VOLUME 2 ISSUE 4 $1.75 SEPTEMBER 1986

"Serving 99'ers Since 1984"

THE SMART PROGRAMMER

This issue marks the second anniversary of Bytemaster newsletters. Well, I'm ready for another two years and, judging by the number of renewals received, you're ready, too. In fact, the mailbag has been keeping me much busier than I anticipated, but that is a good sign. Thanks for the support! While on the topic of mail, when you write, please include the specifics of information you'd like for us to cover. Though it is not possible to send individual replies, I do read all of the mail and almost any topic that might interest some of our readers is considered for publication.

Artist to XB!

by Richard M. Mitchell

Do you find it excessively difficult to do layouts of Extended BASIC graphics? Worry no more! With the program listed below, you can convert a TI-Artist Instance file (Instances are supported in Version 2.Ø and subsequent versions of TI-Artist) into an Extended BASIC screen!

There are many advantages to converting from Artist's Bit-mapped mode to Extended BASIC's graphics mode. Obviously, from the Extended BASIC environment, beginners will find it difficult to implement a bit-mapped screen and, even if one knows how to invoke bit-mapped mode and load a screen, bit-mapped screens require both more file space and more VDP area. Additionally, some XB statements and sub-programs are not usable with a bit-mapped screen (HCHAR, for example).

Converting bit-mapped graphics to graphics mode is not without a down-side. First, bit-mapped screens allow, with only minor exceptions on the 99/4A, turning on the color of any pixel, while XB requires colors being assigned in groups of character codes called sets. So, to convert to graphics mode, we've dropped the color and work from a "black-and-white" (actually, any 2 colors) environment. Additionally, bit-mapped graphics can have as many different patterns as the video permits (768 on the 99/4A), while Extended BASIC's graphics mode permits only 112 different patterns (character codes 32 to 143), so we cannot re-create all of a bit-mapped screen that has a lot of scattered pixels turned on, for we would, as the LOGO turtle says, run "out of ink". So, "Artist to XB" leaves character 32 as a blank, then re-defines the patterns from 143 downward until either the entire Instance is drawn or character 33 is re-defined.

I first planned to undertake this project long ago, when I published "TIW-DUMP" in *Super 99 Monthly*. I received dozens of suggestions on how to do a layout of an XB screen for building a letterhead to be printed from the Formatter of TI-Writer, but all of the suggested methods seemed to me to be too cumbersome (counting

pixels is just not the way I like to spend my evenings). Well, now users can create their letterheads in a free-style format from any of the art programs supported by TI-Artist, then use TI-Artist to create an Instance file, then use "Artist to XB" (below) to convert to an XB screen and finally convert the XB screen to TI-Writer Formatter format with "TIW-DUMP"!

Because many of you who will use "Artist to XB" are not familiar with Assembly, I have converted the object code into XB CALL LOAD's. Because the source code is rather lengthy, it will not be offered at this time. Of the 3 A/L routines, one is very simple ("PDCLR" sets the pattern descriptors of character codes 32 to 143 to all "Ø"'s), one would be of only limited benefit outside of "Artist to XB" ("ARTHEX" converts an Instance record into pattern descriptor format, an ASCII hexadecimal string) and I'll likely publish the source to the third routine ("PDSRCH") in a future article.

While the XB manual states that the inverse of CALL CHAR is CALL CHARPAT, the complement to CALL CHAR is not resident in XB. PDSRCH (Pattern Descriptor SeaRCH) is a complement of CHAR, accepting a character pattern and returning the character code of the first occurrence of the pattern. PDSRCH does not support three of the characteristics of CALL CHAR's parameters; PDSRCH does not support arrays, does not support multiple definitions and obviously cannot return a character code for an unused pattern. Support of arrays would likely increase the size of the Assembly code substantially. Having PDSRCH support 4 pattern descriptions would also increase the amount of code required and might simply be confusing, as the search that would be a true complement of CALL CHAR would be for 4 consecutive patterns. In the case of an unused pattern, PDSRCH returns a zero in the character code argument. PDSRCH does mimic CALL CHAR's support of pattern descriptions of 16 characters or less. For instance, CALL LINK("PDSRCH","Ø",A) implies trailing "Ø"'s, with the pattern being "ØØØØØØØØØØØØØØØØ", 16 "Ø"'s.

Screens created with "Artist to XB" can be used by other programs by adding a line of XB code such as RUN "DSK1.MYPROGRAM" at line 2ØØ. You could use last month's "XB MIRROR" to save your screen (SIT and PDT) for access by other XB programs. Also, a screen dump could be obtained by running Danny Michael's "SCREEN DUMP". Or, you could RUN "DSK1.TIW-DUMP" for the results described in a preceding paragraph.

The time from the point at which the filename is input until an entire screen is built with "Artist to XB" runs a little under 4 minutes on my system, using a floppy disk drive. RAM disk or hard disk access would likely be much quicker. Though I didn't write a version without Assembly, I suspicion such a program would run for 3Ø minutes or more!

