MICROCRUNCH: An Ultra-fast Arithmetic Computing System

Part 2

This article describes software support for the fast mathematics hardware outlined in Part I (39:07). A detailed discussion of machine code routines necessary for communication between the arithmetic processing chip and BASIC is given, along with an overview of a BASIC home-brew compiler.

John E. Hart
Department of Astrogeophysics
University of Colorado
Boulder, Colorado 80309

Part I described a hardware floating point board and demonstrated that truly high speed computing is only possible with a microcomputer if the floating point chip is used in conjunction with a compiler. This is true where the overall program is written in direct machine code. In this case, the source code higher level language statements do not need to be interpreted or pseudo-interpreted (as in Pascal).

This article describes a compiler that is useful for fast arithmetic processing, but does not translate statement types that are rarely if ever used in mathematical problems. The fundamental idea is to use the normal Microsoft BASIC interpreter to do most of the non-mathematical work and to form the overall program structure. When a mathematical loop containing floating point operations needs to be done, a jump is made to a machine language subroutine via USR that executes the equations. It is only this machine language subroutine that is generated by the compiler.

Our system compiles as machine code subroutines, all the time consuming mathematical operations. The source code for these subroutines includes a limited subset of BASIC statements. Then the full BASIC language is used to input variables, set initial conditions, print results of calculations, and perform calculations that, because they are not iterated often, are not time consuming.

There are several problems that need to be discussed.

1. How to communicate between variables used by the mathematical subroutines and variables used in the BASIC main program. Since the C8231 floating point chip uses a nonstandard floating point format (at least it is different from that used by Microsoft) it is necessary for floating point subroutines to have their own variable space. The alternative of converting all BASIC variables to APU (arithmetic processing unit) format upon entry to a subroutine, and then reconverting on exit, is extremely time consuming and wasteful since only a few of the variables used are actually input or output variables. In addition, any time a change is made in BASIC the variable table shifts its position, and BASIC array storage is cumbersome and inefficient.

2. How to get in and out of a large number of compiler-generated machine code mathematical subroutines. Clearly you would like the option of writing several different subroutines and calling them from different points in the main program. Thus some kind of directory management is necessary.

3. What BASIC statements and variable allocations do we allow in the source code for mathematical subroutines?

The Limited BASIC Source Statements

Variable allocation:

Somewhat like a Tiny BASIC, all mathematical subroutine variables are described by a single alphabetic name A-Z. Unlike Tiny BASIC, any variable (except I, J, K, L, M, N that are integer variables for use in FOR loops and indexing) can be either a single number, a vector (e.g. A(I)) or a two dimensional array (e.g. U(I,J)). The vector dimension and the second array dimension must be less than 65 and the first array dimension can be anything consistent with the memory map. Thus there are two types of arithmetic that can be done in a machine language mathematical subroutine: integer and floating point. The integer arithmetic, used mostly for array indexing (e.g. U(I−2+K, J+31)) is done by the 6502 and can only be subtraction or addition.

Statement List:

SUB#, where #=1 to 9 indicating one of 9 possible subroutines.

RETURN, return from subroutine.

GOSUB#, where #=1 to 9. GO to SUB# given.

GOTO#, where #=1 to 9. GO to LABEL# within current subroutine.

FOR I=1TOJ / NEXTI Same as BASIC except no expressions allowed in index setting part of statement.

IF A=0THENGOTO# Same as BASIC, except label referred to 1 to 9, Variable reference (e.g. A) must be simple variable, not vector, etc. Also less than 0 is OK. Only comparisons w.r.t. zero can be made.

LABEL# where #=1 to 9. Jump point for GOTO and IF...THEN.

I = J + K − 25, etc. General integer arithmetic involving only integer variables and numbers less than 256. Only addition and subtraction since these operations are done with the 6502. Mostly used in vector and array indices.

END Denotes termination of a particular subroutine.

In addition to these statements, general mathematical expressions can be written exactly as in BASIC. Example:

```
X = 1.234*U(I-2,J + 1) + B(J)
*SIN(3.141592*Y)
```

This is a marked improvement over such primitive compilers as FLOPTRAN IV and BASEX that do not allow chained calculations or indexing.

