BABY BASIC

A TUTORIAL

ON

HOW TO ACCESS THE EXTRA MEMORY IN YOUR COMPUTER TO STORE AND EXECUTE BASIC PROGRAM LINES.

ALSO, HOW TO CHAIN PROGRAM SECTIONS FROM DISK WITHOUT ERASING VARIABLES.

FOR ANY COCO WITH 64K OR MORE

FROM

DANOSOFT

P.O. BOX 124, STATION A, MISSISSAUGA, ONT.

L5A 2Z7, CANADA

(DOES NOT REPLACE BIG BASIC'S COMPREHENSIVE SYSTEM THAT ALSO PROVIDES EXPANDED DATA STORAGE AND MANY OTHER FEATURES.)

BABY BASIC

A TUTORIAL ON HOW TO ACCESS THE EXTRA MEMORY IN YOUR

COMPUTER TO STORE AND EXECUTE BASIC PROGRAM LINES.

Copyrighted in the United States
and Canada December, 1989 by
BILL DANIELS for DANOSOFT of
Mississauga, Ontario, Canada

(Note: This method does not replace Danosoft's BIG BASIC program which is an easier to use, comprehensive M.L. system that includes provision for expanded data storage and many other features.)

THE DISK

A disk with the basic program "BABY BASIC" is included with this tutorial and is required to use this presentation. The program contains seven subroutines which are necessary to enable the programming methods being described and which will be referred to below.

It is suggested that you run the file "BABY/BAS" for a quick look at the system, and then proceed with reading this tutorial. Later you will want to list the program to see the commented subroutines you will use in your own programs.

THE SEVEN SUBROUTINES ON THE DISK

Sub No.	Line Number	Purpose
1	1000	Set CoCo 3 Protected Memory
1	10000	Jump from Low to High Memory
2	20000	Jump from High to Low Memory
3	26000	Copy Program to High Memory
4	28000	CoCo 3 8K Memory Block Switcher
5	30000	Move System Stack in CoCo 3
6	40000	Reset Protect Changes
7	50000	Copy ROM to RAM - CoCo 2

THE POSSIBILITIES

All CoCo's with 64K or more contain extra unused memory that basic programmers can access using the techniques presented in this tutorial.

CoCo 3 memory is managed in blocks of 8k (8192 bytes) by the operating system. By programming in modules of 8k, you can execute basic program lines anywhere in the computer's normal 64k memory as well as from any other memory in the machine. Maximum in-memory total capacity of all program modules is about 472K in a 512k machine or 92k with a 128k computer.

Coco 2 users have 9728 free bytes at the top of the computer's memory. This provides a maximum of almost 38K for basic program lines in a CoCo 2.

Either model of CoCo can chain unlimited numbers of program sections into the computer from disk using the LOADM command in the manner described below without erasing variables.

THE CONCEPT

The primary concept presented is that all programs consist of groups of instructions to the computer. Any instruction group (such as one that displays a menu) can be considered a "module". A number of modules normally are combined to create an overall program.

By programming in a modular manner, or by breaking a program into appropriate segments, your program does not need to be located in one large chunk of memory. It can be scattered wherever memory is available throughout the computer.

Also, program execution speed is much improved when the computer is concentrating on just one module at a time. Furthermore, any program module created can be used in other programs as required; so that a library of routines can be developed.

THE METHOD

The method presented here is to start with a normal basic program in the lowest area of user memory. Then we take advantage of Basic's SAVEM and LOADM commands. These commands normally save a block of machine language from a designated block of memory.

But using the techniques we shall describe, you can save modules of basic programming with SAVEM and later, using LOADM, load the program module to any user selected memory outside of the normal area in low memory.

Then you can jump from your normal basic program to the module in high memory loaded with LOADM. In the high memory area you will execute your basic code as usual. When you are finished there, you jump back into your normal program in low memory.

As an added bonus, SAVEM and LOADM transfer programming between computer and disk drive much faster than SAVE and LOAD.

THE MEMORY

Coco 3 users will protect an 8K block at the top of user memory with the CLEAR command as per line 1000 in Subroutine No 1 (i.e. CLEAR 200, 24576); then switch in memory from outside of the CoCo 3's normal 64k to that area as required. (Subroutine No. 4). Your normal basic program in low memory controls the memory switching, as well as the LOADM and SAVEM of 8k memory blocks to that protected area.

A TABLE at the end of this tutorial lists all the 8K memory blocks that can be used in a CoCo 3. It is easy to access them with subroutine No. 4. You just put the block number you select into the subroutine, run it, and you will have a new 8K of memory in the protected memory area.

CoCo 2 users won't require setting protected memory with the CLEAR command like CoCo 3 users. Instead, they will simply jump from the normal basic area in low memory straight to the extra 9728 bytes available at the top of the CoCo 2's normal 64K space.