While there were many ways in which I could have approached the conversion process, the method I've employed was chosen to make the "magic" of Assembly a little more understandable, so that programmers with intermediate-level skills will likely be able to modify the XB portion of the program for their own applications.

```
> 100 CALL OBJECT :: DISPLAY A
  T(1,1)ERASE ALL:"ARTIST TO X
  B":"THE SMART PROGRAMMER" ::
   DISPLAY AT(12,1):"INSTANCE
  FILENAME:":"DSK1.X_I"
> 110 ACCEPT AT(13,1)BEEP VALI
  DATE(UALPHA,DIGIT,"*._")SIZE
  (-15):F$ :: OPEN #1:F$,INPUT
> 120 CALL CLEAR :: CALL LINK(
  "PDCLR")
> 130 LINPUT #1:A$ :: A=POS(A$
  ,",",1):: B=LEN(A$):: W=VAL(
  SEG$(A$,1,A-1)):: H=VAL(SEG$
  (A$,A+1,B-A)):: USE=143
> 140 FOR R=1 TO H :: FOR C=1
  TO W
> 150 IF FLAG=1 THEN 190
> 160 LINPUT #1:A$ :: CALL LIN
  K("ARTHEX",A$,B$):: CALL LIN
  K("PDSRCH",B$,A)
> 170 IF A=0 THEN CALL CHAR(US
  E,B$):: CALL HCHAR(R,C,USE):
  : USE=USE-1 :: IF USE=32 THE
  N FLAG=1
> 180 IF A<>0 THEN CALL HCHAR(
  R,C,A)
```

```
> 190 NEXT C :: NEXT R :: CLOS
  E #1
> 200 CALL KEY(5,K,S):: IF S<1
   THEN 200
> 210 END
> 30000 SUB OBJECT
> 30010 CALL INIT
> 30020 CALL LOAD(16360,80,68,
  67,76,82,32,39,86)
> 30030 CALL LOAD(16368,65,82,
  84,72,69,88,38,160)
> 30040 CALL LOAD(16376,80,68,
  83,82,67,72,37,100)
> 30050 CALL LOAD(8194,39,112,
  63,232)
> 30060 CALL LOAD(9460,0,0,0,2
  ,37,45,0,0,0,0,37,45,0,16,0,
  48,0,0,0,16,0,32)
> 30070 CALL LOAD(9482,37,184,
  0,0,9,0,0,5,5,5,2,224,131,22
  4,4,96,0,106,0,4,1,5)
> 30080 CALL LOAD(9504,0,10,0,
  100,31,0,0,0,0,0,0,0,0,48,44
  ,48,44,48,44,48,32,44)
> 30090 CALL LOAD(9526,48,32,3
  2,32,32,48,32,49,57,51,44,49
  ,57,51,0,32,31,16,48,48,48,4
  8)
> 30100 CALL LOAD(9548,48,48,4
  8,48,48,48,48,48,48,48,48,48
  ,0,0,0,0,0,0,0,0,16,0)
> 30110 CALL LOAD(9570,0,1,2,2
  24,36,244,4,192,2,1,0,1,2,2,
  37,71,4,32,32,20,2,3)
> 30120 CALL LOAD(9592,37,72,2
  ,5,37,37,210,96,37,71,9,137,
  2,137,0,0,19,33,34,96,37,98)
> 30130 CALL LOAD(9614,22,5,2,
  6,48,0,218,70,37,72,5,137,2,
  6,0,2,193,51,4,200,193,196)
> 30140 CALL LOAD(9636,9,132,1
  0,135,9,135,6,160,38,44,10,1
  96,162,4,193,7,6,160,38,44,1
  0,132)
> 30150 CALL LOAD(9658,162,4,2
  21,72,2,134,0,16,19,10,130,7
  0,19,3,5,198,16,233,4,198,5,
  198)
> 30160 CALL LOAD(9680,205,64,
  2,134,0,16,22,251,2,10,0,32,
  2,0,4,0,2,1,37,88,2,2)
> 30170 CALL LOAD(9702,0,8,4,3
  2,32,44,2,2,37,37,2,3,0,8,15
  6,177,22,3,6,3,22,252)
> 30180 CALL LOAD(9724,16,9,2,
  32,0,8,2,138,0,143,19,2,5,13
  8,16,234,2,10,0,0,200,10)
> 30190 CALL LOAD(9746,131,74,
  4,32,38,86,4,192,2,1,0,2,4,3
  2,32,8,216,32,37,96,37,71)
> 30200 CALL LOAD(9768,4,96,37
  ,20,2,132,0,58,26,5,2,132,0,
  65,26,11,2,36,255,249,2,36)
> 30210 CALL LOAD(9790,255,208
  ,2,132,0,15,21,4,2,132,0,0,1
  7,1,4,91,2,0,30,0,4,32)
> 30220 CALL LOAD(9812,32,52,3
  2,56,38,90,2,3,131,74,192,83
  ,2,65,0,255,22,3,4,243,4,211
  )
> 30230 CALL LOAD(9834,16,25,2
  ,129,0,99,21,5,2,33,64,0,204
  ,193,4,243,16,15,2,129,0,199
  )
> 30240 CALL LOAD(9856,21,5,2,
  33,255,156,2,2,65,1,16,4,2,3
  3,255,56,2,2,65,2,204,194)
> 30250 CALL LOAD(9878,6,193,2
  04,193,4,243,4,211,3,128,2,2
  24,36,244,4,192,2,1,0,1,2,2)
> 30260 CALL LOAD(9900,37,36,4
  ,32,32,20,5,130,210,160,37,3
  6,9,138,218,160,37,69,37,37,
  2,5)
> 30270 CALL LOAD(9922,37,72,4
  ,201,4,202,4,196,2,3,37,28,2
  10,178,2,138,44,0,19,10,2,13
  8)
> 30280 CALL LOAD(9944,32,0,19
  ,7,9,138,2,42,255,208,6,202,
  220,202,5,132,16,242,6,3,210
  ,19)
> 30290 CALL LOAD(9966,9,136,6
  ,4,19,15,6,3,209,147,9,134,5
  7,160,37,32,162,7,6,4,19,7)
> 30300 CALL LOAD(9988,6,3,209
  ,147,9,134,57,160,37,34,162,
  7,6,4,193,200,9,72,6,200,10,
  199)
> 30310 CALL LOAD(10010,9,71,2
  10,72,6,160,39,68,210,71,6,1
  60,39,68,2,138,32,0,22,208,2
  16,32)
> 30320 CALL LOAD(10032,37,70,
  37,36,2,1,0,2,2,2,37,71,4,32
  ,32,16,4,96,37,20,2,137)
> 30330 CALL LOAD(10054,10,0,2
  6,2,2,41,7,0,2,41,48,0,221,7
  3,4,91,2,224,36,244,2,0)
> 30340 CALL LOAD(10076,4,0,4,
  193,4,32,32,32,5,128,2,128,7
  ,128,22,250,4,96,37,20,82,84
  )
> 30350 SUBEND
```