It can be seen that this subset of statements is sufficient to implement almost any conceivable iterative and/or conditional calculation. The advantages of the restricted variable set and limited statement types are a shorter and faster compiler. You should note that the compiler must trap all possible source code errors during the compilation, or the machine code subroutines will crash (or give back garbage) and debugging will be extremely difficult. This error trapping is the most difficult part of language translation, and it is made easier by using the restricted language outlined above.

Source statements such as those required to do a long mathematical iteration or calculation, are entered into memory under control of an editor, and then are translated into machine code and placed in the upper end of memory. The compiler and editor are written in BASIC, but being essentially word processors and language translators, execute rapidly. The memory maps for the compilation and run modes are shown in figure 1. The APU variable space depends on the precise allocation of variables, dimensions of arrays, etc. The object code is tied to an initial object starting location OI that is set before compilation.

Variable Format and Exchange

Both Microsoft BASIC and the C8231 represent floating point numbers with four bytes. The first byte contains the exponent, and the next three contain the mantissa, with the most significant bit first. Of course here we are talking about a binary representation where a number is written as

$$\left(\frac{a}{2} + \frac{b}{4} + \frac{c}{8} + \ldots\right) \times 2^E$$

***Figure 1:* MEMORY MAP (typical). Addresses are decimal**

```
Compilation                          Run

_______________0                     _______________0
Microsoft Overhead                   Microsoft Overhead
                                     (loc 0-127 swapped out for math. subroutine)
_______________700                   _______________700
BASIC Compiler                       BASIC:
                                     line 0-6 Overhead
                                     line 6-700 Main Program
                                     line 730-790 Overhead Routines
_______________16000                 _______________4000 (typ)
Compiler Variables                   BASIC variables
_______________18500 (typ)           _______________6000 (typ., depends on vari-
Source Code                          APU Variables          able allocation)
_______________20480 (typ)           _______________19768 (OI-200)
Object Code                          Fixed Routines
                                        and
_______________32768                 Swap Storage
                                     _______________20480 (OI + 512)
                                     Object Code
                                     _______________32768
```

***Figure 2:* Floating Point Formats**

```
                       BASIC                        APU
          ___
Byte 1    7            Exponent Sign                Mantissa Sign
          6            Exponent MSB                 Exponent Sign
          5                                         Exponent MSB
          4
          3
          2
          1
          0            Exponent LSB                 Exponent LSB
          ___
Byte 2    7  a         Mantissa Sign (a = 1         Mantissa a = 1 unless 0
          6  b           inferred unless 0)
          5  c                   '
          4  d                   '
          3  e               MANTISSA   (most significant bit = bit 6 byte 2)
          2  .                   '
          1  .                   '
          0  .
             .
          ___
Byte 3    7  .
          6  .
          5  .
          4  .
          3  .
          2  .
          1  .
          0  .
          ___
Byte 4    7  .
          6  .
          5  .
          4  .
          3  .
          2  u
          1  v
          0  w               Mantissa Least Significant Bit
          ___
```

Here E is typically the exponent and a, b, c, and so forth, are the successive bits of the three byte mantissa, and are either 0 or 1. Figure 2 shows the representations for the two systems. In BASIC a 1 in bit 7 of byte 2 indicates a negative number. For the APU, a negative number is indicated by a 1 in bit 7 of byte 1! Also, bit 7 of byte 2 in APU space is always a 1, except if the number is identically zero. That is, a = 1 unless the number is zero. Note that since the mantissa sign occurs in byte 1 for the APU variable, the exponent range is less by a factor of 2 than for the BASIC variable. Indeed the BASIC exponent range is + 127 to − 128, e.g. the exponent is biased by bit 7, or biased negative 128. However, the APU exponent is only biased negative 64 since the mantissa sign bit occupies bit 7. Thus bit 6 gives the exponent sign.

Machine code routines have been written to convert back and forth between these two formats. Whenever you want to input a variable to APU space, or print out such a number, one of these routines is called by USR from a set of BASIC statements that precede the overall program as shown in figure 1. This is discussed in more detail below. First we list a number of machine code routines that are useful in communicating between BASIC and the APU, and between the compiled code and the APU. These routines must be entered along with each object code, but unlike the object they do not change if either the BASIC source code or main program is altered.

