Interfacing KIM/SYM/AIM/OSI with Basic

Jim Butterfield, Toronto

Basic is a convenient and flexible language; but it isn't too fast. Machine language is fast, but rigorous to write. You can get the best of both worlds if you can make the two languages work together.

A hybrid program of this type invariably starts its run in Basic. Basic prints out the program title, and prompts the user for the detail of the job to be done. When it reaches a part where time is important, it will zip into machine language.

Getting there

Basic enters machine language by means of the USR function. The machine language coding will be written as a subroutine whose final command is RTS, signaling a return to Basic.

USR is a function: it's similar to SQR for square roots, RND for random numbers, etc. This means you can't start a Basic statement with USR: it must be part of an expression such as X = USR(0), or PRINT USR(99).

USR takes an argument: USR(6) passes a value of 6 to the machine language program. It returns a value: USR (6) might give back a value of say 12 to Basic. You don't need to use either of these. The machine language program can ignore the argument, and the Basic program can decide not to use the returned value. They are there if you need them.

Single routine

If you want one machine language subroutine and no more, it's quite easy. Poke the USR vector with the address of the subroutine. After that, the USR function will zip to that address every time it's used. The USR vector may be found at the following locations:

```
KIM: 0004 and 0005
SYM: 000B and 000C
AIM: 0004 and 0005
 OSI: 000B and 000C
```

Check your Basic manual, if possible, to confirm that these are the locations that apply to your Basic package.

The address goes in low order first, as usual. Don't forget that Basic uses decimal numbers rather than hexadecimal. An example: to set up the address of the subroutine at OF22 on the KIM or AIM, you would code in Basic: POKE 4,34 : POKE 5,15. This needs to be done only once. After that, any USR reference takes you to OF22. For SYM or OSI, you'd code: POKE 11,34 : POKE 12,15.

Multiple machine language routines

There are several ways you can handle this.

You could repeat the pokes to the USR vector before each call. This is easy to code, but not lightning fast - POKEs from Basic are much slower than machine language.

If your routines come up in a certain order, you could have each machine language subroutine set up the next. A POKE in Basic is roughly equivalent to a STA machine language instruction. Each routine could set up the vector for the appropriate next USR entry.

Finally, you could keep a single entry point and have your machine language program decide which way to go on the basis of information supplied by Basic. This is discussed in the next section.

Single Entry Fanout

There are several ways that Basic could signal the type of job it wants done. It could POKE a location with a value that machine language could read and act upon.

A more complex method is to pass the information in the USR argument. USR(1) would mean, do job 1; USR(2), do job 2; and so on. This is a little trickier, since the argument is held in floating point. The next section will give more details on how to interpret it.

Passing parameters via the argument

When the function USR)6) is given, the argument- in this case, 6 - is placed in the floating-point accumulator. Later, when you return from machine language, the value in the floating-point accumulator is accepted by Basic as the value of the USR function. If you leave the floating-point accumulator alone, the value that went in comes back out. It's handy to keep in mind that you can use an expression as the argument: USR (X + Y*3 -2) is quite acceptable.

The floating point accumulator is at the following locations:

```
KIM - 00AE to 00B3
SYM - 00B1 to 00B6
AIM - 00A9 to 00AE
OSI - 00AC to 00B0
```

Note that the OSI floating point accumulator is one byte shorter than that of the other machines.