You may want to save the sub-program at lines 30000 to 30350 in MERGE format for your own programs.

XB SCREEN EDITOR

by Richard M. Mitchell

Last month, we covered saving and loading Extended BASIC screens quickly and this month we've covered converting bit-mapped mode to graphics mode. There are situations in which other techniques for creating screen images are useful. So, how about a simple screen editor that will allow you to do layouts of text or even graphics directly from Extended BASIC? Here it is:

```
> 100 CALL CLEAR
> 110 I=1
> 120 ACCEPT AT(I,1)SIZE(-28):
  A$
> 130 IF I=24 AND SEG$(A$,28,1
  )="!" THEN 150 ELSE I=I+1
> 140 IF I<25 THEN 120 ELSE I=
  1 :: GOTO 120
> 150 END
```

To use the editor, simply run the program and you'll be able to layout 28 columns of text by 24 rows! You can continue editing indefinitely, wrapping from the 24th row to the first. There are only a couple of stipulations. You must work from top to bottom and to end the program you must enter an "!" at the end of the 24th row. You can advance to the next row by either pressing enter or the up or down arrow. By inserting your own code at line 150, you can add in a screen dump or screen save utility, such as "XB MIRROR". Of course, you'll also want to use a CALL HCHAR at line 150 to eliminate the "!" from the end of row 24.

If you re-define all or some character codes at the beginning of the program, you can also display graphics. Character codes 128 to 143 can be used by pressing control keys. The character codes from 128 to 143 and their corresponding keys are as follows:

character code	keystroke
128	<ctrl ,>
129	<ctrl a>
130	<ctrl b>
131	<ctrl c>
132	<ctrl d>
133	<ctrl e>
134	<ctrl f>
135	<ctrl g>
136	<ctrl h>
137	<ctrl i>
138	<ctrl j>
139	<ctrl k>
140	<ctrl l>
141	<ctrl m>
142	<ctrl n>
143	<ctrl o>

Interrupts

Part 1

by D.C. Warren

An important part of micro-processor (µP) architecture is the interrupt interface. The interrupt interface allows the µP to deal efficiently with external devices which require the attention of the µP. We will first discuss what the interrupt interface consists of in general and then take a look at how the 4A handles interrupts.

Most processors have an interrupt line (sometimes several) which is used to signal the µP that a device needs immediate attention. An example might be an RS232 card which has just received a byte of information for the computer. The computer needs to fetch that byte of information before the next one comes along or the data will be lost. One way to deal with this situation is to have the RS232 device signal an interrupt to the processor. The processor checks the interrupt line at the end of each machine instruction and if active immediately stops what its doing and branches (BLWP) to an Interrupt Routine (IR) which services the device. When the service is complete then a return (RTWP) brings the processor back to the original program from where it branched and things proceed as they were before the interrupt occurred .

There are alternatives to using the µP interrupt to service devices. The program which is running could occasionally check the devices to see if they needed servicing. For example, let's say that we have a disk controller, RS232 port and a low-power sensor as external devices. The program that is running would occasionally check each device to see if it needed servicing. This is called "status polling" and if we checked each device in turn then the program would be using the "round robin" approach of status polling. A variation of status polling might be to check the disk controller several times for each time we check the other devices. This may be necessary because the controller is a faster device and may need attention more often than the other devices.