Fixed Routines

Listing 1 is a BASIC program that will load all the fixed routines needed for execution. This program should be run after entering the initial object address OI. OI must be a multiple of 256. In the example discussed below it is 78*256. The decimal entry points and functions of the routines entered by this program are as follows:

Listing 1

```
600  REM  FIXED ROUTINES
601  DATA 32,166,255,216,181,0,157,128,255,202,16,248,162
602  DATA 127,189,0,255,149,0,202,16,248,32,56,255,162,127,181,0,157,0
603  DATA 255,202,16,248,162,127,189,128,255,149,0,202,16,248
604  DATA 173,6,255,41,30,240,3,76,116,162,96
606  FOR J = OI TO OI + 55: READ Z: POKE J,Z: NEXT J
607  DATA 165,5,240,14,56,233,255,16,9,56,255,48,5,169,30,76,153,255
608  FOR J = OI + 170 TO OI + 187: READ Z: POKE J,Z: NEXT J
609  POKE OI + 176,WL: POKE OI + 180,OI / 256 - 1: POKE OI + 187,OI / 256
610  DATA 173,1,251,173,6,251,145,4,200
612  FOR J = 1 TO 9: READ H(J): NEXT J
614  FOR J = OI + 188 TO OI + 218 STEP 9: FOR N = 1 TO 9: POKE J + N - 1,H(N): NEXT N: NEXT J
618  POKE OI + 223,96
620  DATA 177,4,141,6,251,136
622  FOR J = OI + 228 TO OI + 230: POKE J,200: NEXT J
624  FOR J = 1 TO 6: READ H(J): NEXT J
626  FOR J = OI + 231 TO OI + 249 STEP 6: FOR N = 1 TO 6: POKE J + N - 1,H(N): NEXT N: NEXT J
627  POKE OI + 254,96
630  DATA 173,0,251,173,6,251,48,248,41,30,208,1,96,133,6
632  DATA 104,133,7,104,133,8,76,25,255
634  FOR J = OI + 140 TO OI + 163: READ Z: POKE J,Z: NEXT J
640  DATA 8,16,39,31,47
642  FOR J = 1 TO 5: READ Z: POKE OI + Z,OI / 256 + 1: NEXT J
644  DATA 24,163
646  FOR J = 1 TO 2: READ Z: POKE OI + Z,OI / 256: NEXT J
650  DATA 160,3,177,123,72,9,128,160,1,145,1,200,177,123,56,233,128,41
652  DATA 127,136,136,145,1,104,41,128,17,1,145,1,160,5,177,123,136,136
653  DATA 145,1,200,177,123,136,136,145,1,96
654  FOR J = OI + 56 TO OI + 101: READ Z: POKE J,Z: NEXT J
656  DATA 160,3,169,0,145,1,136,48,251,96
658  FOR J = OI + 102 TO OI + 111: READ Z: POKE J,Z: NEXT J
660  DATA 160,1,177,1,48,12,200,169,0,145,123,200,152,73,6,208,246,96
661  DATA 76,210,255
662  FOR J = OI + 117 TO OI + 137: READ Z: POKE J,Z: NEXT J
663  POKE OI + 137,OI / 256 - 1: POKE OI + 2,OI / 256 - 1
665  DATA 160,5,162,6,181,0,153,3,211,232,200,200,224,17,208,244,96
667  FOR J = OI - 120 TO OI - 104: READ Z: POKE J,Z: NEXT J
670  DATA 41,127,200,200,145,123,160,0,177,1,72,41,128,160,3,17,123,145
672  DATA 123,104,41,127,24,10,48,2,56,234,106,136,145,123,177,1,200
673  DATA 200,145,123,136,177,1,200,200,145,123,96
674  FOR J = OI - 46 TO OI - 1: READ Z: POKE J,Z: NEXT J
676  DATA 165,1,141,224,255,165,2,141,225,255,173,226,255,133,1,173,227
678  DATA 255,133,2,32,57,255,173,224,255,133,1,173,225,255,133,2,96
679  GOTO 684
684  FOR J = OI - 256 + 176 TO OI - 256 + 209: READ Z: POKE J,Z: NEXT J
686  DATA 4,9,12,17,22,25,30
688  FOR J = 1 TO 7: READ Z: POKE OI - 256 + 176 + Z,OI / 256: NEXT J
690  DATA 162,127,169,0,141,6,255,96
692  FOR J = OI - 90 TO OI - 83: READ Z: POKE J,Z: NEXT J
694  POKE OI - 84,OI / 256 + 1: STOP
```

(Continued)

OSI AARDVARK OSI
NOW MEANS BUSINESS!

