

PROM Application Note

Programming CF-Series Flash Cards

This application note discusses the considerations in writing programs for the CF Series of Flash Cards for the Sharp PC-1270 Pocket Computer. It also addresses converting a program written for a RAM card to run on a Flash Card.

Table of Contents

Overview	2
Single-Bank Programs	2
Larger (Multi-Bank) Programs	2
Card Size Compiler Directive	3
How Bank Switching Works	3
Variables are Global	3
Global Routine Addressing in Multi-Bank Programs	3
Common Headers for Multi-Bank Programs	4
Bank Numbers for Larger Flash Cards	4
Design Considerations in Building a Multi-Bank Program	4
Sample Four-bank Program	5
Common Define File	5
Common Header File	6
The Initialization Flag Variable	7
Note on Common Header Variable Usage	7
Source Files for the 4 Banks	8
Space-Saving and Other Programming Considerations	10
Shared Variables	10
Frequently Used Constants	10
Line Numbers	11
Use of the IF ... THEN statement	11
Use of In-Line Logic	12
Backup Function Key	12
Use of Parentheses	13
Optimizing PC-1270 Programs	14
Eliminating Repetitive Calculations	14
Branching	14
Branching to Labels Representing Line Numbers	14
Branching to an S'BASIC Label	15
Perception of Speed	16
GOSUB versus GOTO	17
RESTORE	17
Converting an 8K RAM card program to a Flash card	17
Converting a 16K RAM card program to a Flash card	18
Flash Card Data Sheet CF-2H32M, CF-2H64M & CF-2H128M Cards	18

Overview

Single-Bank Programs¹

To write a program for the PC-1270 and install it on a Flash card, there are three steps that must be taken.

1. Write the program on your PC using `EDIT` or any other "text" or "ASCII" editor (the type you use to modify `AUTOEXEC.BAT` and `CONFIG.SYS` files). MS-DOS™ `EDIT` is furnished free with every copy of Windows so you will usually find a copy of it on your system. You can also use NotePad to edit source files.
2. Compile the program using the Sharp Basic Compiler `SBC.EXE` or other compiler to make an object file (which will have the file-type ".`SBC`"). If errors are detected by the compiler, edit the source file to fix the errors and recompile.
3. Program the Flash card with the compiled "object" file (the object file produced by the compiler will have the type `.SBC`) using `FLASH.EXE`.

Larger (Multi-Bank) Programs²

If your program is larger than 8K, there are six steps required.

1. Write the program using `EDIT`. Make a separate source file for each 8K bank, thus if you need four 8K program blocks for your application, generate four source files. If your program is named `VER1`, name the individual source files `VER1_0.LST`³, `VER1_1.LST`, `VER1_2.LST`, and `VER1_3.LST` corresponding to bank 0, bank 1, bank 2 and bank 3 (the program banks are numbered 0, 1, 2, 3, ...).
2. Make a common-define file named `VER1_DEF.LSH`. This file contains all the variable and constant declarations (by using "`#DEFINE...`"⁴), but may *not* contain any executable code. This file is included at the start of each of the bank source files. (Comments can appear before the point where the common-define file is included, but not executable code⁵.)

¹ A single-bank program has a code section (stored in a single bank) of up to 8K (stored in flash memory), and up to 8K of variables (stored in RAM).

² A multiple-bank program has two or more code sections (each stored in a separate bank) of up to 8K each, and one variable section.

³ In this application note, the sample program and its associated files have the "root" program name "sc4". Unless you are using the Windows 95 version of the SBC tools, the root name should not exceed 4 characters to allow for the addition of the standard extension for the various files types ("`_DEF`" for the common-define file, "`_HDR`" for the common-header file, and so forth).

⁴ See the SBC S'BASIC Compiler Reference Manual for a description of the `#DEFINE` and `#INCLUDE` compiler directives.

⁵ Executable code is any S'BASIC statement. Only comments with the " " comment character can appear in source files before the common-define and common-header files are included.

3. Make a common-header file named `VER1_HDR.LSH`. This file contains the common header information which controls the bank-switching code. This file is included in each of the bank source files immediately after the common-define file. No executable code can appear before the point where the common-header file is included.
4. Compile each source file to make an object file. If you have three source files, you will generate three object files: `VER1_0.SBC`, `VER1_1.SBC`, and `VER1_2.SBC`. (If you make changes to a source file, you need only recompile that source file to make a new object file with the changes. If you make changes to a header file, you need to recompile all the source files.)
5. Link the compiled object files together into a single file using the utility `BANKS.EXE`. This will generate the file `VER1.SBC` which is the complete program in one file.⁶
6. Program the Flash card with the `VER1.SBC` file using the `FLASH.EXE` program.