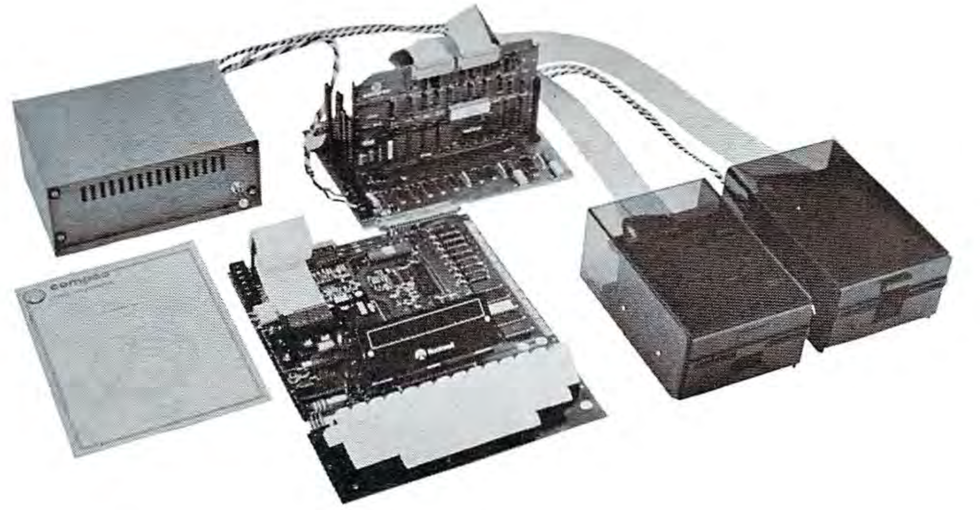
The first location is both zero flag and exponent. If it's zero, the whole number is zero and you don't need to look any further. If it's non-zero, it holds a binary exponent offset by $80. That means if it contains hex 80 or less, the number is a fraction less than 1. If it contains hex 81 or more, the number is greater or equal to 1. Don't worry about the details unless you have a mathematical leaning. It's useful to know, however, that you can double a number by adding one to the exponent, and halve it by subtracting one.

The next four locations are called the *mantissa* and hold the number itself. The number is always normally arranged so that its first 1-bit is in the high-order bit position of the mantissa. So numbers like 3, binary 11, and 6, binary 110 will have exactly the

compas
microsystems

P.O. Box 687
224 S.E. 16th Street
Ames, Iowa 50010
TWX 910-520-1166

DAIM



DAIM is a complete disk operating system for the ROCKWELL INTERNATIONAL AIM 65. The DAIM system includes a controller board (with 4K operating system in EPROM) which plugs into the ROCKWELL expansion motherboard, packaged power supply capable of driving two 5 1/4 inch floppy drives and one or two disk drives mounted in a unique, smoked plastic enclosure. DAIM is completely compatible in both disk format and operating system functions with the SYSTEM 65. Commands are provided to load/save source and object files, initialize a disk, list a file, list a disk directory, rename files, delete and recover files and compress a disk to recover unused space. Everything is complete — plug it in and you're ready to go! DAIM provides the ideal way to turn your AIM 65 into a complete 6500 development system. Also available are CSB 20 (EPROM/RAM) and CSB 10 (EPROM programmer) which may be used in conjunction with the DAIM to provide enhanced functional capability. Base price of $850 includes controller board with all software in EPROM, power supply and one disk drive. Now you know why we say —

There is nothing like a

DAIM

Phone 515-232-8187

same mantissa: 11000000 ... How do we tell them apart? By using the exponent byte - the first location, remember?

The final byte contains the sign of the number. Only the first bit counts. If the first bit is zero, the number is positive; if it's one, the number is negative.

Floating point numbers are nice in Basic, but they can be difficult to handle in machine language. You'll probably want to use the built-in subroutines to covert them to and from the more familiar fixed-point numbers. See the Basic manual for this.

You can do the job yourself, if you prefer. Here's the general method. Assuming that your number is not zero (check the first byte) you can re-arrange it along the following lines. If you add one to the exponent, you will have multiplied the number by two; and if you shift the mantissa right, you will have divided it by two. If you do both, the number will have the same value. It will no longer be a normal floating-point number, since the high-order bit of the mantissa will now be zero, but the value will be the same. If you repeat this procedure until the exponent reaches a value of hexadecimal 90, the integer part of your number will be found in the first two bytes of the mantissa. It works: try it out with pencil and paper.

To go the other way (fixed to floating) you must "normalize" the number so that the high-order bit of the mantissa is 1; this takes left-shifting of the mantissa and decrementing the exponent.

Parameters: easier ways

Floating point is messy, and you may want to pass more than one value to or from machine language. There are other ways of doing the job.

The most obvious way is to have Basic POKE the values it wants to give into memory, and have the machine language program pick them up there. In the other direction, Basic can PEEK the results. If your values go above 255, you'll need to use more than one memory location for each value. Use the standard multiply or divide by 256 techniques to separate or recombine the parts.

