2*

Next the assembler will ask you for the output file:

```
Output Filename ? _
```

Enter any valid name as your binary output file. Examples:

TEST.BIN	*create output file "TEST.BIN" on default drive*
MYFILE.OBJ:2	*create output file "MYFILE.OBJ" on drive 2*

If no drive designator is specified, the output file will be created on the default drive.

Next the assembler will ask you for the assembly options:

```
Options ?_
```

One or more assembler options may be selected at this time. Each option contains one character. Spaces or commas may be used as delimiters. After selecting the desired option(s) hit **<ENTER>**. If no options are needed, just hit **<ENTER>**.

Assembler Options:

```
L = List Output          defaults to screen
F = To File              all output is directed to the .LST output file
S = Symbol Table         includes symbol table in .LST output file
P = Pause on Error
A = Alarm on Error
```

Entering a **? <ENTER>** will display the options if you forget what they are.

After the <ENTER> key is pressed, the assembler will begin compiling your code. Any errors will be displayed in the output listing. The assembler will pause on error if the 'P' option was selected. Hitting <ENTER> resumes the assembly, while the <BREAK> key terminates the assembly.

When the assembly has completed, and assuming there were no errors in your source file, the assembler will respond with:

```
No Errors Found
Hit any Key to Return to Menu
```

At this time, hitting any key will return you to CoCo-C's command coordinator.

If you assembled a file that does not require the CoCo-C library, you may LOADM and EXEC your file, otherwise you need to run CoCo-C's Library Linker. This is explained in the CoCo-C Linker section of this manual.

If you selected the 'L' and 'F' options, your list output file will appear on the same drive, with the same name as your binary file with a .LST extension.

Note: Hitting the <BREAK> key at any time while the assembler is running will terminate the assembly and return you to the command coordinator.

6. Error Messages:

When CoCo-ASM detects an error in the source code, it displays a descriptive error message indicating the possible cause of error. The error message(s) will be contained in the source listing on the line immediately following the line containing the error.

If any forward references are found in any of the EQU, ORG, or RMB pseudo-ops, an 'Undefined symbol' error will be displayed in the first pass of the assembly at the top of the listing.

CoCo-ASM produces the following error messages:

6.1 Duplicate symbol: <symbol name>

The displayed symbol has been defined more than once within this assembly.

6.2 Symbol table overflow

The symbol table has become full due to too many symbols within the program.

6.3 Unknown opcode

The indicated line does not contain a valid opcode or assembler directive within the instruction field.

6.4 Out of range

The operand is not within the range of values (ie. +127 to -128) which can be used with the instruction.

6.5 Illegal addr mode

The instruction on the indicated line does not apply with the addressing mode specified in the operand field.

6.6 Illegal register use

The instruction on the indicated line contains an unrecognized register, or the register is out of context specified by the instruction.

6.7 Undefined symbol

A symbol referenced in the indicated line has not been defined within this assembly, and has no value.

6.8 Invalid expression

The indicated line has an expression which contains a non-valid operator character.

6.9 Illegal argument

Thee operand on the indicated line is not in the proper format.

6.10 Invalid delimiter

The indicated line contains a character string constant which does not have a proper closing delimiter.

1. Introduction:

The purpose of the Library Linker is to combine CoCo-C's 90+ function library along with your compiled and assembled program. The entire function library is contained in the file "CLIB.BIN" on your distribution disk. The file size of this fully relocatable library is only slightly larger than 8K bytes.

The file "CLIB.INC" is the ASCII text file that contains the entry point addresses of each function in the library. These entry points are contained in a jump table at the very beginning of "CLIB.BIN". By default, the CoCo-C Compiler automatically includes "CLIB.INC" at the very end of your compiled code. This in turn informs the CoCo-ASM assembler where each function address in the library is located.

When linking is performed, the CoCo-C function library is appended to the end of your binary file. The resulting file is a stand-alone machine language program ready for LOADM and EXEC.

2. Using the Library Linker:

The linker is first invoked by running CoCo-C's command coordinator. Type **RUN "CC <ENTER>**. After the menu appears, select **L** (for link). The linker will automatically load and execute and request you for the input file:

```
Enter ML 'C' File - _
```

Enter the name of the *binary* file in which your C program was compiled and assembled to. Examples:

TEST.BIN	*read input file "TEST.BIN" from default drive*
MYFILE.OBJ:2	*read input file "MYFILE.OBJ" from drive 2*

Note: Hitting <ENTER> at the prompt without a filename will terminate the linker.

Next the linker will ask you for the output file:

```
Enter ML Out File - _
```

Enter any valid name as your binary output file. Examples:

TEST.BIN	*create output file "TEST.BIN" on default drive*
MYFILE.BIN:2	*create output file "MYFILE.BIN" on drive 2*

Warning:

If your input file and output file have the same name, the linker will overwrite the input file, replacing it with the newly created linked file.