Card Size Compiler Directive

The `#CARD_SIZE` directive is only used if you are compiling a program for a RAM card. You do not need to specify the card size for a Flash card bank as the default card size of 15 is correct.

How Bank Switching Works

Because the Sharp PC-1270 can only address a limited amount of memory, CF-Series Flash cards use a "bank switching" technique to increase the amount of memory. Banks containing program code of up to 8K each can be switched into the PC-1270's "view". There virtually is no limit to the number of banks that can be switched into the PC-1270's view -- the PROM CF-Series 128K Flash card has sixteen 8K banks, and the 512K card has 64. The bank-switching is done in program code that is in the common header file.

In all CF Series Flash cards, the banks can be indirectly addressed, making it easy to access them from tables of data or routine names.

Variables are Global

The PROM CF series of Flash cards provides a separate 8K block of RAM to store the variables. This area cannot be used to store code. Even though programs are built in banks with separate source files, all variables are "global" (visible in every bank). If a variable is modified in one bank, it will have the new value in all banks.

Global Routine Addressing in Multi-Bank Programs

By incorporating routine names into a common define file which is included in each bank (see example later), you can easily move routines from one bank to another, and you can call them from any bank. The S'BASIC Label for a routine is defined in the common define file.

⁶ Steps 4 and 5 can be combined into a single step by using the "-h" command-line option of the SBC S'BASIC Compiler. See the SBC S'BASIC Compiler Reference Manual for details.

If you move a routine from one bank to another to optimize space, you need only change the entry in the common define file and recompile.

Common Headers for Multi-Bank Programs

In order to use the bank-switching feature of Flash cards, the first several lines of code in each bank must be identical, as shown on the following pages. Once these lines are in place, they require little or no maintenance. Although there is a little programming time involved in writing a common header, the freedom it provides in programming is well worth the effort. The bank-switching technique for a 128K Flash card with 16 banks is just as simple as that for the 32K Flash card with 4 banks. As your program grows, you can move to larger cards and add banks of program without having to change the original program.

Bank Numbers for Larger Flash Cards

Up to 16 banks can be addressed by assigning one of the following values to the variable `currBank` (which must be `#defined` as `Z$(1)`): "0" (the number zero), "1", "2", "3", "4", "5", "6", "7", "8", "9", "J", "K", "L", "M", "N", and "O" (as in "Oscar").

When assigning the S'BASIC Label to a routine in the common define file (see `SC4_DEF.LSH` below), the last digit of the S'BASIC Label is the bank number. The S'BASIC Label must be unique within a bank, e.g., you can have "A0" and "B0". Both occur in bank 0. If you had 10 routines in bank 3, for example, you could identify them with "A3", "B3", "C3", ..., through "J3". These S'BASIC Labels can be as short as two characters (and it's just a waste of space to make them longer) as long as they are unique within a bank.

The logical name you assign to the label should be meaningful to you, the programmer. Generally we prefix all routine names with an "R_", e.g., "R_CHK BANK3". Thus, by inspecting the name one can tell it is a routine name (because of the prefixed "R_", it checks the bank, and the routine is located in bank 3.

Design Considerations in Building a Multi-Bank Program

It is worth giving some consideration as to how you divide your program into separate banks. A well-thought-out and logical method will save programming time as well as make the program easier to maintain and enhance in the future.

The best method is really dependent upon the particular program and your programming style, however, there are some general rules and specific suggestions below which may be of aid.

There is no reason not to use all or most of the banks in a multi-bank card. The program may actually run faster if you spread it out over several banks. The tiresome task of searching for space is no longer required with the higher capacity flash cards. Generally programs that are spread out and have lots of space available are easier to write, debug and maintain. (A major limitation of the pocket computers of the past has been the space -- many programs just fit, with no extra space, and only then with the help of a shoe horn!)

It is required that the initialization routine which dimensions arrays be located in bank 0. The simplest split of a program is to put this initialization routine in bank 0 and the rest of the program in bank 1. (Only the `DIM` statements have to be in bank 0, the initialization routine can

call routines in other banks to load data into the arrays or do other initialization tasks after the arrays are dimensioned.)

Another approach is to put one (or more) separate routines in each bank. This way, each source file deals with a specific routine. Using this technique, it is possible to have more than one person working on a project at the same time.

Although dividing a program into banks may seem a little tedious at first, it is well worth the effort. All serious computer programs today are built in modules generating several (sometimes hundreds) of object files which are linked together. The task of maintaining and enhancing your program is greatly simplified when it is divided into logical source files. With larger programs, multi-bank design also saves considerable time. If changes are required in only one bank, you can make the changes to the source file for that bank, recompile it, and relink the program without having to recompile the other banks.