The drawback to status polling is obvious. Much processor time is wasted just checking for a service condition. Also, there could be a relatively large amount of time go by before a device which needs service actually gets it. This happens when the device needs attention just after the program has done a status poll. The device has to wait for the program to come back and do another poll before it can be serviced.

Using the µP interrupt interface avoids the above problems encountered with status polling. The program which is being executed does not need to know anything about external devices. When an interrupt occurs the processor, as described above, branches to an IR which takes care of the device and then returns to the main program as if nothing had happened.

If you have several devices attached to the computer then the IR must determine which device caused the interrupt and execute its appropriate Device Service Routine (DSR). So, another type of poll must take place called an "interrupt poll" to actually find out who caused the interrupt. This is the type of interrupt structure which is used by the 4A. A device signals an interrupt to the µP which branches to the IR. The IR then does a poll to determine which device caused the interrupt so that proper servicing can take place.

Obviously, the interrupt polling method is more efficient than status polling for servicing external devices. It still, however, has to poll to determine which device caused the interrupt. The advantage is that the main program need not have anything to do with checking for a service condition. Also, it is faster and does not needlessly waste processor time as does status polling.

There is yet another improvement that is even more efficient than interrupt polling. We will be more specific here and refer to the TMS9ØØ processor since this method varies from processor to processor. In addition to the interrupt pin on the processor there are four additional pins which are used to signal which device caused the interrupt. Four pins gives a total of 16 different levels (Ø-15) which may be generated and applied to the µP. Each level may be assigned to a device (the 99ØØ reserves zero for a reset, though). For example, the disk controller, RS232 port and low-power sensor we discussed may have the levels one, two and three respectively. When the disk controller needs servicing then it simply signals an interrupt on the interrupt line and puts a binary code of one (ØØØ1) on the four vector pins. The processor then interrupts out of the main program, reads the vector code and branches immediately to the DSR which services the disk controller. It does not have to poll to find out which device generated the interrupt because that information is made available to it on the vector pins.

This type of interrupt is known as "vectored interrupt". Once the µP has received an interrupt and has read the vector code then it needs to know where the DSR is located in memory so that it may branch to it and service the interrupt. With the 99ØØ this is accomplished by assigning specific locations in memory to be used as DSR vectors. A code of zero (reset) has its workspace and program location vectors starting at address HØØØØ in memory. Level one has its vectors starting at address HØØØ4 in memory and

so on through level fifteen (HØØ3E). For example, if the 99ØØ received a second level interrupt from a device it would immediately grab the workspace and program location vectors from memory location HØØØ8 and branch to that particular DSR. Also, the old workspace, main program counter and status words are stored in the DSR's registers 13, 14 and 15 respectively. This is done so that when the DSR is finished the µP can restore these pointers and continue execution of the main program as if nothing had happened.

Although the vector approach is the most efficient method of servicing devices, it is also more complex than the other approaches we've discussed so far and has some potential problems. One problem occurs when more than one device generates an interrupt at the same time. Only one device can put its code on the vector lines or else the µP will become confused and not service the devices properly. This is avoided by using an IC called a Priority Encoder (PE) interface chip. This IC only allows one code to be passed to the µP at any one time. In addition, it allows only the lowest numbered code to pass if more than one device is generating an interrupt. For example, let's say that device two and three generate interrupts at the same time. The PE chip will present code two to the processor and block code three. When the level two device is finished then level three will be presented to the µP for attention. This allows priority assignment to the servicing of external devices. You may, for instance, want to assign the low-power sensor in the above examples the highest priority (level one) since it represents a critical condition.

Another problem occurs when the main program is executing code which is timing critical like a 2ØmS delay loop for an EPROM programmer. If an interrupt occurs in the middle of this loop then by the time the interrupt is serviced and the µP returns to the main program hundreds of milliseconds may have passed. So, the 2ØmS delay loop turns into several hundred milliseconds which is undesirable for your EPROM programmer. This problem is common to not only the vector interrupt scheme but also the polled interrupt scheme. What is used to prevent such an occurrence is an internal interrupt mask within the µP. On the 99ØØ this mask can have a value of zero to fifteen. If the mask is set at three then only interrupts from devices three, two, one and zero will be recognized. The instruction which sets the mask is the LIMI xx instruction.

The mask is set to zero by the programmer whenever the main program is in a timing critical section so that no interrupts can disturb the flow of that section (note that code zero can never be masked out so reset is always active). When finished with the critical section then the programmer can reset the mask to allow interrupts once again.