CoCo 2 users also must use subroutine No. 7 before proceeding with their programming. It copies the operating system from ROM to RAM and leaves the computer in the all RAM mode. The CoCo 2 must be in this mode in order to access the extra 9728 bytes of memory for programming or to poke any modifications to the operating system. (Subroutine 7 is not to be used in a CoCo 3.)

RESET PROTECTION

To preserve programming and to preserve changes to the basic operating system, both CoCo 2 and CoCo 3 users should use subroutine No. 6. Whenever the RESET button is pressed, the computer normally rewrites the operating system to the original ROM. The code in subroutine No. 6 is used to prevent this.

ADDRESSES OF THE LOW AND HIGH MEMORY AREAS

The low memory area for either a CoCo 3 or a CoCo 2 can start as low in memory as computer address 3584 ($E00). This is where you would be if you used the command PCLEAR Ø. Normal basic produces a "FC" error for PCLEAR Ø which you can overcome from direct mode or a program with the following: POKE &H968F,33: POKE &H96A3,33: PCLEAR Ø.

The CoCo 3 low memory area will end at 24575 ($5FFF), the point where the CLEAR command sets protected memory. The CoCo 3 high area starts at 24576 ($6000) and ends at 32767 ($7FFF), a total of 8K.

For the CoCo 2 the end of the low memory area is at 32767 ($7FFF). Its high area starts at 55552 ($D900) and ends at 65279 ($FEFF), a total of 9728 bytes.

HOW TO PROGRAM

Normal basic programming is done in low memory. The low memory routine will include the code found in Subroutine No. 1 of the BABY BASIC program found on the disk that comes with this tutorial.

Your program module for high memory will include subroutines No. 2 and No. 3. Although your high memory module will execute in high memory, it is actually created in low memory. Then it is saved as machine language with the SAVEM command and transferred to high memory by subroutine No. 3 which also processes it to make it usable. Subroutine No. 2 has the code you will use later to return to normal low memory.

Subroutines 1 and 2 simply set the computer system's "start of basic" variable to either the low or high areas of memory. The computer's operating system determines the start of basic from a 2-byte address found in memory locations 25 & 26 ($19 & $1A). The address of the start of basic is the value found in address 25 multiplied by 256 plus the value in address 26.

COPYING A PROGRAM MODULE FROM LOW TO HIGH MEMORY

Subroutine No. 3 is used to copy a program module in low memory to the high area via the disk using SAVEM and LOADM. This subroutine also contains one of the secrets of this system.

The first two bytes of every basic program line represent the memory address of the next basic line. If you use a machine language method such as SAVEM and LOADM to transfer basic programming to another area, the first two bytes of every line will have a wrong address.

But lines 26020, 26030 & 26040 of subroutine No. 3 call a basic operating system subroutine that re-addresses the transferred code and corrects the problem..

Subroutine No. 3 first saves a copy of the program module it is transferring to high memory as "Source" code. Programming located anywhere except in normal low memory cannot be changed or edited. Therefore you need to keep a source copy so you can make any necessary changes later.

USING "GOTO"

Understanding the GOTO command also is a key to learning how to execute programming from anywhere in memory. When the basic interpreter executes a GOTO, it first searches for the line number in higher memory. If the next line number is higher than the GOTO line number, then the search for the line is transferred to the start of basic.

If the computer believes the start of basic is elsewhere in memory, that is where it continues its search for the GOTO line. So simply by changing the operating system's start of basic variable in memory addresses 25 & 26 we can use GOTO to execute program lines anywhere. This is done in subroutines 1 and 2.

DUPLICATE LINE NUMBERING

Strangely enough, every program module can have the same line numbering if you want it to. Both the program module in low memory and the program module in high memory can have a line 10, for instance, and it will not matter. The line 10 that is executed is the one currently being pointed to by the address in the start of basic variable (25 & 26).

HOW TO AVOID THE PITFALLS

Yes, there are pitfalls to this system; but nothing you cannot overcome by being careful. If you are not careful, you can easily have a software crash and erase all programming currently in the user programming area.

There are five categories of pitfalls: Exiting your program from the high memory area; using certain commands in the high memory area; switching in wrong memory blocks; using string variables within a program line in high memory; and system stack conflict.

EXITING YOUR PROGRAM FROM HIGH MEMORY

If an error should occur while executing code in high memory, you must enter a GOTO to the sub that restores you to low memory. Failure to do so could result in a crash. Therefore, CoCo 3 users should use the ON ERR and ON BRK commands to set up an error subroutine that makes sure the start of basic (25 & 26) is poked to the normal low memory area values whenever an error or break occurs.

While in high memory, commands that clear the variable table will confuse the computer and cause a crash. When the variable table is cleared, the computer calculates a new start for its variable table based on the addresses of the start and end of your basic program. In high memory this calculation will be wrong.