Another advantage of multi-bank programs is the ability to make several versions of a program easily. All the banks except one could be the same for different versions, only the version-specific bank would be different. Thus to make 15 versions of a 4-bank program, one would need three "standard" files and one version-specific file for each of the 15 versions, or a total of only 18 files to make 15 different versions of a 4-bank program.

Sample Four-bank Program

This simple little program has four banks. Pushing any function key causes the program to identify the key pressed, and then switches through all the banks before ending. This program requires a 32K Flash card to run.

The program can be compiled and linked with one command, "SBC -H sc4", which will generate the individual object files for each bank (sc4_0.SBC, sc4_1.SBC, sc4_2.SBC, & sc4_3.SBC) and then links these four files into the final file, sc4.SBC, which is used to program the Flash card.

If you don't use the "-H" option with SBC, you can then run BANKS.EXE to link the four individual object files into the final sc4.SBC.

Common Define File

SC4_DEF.LSH

```
' constants defined first

#define YES          1
#define NO           0

' A-Z variables defined here
' last three reserved

#define Routine      X$
#define OldBank      Y$
#define Init_flag    Z

#define CurrBank     Z$(1)
```

```

#define KEY_A      "KA0"      ' last char is bank
#define KEY_B      "KB0"      ' change the last digit
#define KEY_C      "KC0"      ' to represent the bank
#define KEY_D      "KD0"      ' number & move the
#define KEY_J      "KJ0"      ' associated code to
#define KEY_K      "KK0"      ' the new bank.
#define KEY_L      "KL0"
#define KEY_M      "KM0"

#define R_CHK BANK1      "A1"
#define R_CHK BANK2      "A2"
#define R_CHK BANK3      "A3"

```

Common Header File

SC4_HDR.LSH

```

CheckInit:
    if Init_flag = NO \
        gosub "INIT"          ' must be in bank 0
    return

"A" \
    gosub CheckInit :
    Routine = KEY_A :
    goto LongGoto

"B" \
    gosub CheckInit :
    Routine = KEY_B :
    goto LongGoto

"C" \
    gosub CheckInit :
    Routine = KEY_C :
    goto LongGoto

"D" \
    gosub CheckInit :
    Routine = KEY_D :
    goto LongGoto

"J" \
    gosub CheckInit :
    Routine = KEY_J :
    goto LongGoto

"K" \
    gosub CheckInit :
    Routine = KEY_K :
    goto LongGoto

"L" \
    gosub CheckInit :
    Routine = KEY_L :
    goto LongGoto

"M" \
    gosub CheckInit :
    Routine = KEY_M :
    goto LongGoto

```

```

LongGoto:
  CurrBank = right$(Routine, 1) :
  goto Routine

LongGosub:
  OldBank = CurrBank :           ' long gosub's take 2
  CurrBank = right$(Routine, 1) : ' stack positions.
  gosub Routine :               ' Stack has only 5 total.
  CurrBank = OldBank :
  return

```

The Initialization Flag Variable

In the above example, we use the variable `Init_Flag` as an initialization flag. The first time the program is run after being installed on the Flash card, it will equal `no` or 0 and we need to dimension the arrays and do any other initialization steps. Note that the initialization routine sets the `init_flag` to `yes` or 1, a non-zero value because, once the calculator has been initialized, we don't want to initialize it again. You can `#define` the variable `Init_Flag` to any single-letter variable from "A" to "Z" (do NOT use an array variable for `Init_Flag`). Make sure you do not use the initialization variable for any other purpose.

The variable used for the initialization flag must be a single-letter A-Z variable, because it is checked before any array variables are dimensioned. If you try to read an array variable before the array has been dimensioned, `ERROR 3` results.

Note on Common Header Variable Usage

We have used the variables `X$` and `Y$`, and the `Z$(9)` array for the bank-switching operations, however, you can use any other lettered variable or array variables if you wish.

Make sure the arrays are dimensioned in the initialization routine, and change the entries in the common-define file for `OldBank`, `CurrBank`, and `Routine`. Of course, these variables should not be used for anything else.

With Flash cards, you have almost unlimited RAM space, so there is no reason not to use discrete variables for these items. The common practice of "sharing" variables to save space is no longer required with a Flash card.