Also, when an interrupt occurs the µP automatically sets the mask to one level higher than the device which caused the interrupt. Let's set up a condition of an interrupt and step through what happens to the mask. Assume that the mask is set to level four and a level two interrupt occurs. The µP grabs the appropriate DSR vectors out of memory and stores return information, as discussed above, in the new DSR workspace. Note that the current mask value of four is stored within the status word in the new register 15. The mask is now set by the µP to level one. That means that only level one and zero interrupts can interrupt the currently executing level two DSR. As you can see, the processor can get several DSRs deep before it ultimately returns to the main program. When the DSR is finished then the mask is reset to four as the previous workspace, program counter and status words are restored (a RTWP instruction). By this method the main program can continue execution as if nothing had interrupted it.

This month, we've covered the general manner in which interrupts are used, with only casual mention of the 99/4A environment. Next month, we'll continue this discussion and direct our attention more toward the 99/4A's interrupts.

5th 1- =FORTH

by Mariusz Stanczak

The past few weeks were really packed, so let's get to it. First off, there is the latest FORTH issue (#117,July 1986) of *Dr. Dobb's Journal of Sofware Tools for the Professional Programmer.* Oooff! The title of that magazine has grown over the years into quite a mouthful, but don't let it intimidate you, it's still good stuff. And, it looks like it's going to get better for FORTH afficionados. Mr. Michael Ham, new columnist, with his "Structured Programming: Forth" will most likely convince me to finally subscribe. This month his readers get walked through the development of all sorts of one-dimensional array words. Read it! As a supplement to Mr. Ham's article, I would like to share a couple of integer (easily convertible to double or floating which, as my math professor says, is left as an exercise to the reader), multi-dimensional array words I picked up a long time ago. Unfortunately, even though I've searched all my materials, I wasn't able to come up with the source and, give the credit, which he really deserves, to the author. The words follow.

```
: 1ARRAY  ( n --) \followed by a word
    <BUILDS  DUP , 0 DO  0 ,  LOOP
     DOES>   SWAP 2 * + ;
: 2AIM  ( n n -- addr)
    ROT 1- SWAP DUP @ ROT * ROT + 1+ 2 * + ;
: 2ARRAY  ( n n ---)  \followed by a word
    <BUILDS  OVER OVER , , * 0 DO  0 , LOOP
     DOES>  2AIM ;
: 3AIM  ( n n n -- addr)
    DUP DUP @ >R 2+ @ >R >R ROT ROT R> 2AIM SWAP 1- R> * R> * 2 * + 2+ ;
: 3ARRAY  ( n n n ---)  \followed by a word
    <BUILDS  >R OVER OVER , , R> DUP , * * 0 DO  0 , LOOP
     DOES>  3AIM ;
```

Usage of the nARRAY words is really simple. You put on the stack the dimensions and follow them with the appropriate nARRAY word and the name of your structure. For example, an ninety element one-dimensional array SINES would be created by typing 90 1ARRAY SINES, a three element per side cube array CUBE by typing 3 3 3 3ARRAY CUBE etc. Initially, each element in the newly created data structure will contain zero as it is initialized by the nARRAY word. Subscripting into the created data structures follows the model accepted in subscripting matrices in mathematics, so first x then y or, in other words, first row and then column, etc. It follows that to store a number into the cell at second row, third column, first plane of CUBE you would type n 2 3 1 CUBE !. To get the hang of it, build some structures and DUMP them, store some values into them and DUMP the structures again. The first thing you are likely to notice is that the first element(s) contain the dimensions of the structure, which will come in handy in performing matrix arithmetic on the data structures.

Two other pieces of FORTH related info came through the mail. In *Forth Dimensions* (May/June 1986), Mr. Gene Thomas published a set of words that ease the installation of ISR's on the 99/4A and supplied an example of an ISR routine in the form of a current number base display. The installation words follow.

```
: 0ISR  ( --- )   \ISR off
    0 83C4 ! ;
```

```
: 1ISR  ( --- )   \ISR on
    INTLNK @  83C4 ! ;
: INSTALL_ISR  ( --- )   \followed by
                 \word to be installed
    ØISR [COMPILE] ' BL WORD CFA ISR ! ;
```

The word INSTALL_ISR as presented includes an error. Let's see what happens when INSTALL_ISR executes. TICK grabs the next word from the input stream and leaves its pfa on PS, then BL WORD sequence also looks up into the input stream in an attempt to find a string of characters delimited by a blank. It either finds it (Oh, oh!) and, it will copy the string to HERE advancing the pointer into the input stream IN past the found token preventing it from being later properly treated by INTERPRET or, if the stream is exhausted, it will find a null (>ØØ which is the normal and unconditional delimiter standard for input streams) and the execution of BL WORD will end without causing any harm. Next, CFA will convert the pfa on PS into cfa which gets stored into the variable ISR, the intended result. The corrected definition of INSTALL_ISR is:

```
: INSTALL_ISR
    ØISR  [COMPILE] '  CFA  ISR ! ;
```