Either from direct mode or from a program in the high memory area, you will have no difficulty if you do not use these commands: RUN, LOAD, SAVE, EDIT, CLEAR, PCLEAR, FILES. However, from direct mode, you can use LIST or SAVE "FILE",A. (That's SAVE with the ASCII option.)

SWITCHING WRONG MEMORY BLOCKS

CoCo 3 users must take care in selecting 8K memory blocks using subroutine No 4. If you select operating system blocks such as 56,60,61,62 & 63 you will crash. If you use the 40/80 column text screen do not select block 54. Similarly the high resolution graphics blocks 48 to 52 must be used with care; and you must not use block 53. See the TABLE at the end of this tutorial.

STRING VARIABLES IN HIGH MEMORY PROGRAM LINES

If a program line in a high memory module contains a variable statement like: A$ = "TEST", then the pointer in the variable table will always expect to find that variable in a high memory address. If you switch in another memory block from elsewhere in the computer or a program segment from disk, then your variable is going to be wrong.

There are two solutions to this problem. First: Program lines in high memory modules containing literal string statements with a + "" added to the end of the string will be copied to normal low memory string space when the program line is executed. (i.e. A$ = "TEST" + "").

Second: Poking a 33 to $B543 will force all strings in a program line to be copied to low memory string space when the program line is executed. (Normal value in $B543 is a 34.)

MOVING THE SYSTEM STACK -- SUBROUTINE NO. 5

There is a little known problem with the CoCo 3 operating system. The computer keeps its operating system variables in an area known by machine language programmers as "the stack". This stack must be available to the system at all times or the computer will crash. It is located in memory directly below the user cleared string space and most of the time there is no user problem.

However, whenever a user makes a change to the 40 or 80 column text screen, the text memory block No. 54 is momentarily switched by the operating system into logical computer space at addresses 8192 to 16383 ($2000 to $4000). On startup, string space is located directly below 32768. In the 40/80 text mode, a statement such as CLEAR 16500 will cause an immediate crash as text block No. 54 overlays the stack.

In the BABY BASIC system as described above, protected memory is set at 24576 which means string space now resides in memory just below that address and the stack is below that again. Therefore a command such as CLEAR 8000 will cause a crash.

If you are going to use less than 7000 bytes of string space, or you are using only the 32-column text

screen, then you will have no problem with the stack. But chances are, if you are writing a large program, you may want more than 7K of string variables. In that case you must move the stack out of harm's way.

Subroutine No. 7 uses a PCLEAR 1 to set the start of basic at 5120 ($1400). It presumes CoCo3 users will use the high resolution graphics and not require the CoCo 2 type low resolution graphics pages that would exist in memory just below the start of your basic program. This leaves 1536 bytes between 3584 and 5020 where you can put the stack which rarely requires more than 400 bytes.

Subroutine No. 7 sets the stack at 4096 ($1000). The stack grows from its starting point downward toward lower memory. Other pokes in the subroutine adjust commands like COPY and MEM which normally take the location of the stack into consideration when they make memory calculations.

Moving the stack also resets it and erases return addresses for the GOSUB and FOR/NEXT commands. Therefore using subroutine No. 7 in a program means it must end with a GOTO statement to continue with the program.

TO OUR VALUED CUSTOMERS:

Users of our programs are invited to comment. If we have missed some vital point, or something is not explained clearly enough, please contact us and tell us so we can ammend any reprints. We realize others may see things differently than us, and are concerned that users are satisfied with our programs.

Please send all enquiries, comments or requests for assistance to:

DANOSOFT
P.O. Box 124, Station "A",
Mississauga, Ont. L5A 2Z7
Canada

Telephone: (416) 897-0121

Immediate access by phone to a programmer is not always possible and writing is recommended.

MEMORY BLOCK TABLE

Memory is managed in a CoCo3 in blocks of 8K each. (1K = 1024 bytes; 8K = 8192 bytes)

The top 8 blocks are the normally used 64K.

BLOCK	USE	ACCESS by "BABY BASIC"
63	CoCo3 Extra Basic	No
62	Disk Basic	No
61	Color Basic	No
60	Extended Basic	No
59	User Programs/Variables	Yes
58	User Programs/Variables	Yes
57	User Programs/Variables	Yes
56	Basic's Variables/Buffers	No
55	Free Memory	Yes
54	Both 40 & 80 Column Hi-Res Text Screens	Yes if not using Hi-Res Text Screens
53	Secondary Stack Area	No
52	HGET/HPUT Buffer	Yes if not using HGET/HPUT
51	Hi-Res Graphics Screen	Yes if not using Graphics Screen
50	Hi-Res Graphics Screen	Yes if not using Graphics Screen
49	Hi-Res Graphics Screen	Yes if not using Graphics Screen
48	Hi-Res Graphics Screen	Yes if not using Graphics Screen
47 to 0	448K of Free Memory only in the 512K CoCo3	Yes
127 to 64	Another 512K of free memory only with CRC/ Disto 1 Meg Upgrade	Yes