Source Files for the 4 Banks

SC4_0.LST

```
#include "sc4_def.lsh"           ' must be first
#include "sc4_hdr.lsh"           ' must be before any code

KEY_A \
  pause "THIS IS KEY A" :
  goto TestCard

KEY_B \
  pause "THIS IS KEY B" :
  goto TestCard

KEY_C \
  pause "THIS IS KEY C" :
  goto TestCard

KEY_D \
  pause "THIS IS KEY D" :
  goto TestCard

KEY_J \
  pause "THIS IS KEY J"
  goto TestCard

KEY_K \
  pause "THIS IS KEY K"
  goto TestCard

KEY_L \
  pause "THIS IS KEY L"
  goto TestCard

KEY_M \
  pause "THIS IS KEY M"
  goto TestCard

TestCard:
  pause "IN BANK 0"

  Routine = R_CHKBank1 : ' check bank 1
  gosub LongGosub

  Routine = R_CHKBank2 : ' check bank 2
  gosub LongGosub

  Routine = R_CHKBank3 : ' check bank 3
  gosub LongGosub

  print "DONE"
  end

"INIT" \
  clear : ' must appear in bank 0
         ' but can be anywhere in it
  dim z$(9)*1 :
  ' ' dimension other arrays here
  '
  ' ' put any other initialization
  ' ' stuff here
  Init_Flag = YES :
  return
```

SC4_1.LST

```
#include "sc4_def.lsh"
#include "sc4_hdr.lsh"

R_CHKBank1 \
  pause "IN BANK 1" :
  return
```

SC4_2.LST

```
#include "sc4_def.lsh"
#include "sc4_hdr.lsh"

R_CHKBank2 \
  pause "IN BANK 2" :
  return
```

SC4_3.LST

```
#include "sc4_def.lsh"
#include "sc4_hdr.lsh"

R_CHKBank3 \
  pause "IN BANK 3" :
  return
```

Space-Saving and Other Programming Considerations

Shared Variables

Because the RAM memory area is not shared with the code space, you can use variables freely without losing space for your program. Thus the use of “shared” variables (same variable used by different routines) no longer saves you any program space. You have 8K of RAM which can hold nearly 1000 real numbers. By using discreet variables for each of your routines, you can minimize the introduction of bugs through shared variables.

A common method is to use a separate array for the variables in each routine. The first routine may use the C() array for input and working variables, the second routine the D() array, and so forth. The A-Z variables can then be used for temporary variables and for storing constants used frequently.

There are 25 numeric arrays (B() through Z()), and 25 string arrays (B\$() through Z\$()). B() and B\$() arrays are separate and can exist in the same program. The Z\$() array is reserved for the bank switching in Flash cards. The A() array can also be used, but A() and A\$() are not separate. Special rules apply to the A() array.

Frequently Used Constants

In financial software, for example, the constant 100 is used frequently to round computed results to the nearest cent. If the constant is used 100 times in a program, this consumes 300 bytes of program space (“100” is 3 bytes x 100 times = 300).

If you assign the value of 100 to a single letter variable (say “E”) in the initialization routine, you can reduce the amount of code space required from 300 bytes to 106 (“E” is 1 byte x 100 times plus 6 bytes to make the assignment), saving almost 200 bytes of code space.

Another example is printing FORMAT strings. Often used formatting strings can be assigned to a string array and then referenced with array variables. Assume we have DIMensioned the F\$(4) array using the default 16-byte length for each element:

In the include file that contains the program's #defines, put the following:

```
#define      Fmt4V2          f$(0)
#define      Fmt6V2          f$(1)
```

In the bank 0 source file, make sure to dimension the F\$() array and assign the desired formatting strings to the items. This can all be done in the INIT routine, e.g.,

```
clear :
dim Z$(9)*1,      \
    F$(1) :
Fmt4V2 = "####.##" :
Fmt6V2 = "#####.##" :
Init_Flag = YES :
return
```

Then, in the program we can write

```
PRINT USING Fmt6V2; "TOTAL"; T
```

Line Numbers

There is no limit to the length of an S'BASIC line of code. You could write a 1-line program with the line nearly 8K long, subject to certain limitations.

Each new line number consumes 2 bytes of space for the line number. Each line number you remove by concatenating statements with the colon line-continuation symbol ":" saves 2 bytes. Removing 100 or more line numbers (entirely possible in some programs) saves 200 bytes.

The SBC Compiler assigns a line number to every statement that is not continued from the previous line, i.e., the previous line did not end with one of the line-continuation symbols ":" or "\".

Line numbers are not required and can generally be removed except in these situations:

IF statement. If the condition in the **IF** statement is false, program execution will resume at the next line number.

INPUT statement. If no entry is made at an **INPUT** statement, program execution will resume at the next line number.

A label must appear at the head of a new line so it can be used as the destination of **gosub** and **goto** statements. This is true for both SBC labels and the S'BASIC labels.