A better way - but not quite so easy - is to have your machine language program go after the Basic variables in the locations they are stored in memory.

Variables: Ground Rules

Machine language can of course go after any data anywhere in memory. There are a few things you can do, however, to make it much easier to interchange data.

First rule: wherever possible, use Basic integer variables. These are the ones with the percent sign tacked on: J% or D%, for example.

The advantage of integer variables is that they are not stored in floating point notation. Machine language can use them, or change them, in a straightforward manner.

Second rule: arrange for Basic to use these variables at the very beginning of your program. If you want to pass six values (called A%, B%, X%, T1%, T2%, and S%) to machine language, have the first line of your Basic program define them with a line of code like:

```
100 A/ =0 : B/ =0 : X/ =0: S/ =0: T1/ =0 :
T2/ =0
```

This will place the values early in the variable table, where they are easy to access.

Variables: how they are stored

KIM, SYM, and AIM use seven locations for each variable; OSI uses six. The first two locations are the variable name, in ASCII. Fixed-point variables will have the high-order bit set over each byte of the name.

The next two locations of a fixed-point variable contain the binary value - high order first. The remaining two or three locations are not used.

Floating-point variables are also stored in seven (or six for OSI) locations. The format is slightly different from that of the floating-point buffer; a little experimentation should unlock secrets. You will find it generally simpler to use fixed-point format, except on the OSI Basic, which doesn't appear to have this option.

A couple of examples should make fixed-point formats easy to understand. If variable B5% has a value of 22, you'll see it stored as: C2 B5 00 16 00 00 00. C2 is an Ascii letter B with the high bit set; B5 is the Ascii character 5 with the high bit set - together they give the variable name. 00 16 is the value 22 in hexadecimal; and the remaining three locations are not used. If variable C% has a value of 300, you'll see: C3 80 01 2C 00 00 00. Can you figure it out?

Where to find the variables

The variables are normally stored above your Basic program. Since your program could be any size, the variables might start almost anywhere. You'll find out where by looking at your start-of-variables pointer. This is stored - low order first - at the following locations:

KIM - 007A and 007B
SYM - 007D and 007E
AIM - 0075 and 0076
OSI - 007B and 007C

So if your AIM contains the values B3 and 07 in 0075 and 0076, you'll know that your first variable is contained in location 07B3 to 07B9 inclusive. If it's a fixed-point variable, the value will be contained in 07B5 (high-order) and 07B6 (low-order).

You can look through the variable table, jumping seven locations at a time, to find the variable with the name you want. It's easier, as suggested before, to force the variables into the start of the table - that way they will be fast to find.

Here's a handy coding hint. The start-of-variables pointer can be used as an indirect address - after all, it's in zero page. So: if you wanted to get the low-order byte of the x first SYM variable, you could code: LDY #3; LDA ($7D),Y and you've got it. Count carefully; be sure that the variable is defined first in your Basic program; and the job becomes almost routine. You can reach over thirty variables this way, which is plenty for most applications.

If you want to pass values through an array, that's not hard to do. The format is similar to that of variables. Look around and you'll get the idea. One important caution: arrays can move during program execution. Always reference them through the start-of-arrays pointer, which is located directly after the start-of-variables pointer.

Conclusion

Your single-board machine is equipped with very powerful monitor facilities that allow you to look around and see how Basic does things. Use them: you'll find out a lot about how to get Basic and machine language to work harmoniously.

Basic and machine language can be married to give powerful and flexible programs. This brief article won't give you all the marriage counseling you need, but will at least perform the introductions. ©

6502
SOFTWARE

MACHINE LANGUAGE PROGRAMS FOR: *AIM *SYM *KIM *PET *APPLE *OSI & ANY 6502 SYSTEM

■ **FOLIO** A FILE ORIENTED LANGUAGE • A PSEUDO-CODE INTERPRETER • SUPPORTS 27 FUNCTIONS WHICH SIMPLIFY DATA HANDLING • SUPPORTS FILES WITH UP TO 254 RECORDS WITH UP TO 254 FIELDS WITH UP TO 80 CHARACTERS • FIELD SIZE IS VARIABLE FROM ONE RECORD TO ANOTHER • BUILT-IN KEY SORT FUNCTION • FOLIO INTERPRETER OCCUPIES ONLY 1.25K RAM OR ROM • TYPICAL FOLIO PROGRAMS TAKE LESS THAN 1K • DOES NOT REQUIRE BASIC, CP/M*, CBASIC*, 48K OR DISKS • WITH USER'S MANUAL, PROGRAMMER'S MANUAL & COMMENTED SOURCE LISTING ______ $25.00