WORD PROCESSING THE EASY WAY— WITH MAXI-PROS

This is a line-oriented word processor designed for the office that doesn't want to send every new girl out for training in how to type a letter.

It has automatic right and left margin justification and lets you vary the width and margins during printing. It has automatic pagination and automatic page numbering. It will print any text single, double or triple spaced and has text centering commands. It will make any number of multiple copies or chain files together to print an entire disk of data at one time.

MAXI-PROS has both global and line edit capability and the polled keyboard versions contain a corrected keyboard routine that make the OSI keyboard decode as a standard typewriter keyboard.

MAXI-PROS also has sophisticated file capabilities. It can access a file for names and addresses, stop for inputs, and print form letters. It has file merging capabilities so that it can store and combine paragraphs and pages in any order.

Best of all, it is in BASIC (OS65D 5 1/4" or 8" disk) so that it can be easily adapted to any printer or printing job and so that it can be sold for a measly price.

MAXI-PROS — $39.95

NEW-NEW—NEW
TINY COMPILER

The easy way to speed in your programs. The tiny compiler lets you write and debug your program in Basic and then automatically compiles a Machine Code version that runs from 50-150 times faster. The tiny compiler generates relocatable, native, transportable machine code that can be run on any 6502 system.

It does have some limitations. It is memory hungry — 8K is the minimum sized system that can run the Compiler. It also handles only a limited subset of Basic — about 20 keywords including FOR, NEXT, IF THEN, GOSUB, GOTO, RETURN, END, STOP, USR(X), PEEK, POKE, -,=,*,/,<,>,<>,Variable names A-Z, and Integer Numbers from 0-64K.

TINY COMPILER is written in Basic. It can be modified and augmented by the user. It comes with a 20 page manual.

TINY COMPILER — $19.95 on tape or disk

THE AARDVARK JOURNAL

FOR OSI USERS — This is a bi-monthly tutorial journal running only articles about OSI systems. Every issue contains programs customized for OSI, tutorials on how to use and modify the system, and reviews of OSI related products. In the last two years we have run articles like these!

1) A tutorial on Machine Code for BASIC programmers.
2) Complete listings of two word processors for BASIC IN ROM machines.
3) Moving the Directory off track 12.
4) Listings for 20 game programs for the OSI.
5) How to write high speed BASIC — and lots more —

Vol. 1 (1980) 6 back issues - $9.00
Vol. 2 (1981) 2 back issues and subscription for 4 additional issues - $9.00.

ACCOUNTS RECEIVABLE — This program will handle up to 420 open accounts. It will age accounts, print invoices (including payment reminders) and give account totals. It can add automatic interest charges and warnings on late accounts, and can automatically provide and calculate volume discounts.

24K and OS65D required, dual disks recommended. Specify system.
Accounts Receivable. $99.95

*** * * SPECIAL DEAL — NO LESS! * * ***

A complete business package for OSI small systems — (C1, C2, C4 or C8). Includes MAXI-PROS, GENERAL LEDGER, INVENTORY, PAYROLL AND ACCOUNTS RECEIVABLE — ALL THE PROGRAMS THE SMALL BUSINESS MAN NEEDS. $299.95

P.S. We're so confident of the quality of these programs that the documentation contains the programmer's home phone number!

SUPERDISK II

This disk contains a new BEXEC* that boots up with a numbered directory and which allows creation, deletion and renaming of files without calling other programs. It also contains a slight modification to BASIC to allow 14 character file names.

The disk contains a disk manager that contains a disk packer, a hex/dec calculator and several other utilities.

It also has a full screen editor (in machine code on C2P/C4)) that makes corrections a snap. We'll also toss in renumbering and program search programs — and sell the whole thing for — SUPERDISK II $29.95 (5 1/4") $34.95 (8").

AND FUN, TOO!

BOOKKEEPING THE EASY WAY — WITH BUSINESS I

Our business package 1 is a set of programs designed for the small businessman who does not have and does not need a full time accountant on his payroll.

This package is built around a **GENERAL LEDGER** program which records all transactions and which provides monthly, quarterly, annual, and year-to-date PROFIT AND LOSS statements. GENERAL LEDGER also provides for cash account balancing, provides a BALANCE SHEET and has modules for DEPRECIATION and LOAN ACCOUNT computation.
GENERAL LEDGER (and MODULES) $129.95.