You can have both an S'BASIC and SBC label on the same line. The SBC label must occur first, e.g.,

```
SBCLabel:           ' this is an SBC label which references the line
                   ' number and can be addressed within this bank
R_ROUTINE_ALPHA \ ' this is a S'BASIC label defined in the common-
                   ' define file and used in LongGoto and LongGosub
let .....
```

DATA statements must appear at the start of a new line.

Use of the IF ... THEN statement

The S'BASIC **IF .. THEN** statement has some non-standard properties.

Use of the keyword **THEN** is really never required. The statement

```
IF X > 5 THEN GOTO Label_A
```

can be written as

```
IF X > 5 GOTO Label_A
```

Similarly, the **IF** condition can be followed by any BASIC verb without an intervening **THEN**. If you make an assignment as a condition of an **IF** statement, you must use the **LET** verb immediately after the condition, e.g.,

```

IF Y * 5 > T + W    \
  LET R = 800 :
  Z = 3.4 : ...

```

If you omit the `LET` statement, a run time error may result or the remainder of the line (the portion following the omitted `LET`) will be skipped at run time.

As is true for most `IF` statements, the condition evaluates to either a true or false and the statements following the condition are executed if it evaluates to true. If the condition is false, execution starts at the head of the next BASIC line (the next line that will be assigned a number by the compiler).

“True” for Sharp S’BASIC means any value greater than 0. If the “condition” has any positive value, it is “true”. If the “condition” evaluates to 0 or any negative number, it is false. Thus if X has a value of 1, the following line will set A to a value of 5:

```
IF X LET A = 5
```

If X has a value less than or equal to 0, the condition is false. Thus, if X had a value of -1 in the above statement, the assignment to A would not occur.

Use of In-Line Logic

Because an `IF...THEN` statement requires a new line number, it is often easier to use “in-line” logic in S’BASIC. If we want to assign a value to A of 10 if $Z \neq 0$, or a value of 0 to A if $Z = 0$, we could write

```

A = 0 :
IF Z <> 0    \
  LET A = 10

```

or

```

A = 10 * (Z <> 0) :
...

```

In this latter case, we can put the statement anywhere in a line and do not need a new line after it. If Z is not equal to zero, the expression $(z \neq 0)$ evaluates to 1, which, multiplied by 10, assigns a value of 10 to A. If Z is equal to 0, then the expression $(z \neq 0)$ evaluates to 0, 10 times 0 is 0, so 0 is assigned to A.

Backup Function Key

A useful feature of the Sharp S’BASIC language is its ability to use a variable (as well as an expression resolving to a number) as the argument for a `goto` or `gosub` statement. This makes it easy to make one of the function keys act as a backup key.

Just before each input prompt, you need to store the line number that you want the program to go to if the user presses the backup function key instead of `[ENTER]`.

The code for the backup function key simply executes a `goto` to that line number. In the following code segment, we’ve used the variable `bptr` to store the backup line number:

```

Inp_factor_1:
  bptr = Inp_factor_0 :

```

```

INPUT "FACTOR 1"; Fact1

Inp_factor_2:
  bptr = Inp_factor_1 :
  INPUT "FACTOR 2"; Fact2

```

When the calculator is waiting for entry at the prompt "FACTOR 2", the variable `bptr` has the line number where we want the calculator to go if the user pushes the backup function key instead of the ENTER key. Note that we assign the label `Inp_factor_1` to `bptr`. At compile time, the compiler will replace `Inp_factor_1` with the line number the compiler has assigned to it.

Now we need the following code for the backup function key (we've used function key "D" in this example):

```

"DD" :
GOTO bptr

```

If you're using multiple banks for your program, you can see the advantage of having all of the input prompts in one bank for a particular routine. (If they're not, you need to assign the bank number as well as the line number before each input prompt and include bank-switching code for the backup function key.)

Use of Parentheses

Use of parentheses is conventional in S'BASIC with the exception that they frequently are not required when passing simple arguments to functions. For example,

```

LenName = LEN (Name$)      can be written as      LenName = LEN Name$

```

Similarly, virtually all functions that return a value can be used without parentheses around the argument if it is a single variable. Compound arguments require parentheses to make sure that the argument is evaluated before its value is passed to the function. Thus the parentheses are required in

```

Y = LOG (Y + 2 * Z)

```

Optimizing PC-1270 Programs

The Sharp PC-1270 has a relatively fast CPU (clock speed is 1 MHz). It also has enhanced floating-point numeric processing.

For speed, spreading a program out among several banks will often result in faster execution. In-line code (as opposed to calling subroutines) will result in faster execution.

Following is a list of some optimizing techniques for PC-1270 programs.