FOLIO PROGRAMS:

■ **FILE CABINET** A MULTI-PURPOSE DATA STORAGE AND RETRIEVAL PROGRAM ONLY TAKES 320 BYTES • MENU DRIVEN • SELECT RECORDS KEYED BY ANY FIELD • SORT • CHOICE OF REPORT FORMATS • USE FOR MAILING LIST, INVENTORY, EMPLOYEE DATA, ETC •

■ **BUDGET** FOR HOUSEHOLD BUDGET PLANNING • PRINTS A 12 MONTH ACCOUNTING WITH UP TO 26 EXPENSE OR INCOME ITEMS WITH MONTHLY TOTALS AND BALANCE • MENU DRIVEN • EASY TO CHANGE ITEM NAMES OR AMOUNTS • REQUIRES 64 OR 72 CHARACTER LINE TERMINAL

■ **CHECKBOOK** ENTER CHECKS AS YOU WOULD IN YOUR CHECK REGISTER PLUS A CODE NUMBER FOR UP TO 256 CATEGORIES SUCH AS "FOOD","MEDICAL", ETC. • PRINTS THE STATEMENT WITH RUNNING BALANCE FOR SELECTED DATES OR IN NUMERICAL ORDER • PRINTS AND TOTALS CHECKS WITH SELECTED CODE # - FOR SELECTED DATES • PRINTS CODE TOTALS

■ EACH FOLIO APPLICATION PROGRAM COMES WITH THE FOLIO USER'S MANUAL, APPLICATION PROGRAM USER'S MANUAL, AND A HEX DUMP OF FOLIO AND THE PROGRAM ______ $10.00

OTHER 6502 PROGRAMS:

■ **TEA** TINY EDITOR/ASSEMBLER • CHARACTER ORIENTED EDITOR • SINGLE PASS ASSEMBLER • MAY BE USED SEPARATELY TO CONSERVE RAM • 1K EACH! • CLOSELY USES MOS MNEMONICS • COMES WITH USER'S MANUAL AND COMMENTED SOURCE LISTING ______ $20.00

■ **DISASSEMBLER** DISPLAYS SYMBOLIC LABELS AND OPERANDS FROM SYMBOL TABLE GENERATED BY TEA (ABOVE) • OR DISPLAYS ADDRESS VALUE • GREAT FOR DEBUGGING MACHINE CODE • WITH USER'S MANUAL AND COMMENTED SOURCE LISTING ______ $10.00

■ **ROBOT** INTERACTIVE PROGRAMMING LANGUAGE TO CONTROL ROBOT, PLOTTER OR CRT CURSOR • USER DEFINED COMMANDS & COMMAND SUBROUTINES • COMES WITH CRT ROUTINES WITH USER'S MANUAL AND COMMENTED SOURCE LISTING (1.5K) ______ $ 5.00

■ **MUSIC** INTERACTIVE PROGRAMMING LANGUAGE FOR THE CREATION OF PATTERNS OF SOUND "MUSIC" • A COMPOSITION TOOL • NOT A NOTE TABLE COMPILER OR "PIANO ROLL" • COMPLEX HIERARCHIES OF USER DEFINED FUNCTIONS - STRINGS OF MUSICAL EVENTS, WHICH CAN BE CALLED LIKE SUBROUTINES, ALLOW THE PROGRAMMING OF SURPRISINGLY INTRICATE COMPOSITIONS • WITH USER'S MANUAL & COMMENTED SOURCE LISTING ______ $10.00