PAYROLL is designed to interface with the GENERAL LEDGER. It will handle annual records on 30 employees with as many as 6 deductions per employee.
PAYROLL - $49.95.

INVENTORY is also designed to interface with the general ledger. This one will provide instant information on suppliers, initial cost and current value of your inventory. It also keeps track of the order points and date of last shipment.
INVENTORY - $59.95.

GAMES FOR ALL SYSTEMS

GALAXIAN - 4K - One of the fastest and finest arcade games ever written for the OSI, this one features rows of hard-hitting evasive dogfighting aliens thirsty for your blood. For those who loved (and tired of) Alien Invaders. Specify system — A bargain at $9.95

NEW — NEW — NEW

LABYRINTH - 8K - This has a display background similar to MINOS as the action takes place in a realistic maze seen from ground level. This is, however, a real time monster hunt as you track down and shoot mobile monsters on foot. Checking out and testing this one was the most fun I've had in years! — $13.95.

NIGHT RIDER - You've seen similar games in the arcades. You see a winding twisting road ahead as you try to make time and stay on the road. NIGHT RIDER uses machine code to generate excellent high speed graphics - by the same author as MINOS.
NIGHT RIDER — $12.95 cassette only

THIEF - Another machine code goody for the C1P cassette only. You must use mobile cannon to protect the valuable jewels in the middle of the screen from increasingly nasty and trigger happy thiefs. Fast action and fun for one or two players. THIEF $13.95 on C1 cassette only!

SUPPORT ROMS FOR BASIC IN ROM MACHINES — C1S/C2S. This ROM adds line edit functions, software selectable scroll windows, bell support, choice of OSI or standard keyboard routines, two callable screen clears, and software support for 32-64 characters per line video. Has one character command to switch model 2 C1P from 24 to 48 character line. When installed in C2 or C4 (C2S) requires installation of additional chip. C1P requires only a jumper change. — $39.95
C1E/C2E similar to above but with extended machine code monitor. — $59.95

Please specify system on all orders

This is only a partial listing of what we have to offer. We now offer over 100 programs, data sheets, ROMS, and boards for OSI systems. Our $1.00 catalog lists it all and contains free program listings and programming hints to boot.

VISA

OSI

AARDVARK TECHNICAL SERVICES, LTD.
2352 S. Commerce, Walled Lake, MI 48088
(313) 669-3110

master charge

OSI

OI-80	Protect zero page address 1 and 2 for APU BASIC conversions, and jump to proper conversion routine.
OI-46	Convert APU variable whose start address is set in location 1 and 2 and place result in BASIC variable pointed to by location 123-124 (the BASIC variable X since this is the first variable called by the main program as given in LIST 2 below).
OI	Entry to object code. Swap lower half of page zero to upper memory, jump to routine called from main program, swap back page zero, check for address range error, and return (warm start if error set).
OI+56	Convert BASIC variable pointed to by 123-124 to APU variable and set in four locations starting with that pointed to by 1-2.
OI+117	APU to BASIC conversion entry. Check if APU =0, if so set X=0, otherwise jump to OI-46.
OI+140	Check APU for error and busy status. If there has been an error (see part I), pull program counter off stack and exit.
OI+170	Read APU floating point number on top of APU stack to memory starting with location pointed to by 4-5.
OI+228	Write memory floating point number starting at location pointed to by 4-5 to top of APU stack.

BASIC Fixed Routines

When a machine code mathematical subroutine is run a few BASIC statements must be included in the main program. These are given in list 2. The first line makes sure X is at the head of the variable table by setting it equal to zero. It also sets OI. Subroutines 730 and 735 set the variable address bases for the variable A,B,C,D,E,F,G,H,T,X,Y,Z. That is, NF contains the relative address on page zero (after swapping) for these variables. For example, A starts at location 20, B at 24, etc., X at 56, Y at 60, and Z at 64. These subroutines are called before the main program in lines 8-700.

The main program written out in list 2 is used to run the mathematical test loop described below. Line 8 identifies the APU variable X, sets BASIC X=1, and calls subroutine 770 which executes a USR jump to the fixed routine that converts between these variables. Similarly, line 9 causes APU variable A to be set equal to the constant 1.00013. Line 10 identifies a call to the subroutine J whose starting address is set in line 2 (the first subroutine always starts at location OI+512), then executes this jump. Finally, line 11 identifies a variable X that is converted, and then printed. In the conversion calls, first set Z$ equal to the desired variable name, then CALL 770 to go from BASIC to APU, or 780 for the inverse.