After the <ENTER> key is pressed, the linker will begin linking your code. You should then see the following messages:

```
Loading Files...
Saving New File...

"TEST.BIN" Created
```

This would indicate that the linker has successfully linked your code, the next message would be:

```
Hit any Key to Return to Menu
```

At this time, hitting any key will return you to CoCo-C's command coordinator.

From the command coordinator, select **R** (for re-boot). You may now LOADM and EXEC your program.

3. Creating Non-Library Programs:

With CoCo-C it is possible to create programs which do not rely upon the CoCo-C function library. This should only be necessary when creating programs that need not communicate via the standard I/O (ie. user created graphics). An alternate reason would be when compiling for non-CoCo ROM-based applications (ie. stand alone single board computers) or applications that have very strict memory constraints.

As an option, the CoCo-C Compiler provides a method of disabling the inclusion of "CLIB.INC" in your compiled output code. By defining the following:

```
#define NOCLIB
```

at the beginning of your C program (before the #includes), will instruct the compiler to *not* include "CLIB.INC" at the end of your compiled code. This, of course, will prohibit the use of *all* the functions contained within the CoCo-C Library.

If the need arises to use some of the standard C functions, you may extract them from the source file "STDLIB.C" on the distribution disk. Also, the source file "CHARIO.ASM" contains "character in" and "character out" routines, and may be included within your program.

By not having the CoCo-C Library appended to your program, there is no need to run the linker. You may simply LOADM and EXEC your program after it assembles.

1. Preliminary:

In the following examples, we will walk you through the steps necessary for creating two machine language programs using CoCo-C. Any differences between the CoCo 2 and CoCo 3 versions will be indicated. First the following assumptions must be made:

1. Your monitor must be capable of displaying 80 columns of text (CoCo 3).

2. You are using a single-drive system.

3. You are using a backup copy of CoCo-C.

2. Example 1 (HELLO.C):

In this example, we will create the infamous "hello world" program. This program will require the use of the editor and will not need the CoCo-C Library.

2.1 Insert the backup copy of CoCo-C into the disk drive.

2.2 Type **RUN "CC <ENTER>**

2.3 Enter **E** (for edit)

2.4 You should now be in Ultra Editor's main menu or in Line Editor's 'Filename:' prompt. At the 'Filename:' prompt, type the following:

Ultra Editor (CoCo 3):	*Line Editor (CoCo 2):*
HELLO.C <ENTER>	**<ENTER>**

2.5 Now you should be in the edit mode. At the cursor, type in the following text (do not include spaces that surround <ENTER>).

Ultra Editor:

```
<ENTER>
#define  NOCLIB  <ENTER>
#include  "CSTART.C"  <ENTER>
#include  "CHARIO.ASM"  <ENTER>  <ENTER>

main()  <ALT < >  <ENTER>
    putstr("HELLO  WORLD");  <ENTER>
   <ALT > >  <ENTER> <ENTER>

putstr(str)  char  *str;  <ALT < >  <ENTER>
    while(*str)  charout(*str++);  <ENTER>
   <ALT > >  <ENTER>
```

Line Editor:

```
I
#define  NOCLIB  <ENTER>
#include  "CSTART.C"  <ENTER>
#include  "CHARIO.ASM"  <ENTER>  <ENTER>

main()  <SHIFT  CLEAR>  <ENTER>
    putstr("HELLO  WORLD");  <ENTER>
    <CLEAR>  <ENTER> <ENTER>

putstr(str)  char  *str;  <SHIFT  CLEAR>  <ENTER>
    while(*str)  charout(*str++);  <ENTER>
    <CLEAR>  <ENTER>  <BREAK>
```

2.6 Examine the text. It should look like this:

```
#define NOCLIB
#include "CSTART.C"
#include "CHARIO.ASM"

main() {
  putstr("HELLO WORLD");
  }

putstr(str) char *str; {
  while(*str) charout(*str++);
  }
```

Ultra Editor:

2.7 Next type **<CTRL Q >** to exit edit mode.

2.8 In Ultra Editor's main menu enter **S** (for save buffer).

2.9 Next hit **<ENTER>** at the 'Filename' prompt.

2.10 Now hit **C** to return to CoCo-C's Command Coordinator.

Line Editor:

2.7 Next type **Q** to exit edit mode.

2.8 In Line Editor's main menu, enter **W** (write buffer as new file).

2.9 At the 'WRITE AS:' prompt, type **HELLO.C <ENTER>**

2.10 After the file is saved, you will automatically return to the Command Coordinator.

2.11 Next enter **C** (for compile).

2.12 After the compiler loads, you should see the CoCo-C copyright message, followed by an 'Options' prompt. Respond with the following:

```
Options ? <ENTER>

Output Filename ? HELLO.ASM <ENTER>
Input Filename ? HELLO.C <ENTER>
Input Filename ? <ENTER>
```