Now, some very good news for those who are more than casually interested in FORTH. For none, I think, should the name Charles Moore be unfamiliar. Moore's newest development in FORTH is the NC4ØØØ chip. As you may know, there is the very expensive 8Mhz Beta Board, then the 4Mhz Delta Board which, although a lot less expensive, at $9ØØ.ØØ it is still way out of reach for most. Do not despair! To the rescue comes Computer Cowboys (41Ø Star Hill Road, Woodside, CA 946Ø2, tel. (415)851-4362) with what appears as the Delta Board in a kit form (Gamma Board? A never marketed product because of its unprofitable pricing that was still too high for the mass market). For $4ØØ.ØØ you get the board and all, but memory, chips (of which you have to buy 6) as well as cmFORTH listing, Application Notes and Brodie on NC4ØØØ book. Put it all together (11 socketed chips, 3 capacitors, 2 resistors), connect an RS-232 cable (from a terminal or a computer) and you have a 4 MIPS (million instructions per second) computer with 16-bit, parallel, video, floppy and printer interfaces. But, if you are like me and still can't come up with enough money to say "YES!" to this great offer, another alternative is to buy Mr. C. H. Ting's Footsteps in an Empty Valley -- NC4ØØØ Single Chip Forth Engine, which includes, in addition to the description of the architecture, instruction set and cmFORTH operating system's source code, a schematic of a single board computer. All for a head spinning price of $25.ØØ. The book is available from Offete Enterprises, Inc., 13Ø6 S. B Street, San Mateo, CA 944Ø2. Well, now we are finally in the price range my student packet can accommodate, even after the 1Ø% for handling and 6.5% California tax is added. The price of the 4Mhz microprocessor is still about $25Ø.ØØ but it is expected to drop to about $5Ø.ØØ within a year or two.

That's all about near future. Let's go back to present. It was a real joy to recieve the recent installments of this newsletter and, if you enjoyed it as much as I did then hacking assembler is on the rise. Although I'm no old hand in assembly on the 99/4A, my first job in computers included writing in PLAN, an ICL-19ØØ structured assembler. Now, nobody seems to do that (writing in assembler) on mainframes and where I work now, the language of choice (read necessity) is either Fortran or COBOL. What a shame! Anyway, not to stay too far behind the masters of the previous issues, I made an attempt to correct one of the most annoying omissions in TI-FORTH, the lack of an auto-repeating cursor. The system I use only looks like TI-FORTH and, when I tried to transplant its KEY routine into the original version, I was in for a surprise. It didn't work as expected! After three days and about twenty less or more satisfying variations, working and not, it finally came to me. I'm not going

to fit the thing into the space of the original routine. Somebody maybe, but not me. The version that you find below works and that's more than you could say about all the others I dug up from the pile of printouts before I (re)found it. I'd give my head there was somewhere a prettier version, but what the heck. You will have more fun correcting, from the most obvious to the subtle, the inefficiencies of it. Have fun and let me know if you succeed in squeezing its full functionality into the space between >3654 and >36CA (inclusive).

```
BASE->R HEX  HERE  3654 DP !
C028 , 0016 ,  \KEY0   MOV  @CURPO$(U),R0     addr of current cursor position
D0A0 , 3CD6 ,  \       MOVB @CURCHR,R2        last read character from screen
0203 , 1E00 ,  \       LI   R3,>1E00          cursor name
C120 , 3CD4 ,  \       MOV  @KEYCNT,R4        := -1 initial, 0<= on reentry
1509 ,         \       JGT  KEY3              if reentry
1308 ,         \       JEQ  KEY3
0420 , 3A1C ,  \       BLWP @VSBR             read chr at cursor position
D801 , 3CD6 ,  \       MOVB R1,@CURCHR        save it
D081 ,         \       MOVB R1,R2
D043 ,         \KEY1   MOVB R3,R1             flash cursor
0420 , 3A14 ,  \KEY2   BLWP @VSBW
0205 , 0258 ,  \KEY3   LI   R5,600            delay factor
04C6 ,         \       CLR  R6
D806 , 837C ,  \       MOVB R6,@KYSTAT        clear GPL status register
0420 , 3A10 ,  \       BLWP @KSCAN            read keyboard
9806 , 837C ,  \       CB   R6,@KYSTAT        was key pressed
161B ,         \       JNE  KEY8              if yes
0620 , 3CDC ,  \       DEC  @DELAY            is it time to check for same key?
1311 ,         \       JEQ  KEY7
0584 ,         \KEY4   INC  R4
0284 , 0096 ,  \       CI   R4,150            is it time to flash original chr?
1305 ,         \       JEQ  KEY5              if yes
0284 , 012C ,  \       CI   R4,300            is it time to flash cursor chr?
1204 ,         \       JLE  KEY6              if not
04C4 ,         \       CLR  R4
10E7 ,         \       JMP  KEY1
C042 ,         \KEY5   MOV  R2,R1             prep. for flashing the orig. chr
10E6 ,         \       JMP  KEY2              go do it
C804 , 3CD4 ,  \KEY6   MOV  R4,@KEYCNT        preserve flash count
0300 , 0002 ,  \       LIMI 2                 enable interrupts
064D ,         \       DECT IP                reenter KEY
045F ,         \       B    *NEXT
0206 , FF00 ,  \KEY7   LI   R6,>FF00          'no character' code
9806 , 8375 ,  \       CB   R6,@KYCHAR
13EA ,         \       JEQ  KEY4
0205 , 003C ,  \       LI   R5,60             autorepeat factor
C805 , 3CDC ,  \KEY8   MOV  R5,@DELAY         set DELAY
0460 , 3CDE ,  \       B    @KECONT
3CDC DP !
0258 ,         \DELAY  DATA 600               initial delay factor
0720 , 3CD4 ,  \KECONT SETO @KEYCNT
D060 , 3CD6 ,  \       MOVB @CURCHR,R1        restore original character
0420 , 3A14 ,  \       BLWP @VSBW
D020 , 8375 ,  \       MOVB @KYCHAR,R0        return new character in LSB of R0
0980 ,         \       SRL  R0,8
0460 , 3482 ,  \       B    @BKLINK           exit the function
DP !  R->BASE  ;S
```