The fixed routines outlined above, and these BASIC overhead instructions, are sufficient to manage a large number of mathematical subroutines and APU variables. If there is a warm start after a mathematical subroutine call, a GOSUB750 will print out the error code and an object address of a place near where the error occurred.

An Example

Consider the multiplication test program discussed in part I. This called for consecutive multiplication of X by a specified constant A for 40,000 times. One program to do this would set A and X, and call the following mathematical subroutine.

```
SUB1
FORI=1TO200
FOR J=1TO200
X=X*A
NEXTJ
NEXTI
RETURN
END
```

Note that two nested FOR loops are needed to get 40,000 because integer variables are limited to a range of 0 to 255 each.

List 3 gives detailed description of the object code generated by the compiler when the above statements were entered as a source code. Note OI=19968 for this example.

By inspecting this program you can see that the 6502 is used for loop control. The variable table is the same as was set in statement 733-734 of list 2. A is at loc 20, X at 56. Note that some 6502 statements are executed concurrently while the C8231 is multiplying (20541-20549). Writing short mathematical expressions like X=X*A does not allow much co-processing because you are primarily reading and

Listing 2

```
1 X = 0: DIM S(20):OI = 78 * 256
2 S(1) = 20480
5  GOSUB 730: GOSUB 735
6  REM  END HEADER. MAIN PROGRAM, LINES 8-700.
8 Z$ = "X":X = 1: GOSUB 770: REM  BEGIN MAIN PROGRAM, SET SUBR. VARIABLE X=1.
9 Z$ = "A":X = 1.00013: GOSUB 770: REM   SET MATH SUBROUTINE CONSTANT A=1.00013
10 J = 1: PRINT "START": GOSUB 760: REM  ENTER MATH SUBROUTINE
11 Z$ = "X": GOSUB 780: PRINT X: REM   PRINT FINAL VALUE OF X AFTER 40,000 MULTS
20  STOP
730  REM   VARIABLE ADDRESS BASES--SINGLE VARIABLES ONLY
731  DIM NF(26)
733  FOR J = 1 TO 8:NF(J) = 16 + 4 * J: NEXT J:NF(20 = 52:NF(24) = 56:NF(25) = 60
734 NF(26) = 64: RETURN
735  REM  SET CONSTANTS FOR OVERHEAD ROUTINES
738 S(10) = OI + 226:S(11) = OI + 227:S(15) = 256
739 S(14) = OI / 256 - 1:S(16) = OI + 23:S(17) = OI + 24:S(18) = OI - 59: RETURN
750  REM  ERROR CHECK
751  PRINT "ERROR CODE="; PEEK (OI + 262) AND 30
752  PRINT "ADDRESS="; PEEK (OI + 263) +  PEEK (OI + 264) * 256: PRINT : RETURN
759  REM  760 IS SUB CALL ENTRY J=SUB#
760  IF J > 9 OR J < 1 OR S(J) = 0 THEN  PRINT "ILL SUB CALL TO #";J: STOP
761 X = S(J):XS =  INT (X / 256): POKE S(17),XS: POKE S(16),X - 256 * XS
763  POKE 11,0: POKE 12,OI / 256:X =  USR (0): RETURN
770  REM  BASIC TO APU CONV, Z$=CHAR, J=INDEX, I=PAGE INDEX
771 XS =  ASC (Z$) - 64: IF XS <  > 10 THEN  IF NF(XS) <  > 0 THEN I = 1:J = 1
772  POKE 11,176: POKE 12,S(14): POKE S(10),NF(XS)
773  POKE S(11),79: IF X = 0 THEN  POKE S(18),102
774  IF X <  > 0 THEN  POKE S(18),56
775 XS =  USR (0): RETURN
780  REM  APU TO BASIC
781 XS =  ASC (Z$) - 64: IF XS <  > 10 THEN  IF NF(XS) <  > 0 THEN I = 1:J = 1
782  POKE 11,176: POKE 12,S(14): POKE S(10),NF(XS)
783  POKE S(11),79: POKE S(18),117
784 XS =  USR (0): RETURN
```