Eliminating Repetitive Calculations

Small programs generally run faster than larger ones. If you can optimize your formulas and algorithms to reduce their size and eliminate unnecessarily repeated calculations, speed and size are both improved. This is especially true of calculations that are done many times. For example, the following simple loop

```
A = 0 :
FOR K = 1 TO 100 :
    B = 128
    A = A - C(K) :
NEXT K
```

would run faster (and takes no more space) if changed to

```
A = 0 :
B = 128                                ` do this once before we start loop
FOR K = 1 TO 100 :
    A = A - C(K) :                      ` this line executed 100 times
NEXT K
```

Branching

Branching in the PC-1270 S'BASIC language includes the commands GOTO, GOSUB, ON GOTO, ON GOSUB, and IF...THEN.

Many BASIC programs suffer from excess branching which makes them hard to understand, and difficult to debug and maintain. In the PC-1270, it also slows programs down considerably because the calculator spends so much time scanning for line numbers or labels.

It's best to eliminate branching wherever possible, and to optimize branching that is required. These two changes will often both improve execution speed and reduce program size. Smaller programs not only take less space (and will fit on a smaller Flash card) but are also often easier to maintain and execute faster.

In the SBC Sharp BASIC Compiler, there are two methods of branching, and understanding how the PC-1270 performs them will help you optimize your program.

Branching to Labels Representing Line Numbers

Branching to a label representing a line number is the same as in traditional BASIC, e.g., GOTO Point_B, OR ON X GOSUB Type_1, Type_2, Type_3. When a branch is specified, the PC-1270 determines whether the new line number is greater than or less than the current line number and then scans the program in that direction looking for the specified line number.

(Although there are only labels and no line numbers in the source file, remember that the compiler replaces the labels with line numbers. Thus when the PC-1270 executes the code, it's looking for actual line numbers.)

If you are on line 100 and call a subroutine on line 200, it will scan upwards until it finds line 200 (or will report `ERROR 4` if it cannot find it). The amount of time it takes to find line 200 is based solely on the number of characters between line 100 and line 200 (not the number of lines).

Thus, to minimize the time to make a subroutine call, you should call the subroutine by a label representing a line number (as opposed to an S'BASIC label [see below]) and locate the subroutine near where it is called. If a subroutine is called from more than one place, you should locate it near the place where it is called most often or where time is critical. Because there are no line numbers in your source file, it is easy to move the entire subroutine to any location.

If you use a subroutine call in a loop where it is called 5 or more times, consider eliminating the subroutine and putting the code inside the loop.

Similarly, a `GOTO` to a nearby line number will execute quickly, whereas a `GOTO` to a distant line number takes more time.

Both the `GOTO` and `GOSUB` will accept labels and/or numeric variables as the line number, e.g.,

```
GOTO B
```

The next line adds the value in `B` to the line number assigned to `Point_A`, and then goes to that line.

```
GOTO Point_A + B
```

The next line goes to the line number assigned to `sum_totals` if `R` is not equal to 0, or to the next line number after `sum_totals` if `R` is equal to 0.

```
GOSUB Sum_totals + (R = 0)
```

Branching to an S'BASIC Label

S'BASIC Labels are specific to Sharp S'BASIC and not usually found in BASIC languages. An S'BASIC Label starts with a letter, and can be followed by 0 or more letters or numbers, e.g., "A", "A2", "TOT8", etc.

This is how the PC-1270 starts a routine when one of the function keys is pressed. When the upper-left function key is pressed, the calculator scans for the label S'BASIC Label "A" at the start of a line within the current bank. If found, it starts executing the program from that point (if not found, it reports `ERROR 4`).

When a branch to an S'BASIC Label is executed, the calculator has no idea where the label is located, so it has to go to the very first line of the program in the current bank and start scanning from there. Thus, if lines or routines identified with an S'BASIC Label are located near the start of the program, the scanning time required to locate them will be minimized.

Using S'BASIC Labels with `GO SUB` and `GO TO` is generally slower than using labels representing line numbers unless the S'BASIC Labels are located near the start of the program.

When calling subroutines in another bank, S'BASIC Labels should be used for several reasons: First, the bank switching takes place within the first several lines of the bank so the PC-1270 will always have to scan from the beginning of the program. Second, because the line numbers are assigned at compile time, you won't know the line number unless you specifically assign it (which is less efficient). And third, using an S'BASIC Label will minimize future maintenance. Routines should be assigned to S'BASIC labels in the common-define include file.

Perception of Speed

The user's perception of speed needs to be understood to optimize a program so it will appear to be fast. Some users measure the time from the last prompt to the first computed result. In this case, the technique is to make as many calculations as you can between prompts before you get to the last prompt. When the last prompt is presented, you may have to do only one more calculation to present the first computed result. This method slows down the calculator between prompts, but this is often not noticed by the user.