That's all for this month. For now, may FORTH be with you.

XB/Assembly:
De-bugging De-mystified

by Richard M. Mitchell

Any BASIC programmer can write Assembly instructions! Yes, all that is required is the E/A manual, a set of memory maps and maybe a pad of paper. All one has to do to write the instructions is to look 'em up in the E/A manual! Now, before you plunge into writing the all-time greatest piece of software, there is one more fact about writing Assembly. Once programs exceed about 15 lines long, the likelihood of getting the program to work properly the first time it is run is about 50-50 between a slim chance and no chance at all! Yes, de-bugging is a fact of life for Assembly programmers.

In years gone by, de-bugging was, for many (most?) users, an arduous task, offering the ultimate test of a programmer's patience. There were only a few de-bugging utilities available, so if you didn't understand how to use the de-bugger on the E/A disk, you were out of luck! Today, several utility programs for de-bugging and related tasks are available. While my preference lies with Millers Graphics' MG Explorer program, I suggest that if de-bugging has been your problem in the past, you need to either try another de-bugger or seek help with the one you have used. Keep in mind that nearly every de-bugger available has been used successfully, so use the one with which you are most comfortable. The remainder of this article will deal with use of Explorer, as I likely would not do justice to the strong points of the other programs.

For those of you who are not familiar with Explorer, the program is a CPU emulator and does much more, supplying information on virtually everything an Assembly programmer would want to know about code that is executing and updating the information on each machine instruction. Explorer has windows for memory contents, instruction disassembly, CPU and VDP register contents, status bit values, etc. Also, a number base conversion utility is available with a keystroke. Explorer has to be seen to be believed!

Using Explorer should not be considered the first step in the de-bugging process. As I have stated on many occasions, de-bugging should begin when a program is written. Planning a program and laying it out in a structured fashion reduces problems later.

Once source code is completed, it can be assembled, at which time a LIST file can be produced (use the L and S options when assembling). I usually send the LIST file to disk in an effort to save paper. When I reach the stage that I feel I must have a printout of the LIST file, I then print it. If you don't have either a printer or a second system for referencing information such as LIST files, you'll likely find it very cumbersome to write Assembly! Fortunately, printer prices have declined in the past couple of years and good ones are available for $100 to $300 (very good ones for under $600).

While the LIST file is very helpful, the LIST shows addresses relative to >0000, which is the offset to the load address for relocatable code (this does not apply to AORG code, as the load address for AORG code is specified with the AORG directive). To determine the specific address for a line of code, one must add the LIST file address to the load address. For instance, if an A/L program is the first loaded into XB, the load address is >24F4 (the First Free Address in Low Memory in XB), so whatever is at >0000 in the LIST file will load to >24F4 (subsequent loads of relocatable code load at the new FFALM, at the next word after the preceding file). Well, I'm not about to add the offset to the load address for every byte in the LIST file and jot the addresses on the LIST file, as that is "busy work"! Instead, I calculate only the beginning and ending addresses, then load the program and then use DISkASSEMBLER™ to disassemble memory and print the output. Then I can use both the LIST file's commented code and the DISkASSEMBLER™ output's specific addresses. I have found bugs simply by running a program and

disassembling, locating erroneous register values, etc. that I tracked to my errors!

Armed with the LIST and disassembly printouts, it is fairly easy to run Explorer. Because we are assuming in this article a LINK to XB, the CALL LOAD to load Explorer should immediately precede the CALL LINK that we wish to explore. After Explorer loads and is started, we find that a LINK goes through in excess of 1,ØØØ machine instructions (this number varies and can range up to 4,ØØØ or more!) before beginning execution of the user's code! The LINK executes rather quickly at full speed, but Explorer slows everything down so we can follow things. So, we need to set a breakpoint so we can go quickly through the LINK executions (assuming we aren't interested in system operation, only our own code).