***List 3:* A sample object code (all addresses decimal).**

```
20480  162,1       LDX-IMM   1
20482  202         DEX
20483  134,10      STX-Z     10     initialize integer I (I at loc 10
                                    page zero)
20485  166,10      LDX-Z     10     load I
20487  224,200     CPX-IM    200    I equal to 200?
20489  208,3       BNE       3
20491  76,81,80    JMP       20561  If true jump out of For loop.
20494  232         INX              If I less than 200 increment.
20495  134,10      STX-Z     10     restore I

20497  162,1       LDX-IMM   1
20499  202         DEX
20500  134,11      STX-Z     11     initialize integer J (J at loc 11
                                    page zero)
20502  166,11      LDX-Z     11     load J
20504  224,200     CPX-IM    200    J equal to 200?
20506  208,3       BNE       3
20508  76,78,80    JMP       20558  If true jump out to next I
20511  232         INX              If J less than 200 increment J
20512  134,11      STX-Z     11     restore J


20514  169,56      LDA-IM    56     load address base for variable X
                                    (lo)
20516  133,4       STA-Z     4      put into zero page loc 4 (variable
                                    pointer)
20518  160,0       LDY-IM    0
20520  132,5       STY-Z     5      put address base (hi) into loc 5
20522  32,228,78   JSR       20196  goto fixed routine to write X to
                                    top of APU stack
20525  169,20      LDA-IMM   20     load address base for variable A
                                    (loc 20, page 0)
20527  133,4       STY-Z     4      set address pointer
20529  160,0       LDY-IM    0
20531  132,5       STY-Z     5
20533  32,228,78   JSR       20196  write variable A to APU stack (to
                                    OI+228)


20536  169,18      LDA-IM    18     load op code for multiply
20538  141,7,251   STA-AB    64263  command APU to multiply top of
                                    stack by next on stack, result to
                                    top of stack
20541  169,56      LDA-IM    56     set address base for variable X
                                    (loc 56 page 0)
20543  133,4       STA-Z     4
20545  160,0       LDA-IM    0
20547  132,5       STY-Z     5
20549  32,140,78   JSR       20108  APU busy-error check (to
                                    OI+140)
20552  32,170,78   JSR       20138  Read APU to memory (to
                                    OI+170)
20555  76,22,80    JMP       20502  J loop return
20558  76,5,80     JMP       20485  I loop return
20561  96          RTS              return from subroutine 1.
```

writing from the APU. However, in longer calculations involving arrays and complicated indexing, time saved by co-processing can amount to a factor of 2 or more.

The above listings, along with this example, should give the reader enough information to write machine code subroutines by hand. The 6502 just implements, writes, and reads to and from the APU, sends it commands and checks its status. Standard 6502 operations can be used for loop control, jumps between subroutines, etc. It should be possible, without undo effort, to write out such object codes for fairly straightforward calculations. If you want to try this particular program the DATA list in listing 4 should be helpful.

```
DATA 162,1,202,134,10,166,10,224,200,208
DATA 3,76,81,80,232,134,10,162,1,202
DATA 134,11,166,11,224,200,208,3,76,78
DATA 80,232,134,11,169,56,133,4,160,0
DATA  132,5,32,228,78,169,20,133,4,160
DATA 0,132,5,32,228,78,169,18,141,7
DATA 251,169,56,133,4,160,0,132,5,32
DATA 140,78,32,170,78,76,22,80,76,5
DATA 80,96
```

Of course, the ultimate situation is to have the compiler write out the object code as illustrated above. Clearly it takes each BASIC source statement and branches out to routines that parse through the line according to the fundamental operation (e.g. FOR, NEXT, a mathematical expression, etc.). The most complicated aspects of a compiler involve rewriting general mathematical expressions into a stack-processing type form suitable for the C8231, and in the process trapping any errors in the source code. The compiler is much too long to list here (16K of BASIC statements), or to describe in detail. However, I hope these two articles have illustrated how fast mathematical processing can be carried out on a simple micro at minimal cost. Enough material has been presented to write and execute simple mathematical subroutines. For further information (a complete manual and cassette tape) on the compiler please write the author.

Two years ago John Hart became interested in using a microcomputer to control laboratory experiments, and to do theoretical calculations involved with his research in meteorology and physical oceanography. The system described above has been used to solve a variety of problems concerned with flow over or around mountains and simple climate models.