As an alternative, the prompting routine can be optimized to let the user make entries as quickly as possible, and then calculations are done.

(An interesting note is that if the computed result appears too quickly, the calculation may appear trivial to the user. If a programmable calculator is really needed to perform the calculation, it ought to take at least a respectable amount of time to complete it.)

A more scientific user might measure the speed from the time the function key is pressed to the first (or last) computed result. This is more difficult to minimize and takes good programming techniques.

Generally, PC-1270 programs have one or a few main routines and other "what-else-does-it-do" routines. The perception of speed is usually based on the "main" routine(s) and it often does not matter whether the "what-else-does-it-do" routines are fast at all. If the main and other routines call a common subroutine, you should locate it near the calls in the "main" routines and call it by line number for the best results. Similarly, the "what-else-does-it-do" routines should be located at the end of the source file (with the highest line numbers).

If you locate a subroutine in another bank, put it at the very beginning of the bank to minimize the time it takes to find it.⁷

When printing output, you can use subroutines freely because the calculator usually spends most of its time waiting for the printer and no decrease in speed will be detected.

A common misconception is that fresh batteries make the PC-1270 run faster. In fact, the speed of the calculator is controlled by its oscillator and as long as it runs at all, it will run at the same speed regardless of the condition of the batteries.

⁷ The call to the routine will be with an S'BASIC label, and the PC-1270 always scans from the first line of the program when searching for an S'BASIC label.

The LCD screen will update faster with fresh batteries, and this gives the appearance that the machine is running faster.

GOSUB versus GOTO

When possible, `GOSUB` is generally preferred to `GOTO` for several reasons. The program flow is easier to keep track of because the program will return to the statement immediately after the `GOSUB` call to continue executing. The `GOTO` jumps to another place and never comes back.

The scanning time for a `GOSUB` or `GOTO` to find a line is the same, however a subroutine call will return to the statement following the `GOSUB` call very quickly with no scanning time.

S'BASIC allows five levels of subroutine nesting, i.e., you can go five levels "deep" when calling subroutines. When executing a "long" `gosub` to another bank, two levels are used (one to switch to the other bank and one for the actual subroutine call).

Thus, if you limit yourself to one "long" `gosub`, you still have 3 levels of subroutines you can use. If you nest two long `gosub`'s, you have only one subroutine level left. Finding errors caused by exceeding the maximum level of nesting can be tricky. The PC1270 does not call an error when the excessive `gosub` is encountered, but rather when the 6th `RETURN` is issued.⁸

RESTORE

`RESTORE` can reference either a label representing a line number or an S'BASIC Label, and scans for it in the same manner as `GOTO` and `GOSUB`. A clever way to `RESTORE` to a series of `DATA` lines based on a variable is to `RESTORE` to the sum of the variable and the Label pointing to the first of a series of `DATA` lines, e.g.,

```
Blue_data:
  DATA 123, 456, 789      \ type 0 data
  DATA 321, 654, 987      \ type 1 data
  DATA 213, 546, 879      \ type 2 data

A = 2 :                    \ A is type (0, 1 or 2)
RESTORE Blue_data + A :    \ point to correct line
READ B, C, D : ...        \ get the type 2 data
```

Converting an 8K RAM card program to a Flash card

If your program fits on an 8K RAM card, it is very easy to convert it to a one-bank Flash program.

If you use arrays (the `DIM` command) in your program, you need to add a new array at the beginning of the list. This new array must be first on the list and immediately after the `CLEAR` statement.

For example, if your program contains the following statements:

⁸ When the 6th `gosub` is encountered, the address for the 1st `gosub` is bumped off the stack as the new address for the 6th is added. Thus, when the 6th return is encountered, there is no information on the stack and the program stops running, reporting error 5.

```
CLEAR : DIM A(27), B$(9), K(20)
```

change it to read

```
CLEAR : DIM Z$(9)*1, A(27), B$(9), K(20) :
```

We've used `z$()` for this array, however, if you have already used the `z$()` array, you can use any other unused letter from `B$` to `Y$` for this array (do not use the letter A).

Do not use any of the elements in the `z$()` array, except as specifically shown below for multi-bank programs. The `z$()` array consumes 16 bytes of RAM space.

Converting a 16K RAM card program to a Flash card

If your program requires a 16K RAM card, it is more likely than not that it will have to be separated into 2 banks before it can be put on a Flash card. If you are using the SBC Compiler, you can tell quickly whether the program will fit within a single 8K Flash bank by setting⁹ the `#CARD_SIZE` directive to a value of 15 and recompiling the program. If the code segment is larger than 8K, the compiler will report an error in which case you need to separate the program into two or more banks (see below).