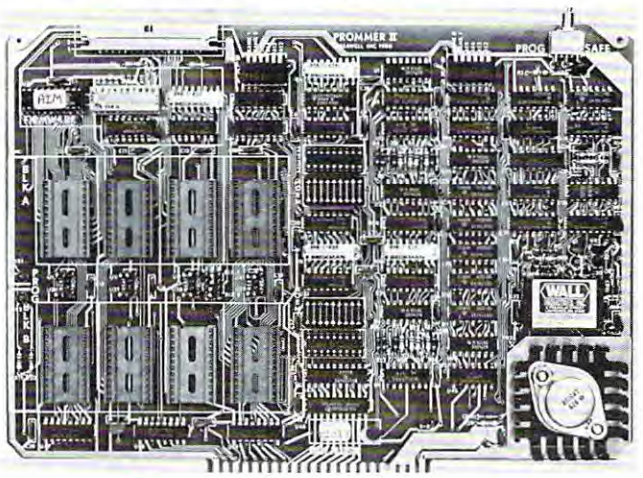
■ ALL PROGRAMS WORK IN A 4K SYSTEM • MANUALS CONTAIN INSTRUCTIONS FOR MODIFICATION USER EXTENSION, RELOCATION, AND INPUT & OUTPUT INTERFACING FOR 6502 PROGRAMMERS

■ CASSETTE TAPE (KIM-1 "HYPERTAPE") ______ $ 3.00

■ ORDER FROM: MICHAEL ALLEN: 6025 KIMBARK: CHICAGO, ILLINOIS 60637

*TM

SEAWELL PROMMER II -- There's Nothing Like It!



- Two independent blocks of 4 EPROM sockets
- KIM, SYM or AIM programming firmware
- Programs 1, 2 or 4K 5-Volt EPROMS: TMS2508, 2516, 2532 and 2758, 2716, 2732
- Read-Only/Deselected/Read-Program for each socket
- Program-protect toggle switch for whole board
- Provision for remoting 4 sockets
- On-board generation of programming voltage.

The Seawell PROMMER II is a general purpose EPROM tool designed for use in a development/production environment. Connects to a KIM, SYM or AIM with a Seawell LITTLE BUFFERED MOTHER motherboard, or to a SEA-1 single-board computer. The PROMMER II is all you need to read, program and execute 1, 2 or 4K 5-Volt EPROMs.

The PROMMER II allows you to put as little as 1K or as much as 32K of EPROM on the bus. You never have to give up address space to empty sockets. Each socket can be enabled to READ only, READ and PROGRAM, or can be DESELECTED entirely. Simply moving a shunt sets one block of 4 sockets to 1, 2, or 4K. The other block can be set independently.

Addresses are selected by piano-type switches on the top edge of the board. The whole board can be program-protected by a toggle switch on the top right corner of the board. A separate one-page ROM containing relocatable firmware for KIM, SYM or AIM is provided which can be set to any page in memory in either of two banks or deselected entirely. A satellite board with four sockets and program-protect switch will be available soon.

PROMMER II EPROM PROGRAMMER -- $299

OTHER SEAWELL PRODUCTS:

SEA-1	SINGLE BOARD DEVELOPMENT SYSTEM	$595
SEA-16	16K RAM BOARD	$280
SEA-FDC8	DOUBLE DENSITY 2-SIDED DISK CONTROLLER	$425
SEA-DEBUG	HARDWARE BREAKPOINTS AND TRIGGERS	$310
SEA-PROMMER II	EPROM PROGRAMMER	$299
SEA-PROTO	COMPLETELY DECODED PROTOTYPING BOARD	$ 99
SEA-CMOS	8K CMOS RAM, 16K EPROM, DAY/DATE CLOCK	$395
SEA-PIOB	4 FULLY-BUFFERED 6522S	$260
SEA-ISDC	8 SERIAL PORTS WITH FOCAL PROCESSOR & DUAL PORT RAM	$595
SEA-LBM	LITTLE BUFFERED MOTHER FOR KIM, SYM, AIM, SEA-1	$199
SEA-MOTHERS	4-SLOT MOTHERBOARD	$70.00
	10-SLOT MOTHERBOARD	$135

ALL PRODUCTS ASSEMBLED.
NO KITS MASTERCHARGE/VISA ACCEPTED

SEAWELL MICROSYSTEMS
P.O. Box 30505,
Seattle, Washington 98103, U.S.A.,
(206) 782-9480