2.13 The compiler will begin compiling immediately after the input file has been entered. After entering the last <ENTER>, the following messages should be displayed:

```
xxx Lines Compiled
No Errors Found

Hit any Key to Return to Menu
```

2.14 At this time, hit any key. You should now be back in the Command Coordinator.

2.15 Next enter **A** (for assemble).

2.16 After the assembler loads, you should see the CoCo-ASM copyright message, followed by an 'Input Filename' prompt. Respond with the following:

```
Input Filename ? HELLO.ASM <ENTER>
Output Filename ? HELLO.BIN <ENTER>
Options ? <ENTER>
```

2.17 After entering the last <ENTER> the compiler will assemble the "HELLO.ASM" file. The following messages should be displayed:

```
* PASS 1 *
* PASS 2 *

No Errors Found
Hit any Key to Return to Menu
```

2.18 Again, hit any key to bring you back to the Command Coordinator.

2.19 Since this example requires no linking, you may return back to BASIC. Now enter **R** (for re-boot) to exit CoCo-C.

2.20 You may now test your program. Enter the following:

```
LOADM "HELLO
EXEC
```

The "HELLO WORLD" message should now be displayed.

Congratulations on creating your first CoCo-C program!

3. Example 2 (FILELST.C).

In this example, we will compile, assemble, and link the program "FILELST.C" which is included on the distribution disk. Since this program is already written, the editor will not be required. After the procedure is complete, you may use this useful program to examine ASCII text files on the screen, or send them to your printer.

3.1 Insert the backup copy of CoCo-C into the disk drive.

3.2 Type **RUN "CC <ENTER>**

3.3 Now enter **C** (for compile).

3.4 After the compiler loads, you should see the CoCo-C copyright message, followed by an 'Options' prompt. Respond with the following:

```
Options ? <ENTER>

Output Filename ? FILELST.ASM <ENTER>
Input Filename ? FILELST.C <ENTER>
Input Filename ? <ENTER>
```

3.5 The compiler will begin compiling immediately after the input file has been entered. After entering the last <ENTER>, the following messages should be displayed:

```
xxx Lines Compiled
No Errors Found

Hit any Key to Return to Menu
```

3.6 At this time, hit any key. You should now be back in the Command Coordinator.

3.7 Next enter **A** (for assemble).

3.8 After the assembler loads, you should see the CoCo-ASM copyright message, followed by an 'Input Filename' prompt. Respond with the following:

```
Input Filename ? FILELST.ASM <ENTER>
Output Filename ? FILELST.OBJ <ENTER>
Options ? <ENTER>
```

3.9 After entering the last <ENTER> the compiler will assemble the "FILELST.ASM" file. The following messages should be displayed:

```
* PASS 1 *
* PASS 2 *

No Errors Found
Hit any Key to Return to Menu
```

3.10 Again, hit any key to bring you back to the Command Coordinator.

3.11 Now enter **L** (for link).

3.12 After the linker loads, you should see the CoCo-C Library Linker copyright message, followed by an 'Enter ML 'C' file prompt'. Respond with the following:

```
Enter ML 'C' File - FILELST.OBJ <ENTER>
Enter ML Out File - FILELST.BIN <ENTER>
```

3.13 After entering the last <ENTER> the linker will link the "FILELST.OBJ" file. The following messages should be displayed:

```
Loading Files...
Saving New File...

"FILELST.BIN" Created

Hit any Key to Return to Menu
```

3.14 Once again, hit any key to bring you back to the Command Coordinator.

3.15 Now enter **R** (for re-boot) to exit CoCo-C.

3.16 You may now test your program. Enter the following:

```
LOADM  "FILELST
EXEC
```

4. Using the ASCII File Lister:

The included ASCII file lister is provided to view ASCII text files (C, assembler, etc.) to either the screen, or to the printer. It is self-prompting and will keep requesting input files until a non-Yes answer is entered.

4.1 Screen Output:

During the displaying of text, you may hit the spacebar to pause the display. Hitting the spacebar again resumes the display. The <BREAK> key terminates the display.

4.2 Printer Output:

The printer output defaults to 2400 baud. As an option, you may add a Line Feed after each Carriage Return (this eliminates setting DIP switches on your printer). If the printer is not ready while sending data, an error message will be displayed on the screen. Your options are:

1. Make the printer ready (select it, add paper, etc.)

2. Hit **<BREAK>** to exit the lister.

4.3 Making Modifications to the Lister:

Since the ASCII File Lister is supplied in C source, it may be modified to suit your needs. For example, you may want to change the screen width, change the foreground and background colors, or even change the printer baud rate. Margins, page numbering and title headers can improve the printer output if you feel ambitious. Making these or any modifications on an existing C program is the best way to get familiar with C and the CoCo-C Compiler.