When the compiler reports an out-of-memory error, the line number reported is the line at which the 8K bank became full. If the line is near the end of your program, you may be able to squeeze enough room out of your program to keep it in one bank.

If you have one, big 16K RAM card program with one routine, we suggest you put the initialization routine in bank 0, all the input and output statements in bank 1, and all computational tasks in bank 2. At the end of the inputs in bank 1, you would execute a subroutine call to the computational routine in bank 2. This method allows you to work on the input flow and output presentation all in one source file, and on the computational aspects in another source file. A further step, using a larger Flash card, would be to put the input flow in one bank, and the output section in its own bank.

For more information, contact

**PROM Software, Inc. 1820 Shelburne Road, Burlington, VT 05406-4027
800 843-7766 802 862-7500 Fax 802 862-8357**

Flash Card Data Sheet **CF-2H32M, CF-2H64M & CF-2H128M Cards**

These three Flash cards are high speed, very high capacity, permanent memory cards for the Sharp PC-1270 Pocket Computer with CMOS logic and Flash ROM. The CF-2H32M provides 32K of program space, the CF-2H64M, 64K, and the CF-2H128M, 128K. All cards provide 8K of RAM for variable storage.

Features:

⁹ Alternatively, you can remove the `#card_size` directive because the default value is proper for a Flash card bank.

- Large data storage capacity is ideal for applications that require large amounts of data storage e.g., look-up tables, price lists, data tables, and so forth.
- Program is stored permanently (but can be changed) with the inexpensive and fast F512A Flash Card Programmer. No loss of program when batteries are changed or card is left out of calculator.
- Program is stored in up to 16 banks each. Each bank holds 8K and banks can be indirectly addressed.
- "Data-saver" feature protects RAM data when batteries are changed or card is moved from one calculator to another.
- Typical programming time is 5.5K of program per second or less than 10 seconds per card for a 50K program, over 300 cards an hour.
- Cards can be automatically assigned a unique serial number when programmed.
- Designed for use with the SBC compiler for the Sharp PC-1270 Pocket Computer. Application notes and technical assistance are available from PROM). F512A Flash Card programmer is required to program cards.
- Maximum security is provided to protect your programs. The program on a CF-Series Flash card can only be executed by the Sharp PC-1270 calculator. The F512A Flash Card programmer can only erase and program cards. The security of your program is guaranteed because no "read" function is provided. (Program verification is done "on-the-fly" as card is programmed.)
- The inexpensive PC-1270 Developer's Kit includes the SBC S'BASIC compiler, linker, and F512A Flash card programmer, everything you need to produce software on CF Series Flash cards for the PC-1270.
- F512A Flash Card Programmer is inexpensive and easy to use. Connects to any serial port (COM1, 2, 3 or 4). Compact size (1¼" high by 3" wide by 5" long), rugged, long-life, extruded aluminum case is designed for years of use. Operates on 115 vac or 12 vdc allowing use in the field with portable computers.

CF-2H32M, CF-2H64M & CF-2H128M Specifications:

Size: 0.150" high x 1.650" wide x 2.125" deep (3.8mm x 42 mm x 54 mm)
Fits a Sharp PC-1270 Pocket Computer.
Case is high-impact injection-molded plastic (matches the Sharp PC-1270).

Flash-gold circuit board with all surface-mount components.
Built in U.S.A. to ISO 9000 standards.

High-speed, CMOS logic design with very low current drain.

Program capacity: 32K (CF-2H32M), 64K (CF-2H64M), & 128K (CF-2H128M). Data capacity: 8K of CMOS RAM on all models

Erase and Programming Times:

Cards program at the rate of about 5.5K per second. Erase times vary from 1 to several seconds, depending upon the amount of information being erased. Approximate times to program a card are shown below. All A-Z and array variables can be initialized using the Automatic Variable Initialization option in an additional 1.6 seconds.

	CF-2H32M	CF-2H64M	CF-2H128M
Capacity	32K + 8K	64K + 8K	128K + 8K
# of Banks	4	8	16
50% Full	3.0 secs	6.0 secs	11.8 secs
75% Full	4.5 secs	8.9 secs	17.7 secs
100% Full	5.9 secs	11.8 secs	23.5 secs

Data-Saver feature:

Contents in RAM memory are saved for about 10 minutes when card is removed from calculator or batteries are changed. Program is retained permanently.

Security:

Program cannot be read from card by F512A Flash card programmer. Program cannot be retrieved from PC-1270 if the Sharp password protection is used (the programmer automatically issues a warning if a program is not password-protected).

For more information, contact
PROM Software, Inc. 1820 Shelburne Road, Burlington, Vermont 05406-4027
800 843-7766 802 862-7500 Fax 802 862-8357