Before discussing breakpoints, let's take a brief look at all of those instructions for a LINK! Exactly what LINK does is beyond the scope of this article, but I think you should stop to consider that while Assembly code, even including the LINK, is much, much faster than equivalent BASIC code, LINK'ing to BASIC or XB obviously cannot begin to compare to the speed of a 1ØØ% Assembly program! Additionally, because XB makes use of nearly all of high-speed Scratchpad RAM (some other environments make lesser use of Scratchpad), we generally cannot use Scratchpad for our own XB A/L code and therefore cannot execute Assembly code as quickly from the XB environment as from a 1ØØ% Assembly environment (yes, there are fancy tricks for using Scratchpad from XB, such as what I did in "XB MIRROR"). But, 1ØØ% Assembly is sometimes difficult to code and often is not very byte-efficient. So, XB is an environment in which a primary objective should be accomplishing a task, with only moderate emphasis on raw speed. Once you have a functioning program, look for possible improvements if you have time. Well, the point of this digression is that Explorer enabled me to grasp the significance of the effects of LINK and to deal with XB from a pragmatic perspective! Often, explorations are very enlightening!

From discussions I've had with many users, it seems that many people are not understanding Explorer's breakpoints. A breakpoint is simply a point at which you want continuous instruction execution to pause, enabling you to examine everything at that point and/or single execute instructions beginning at that point. From XB, we would normally want to set the first breakpoint as a CPU breakpoint at the entry point to the program. Remember, the entry point is the address of the label you DEF'd and LINK'ed. Obtain the entry point address from your disassembly printout. The degree to which you grasp breakpoints seems to be what determines whether you'll call Explorer a "fancy thingamabob" or a fantastic tool!

Upon reaching your routine's entry point, keep track of single executions by comparing your LIST file with Explorer's disassembly window. On each execution, check the values in the workspace registers and any other values you possibly can. If you're a beginner, at least look for the obvious errors. For instance, if you know that a register should contain a nibble value (Ø to >F), but it contains other than a zero in other nibbles (an example would be >25C4), then you should suspicion that you have placed an address rather than the contents of an address into the register, which is a common error! Continue looking for errors through each execution -- as you gain experience, you'll find it increasingly easy to spot errors.

After executing a few instructions, you may come across a BLWP to a resident routine, such as a BLWP @VMBW, or a BLWP to a routine of your own that you've previously de-bugged. Well, it can take quite awhile to single-step through such routines and it is fairly safe to assume that since you're trying to improve your own code, you'll not be interested in examining resident routines. So, get ready for continuous execution by setting a breakpoint! "Where?", you ask! Simple! After the workspace change of the BLWP, R14 contains the address for your breakpoint! There is an exception, though. If the BLWP is followed by DATA, then R14 will be incremented as

the DATA is used within the routine in the new workspace. Because of the DATA exception just noted, I expand Explorer's disassembly window to 3 lines either all of the time or when I encounter a BLWP. If no DATA follows the BLWP, then I know R14 will contain the address for the breakpoint. If DATA does follow, R14 will be incremented in relation to the words of DATA encountered. When DATA does follow the BLWP, you may find it easier to obtain the address for the breakpoint from your LIST and/or disassembly printouts. Continuous execution through the resident routines will make your de-bugging go much faster!

Well, discussions of Explorer and de-bugging could easily ramble on through volumes, so for now I'll simply ask that you write in to let me know what you'd like to see covered more extensively.

Extracting Values from Spreadsheet Strings

by Richard M. Mitchell

In a previous issue, I described using Multiplan™'s MID and VALUE to extract a value from a string. With the formula given in that issue, the number had to fall at a specific position and be a specific length. Not being satisfied with those limitations, I set about trying to extract the first number that occurs within a string, regardless of its position or its length. Unlike BASIC, Multiplan™ has no POS (position) function and no formal loop structure. Ouch! So, after pondering the problem, the only solution I have come up with is to use iteration in a somewhat unique fashion. By using ITERCNT(), we can increment the parameters of MID until ISERROR returns FALSE (no error), incorporating all of that into an IF! If the formulae shown below initially confuse you, I won't be surprised! It took me about 9 hours to figure it out! Here is the spreadsheet:

	1	2	3	4
1				

	5	6	7	8
1		LEN(RC[-5])	IF(ISERROR(VALUE(MID(RC[-6],MIN(RC,ITERCNT()),1))),MIN(ITERCNT(),RC[-1]),IF(ISNA(RC),1,RC))	IF(ISERROR(VALUE(MID(RC[-7],RC[-1]-1,2))),RC[-1],RC[-1]-1)

	9	10	11	12
1	MID(RC[-8],RC[-1],RC[-3]-RC[-1]+1)	IF(ISERROR(VALUE(MID(RC[-1],1,ITERCNT()))),RC,MIN(ITERCNT(),LEN(RC[-1])))	MIN(ITERCNT(),LEN(RC[-2]))	IF(ISERROR(VALUE(MID(RC[-3],1,RC[-2]))),0,VALUE(MID(RC[-3],1,RC[-2])))

After keying the formulae, the iteration function from (O)ptions must be turned on with (Y)es. Your string goes into column 1. The first occurrence of a number (yes, even in scientific notation!) in that string is returned in column 12. If there is no number, a Ø is returned. Columns 2 through 5 are not used, as I set the spreadsheet up to support SYLK files created from Extended BASIC with the "8ØSYLK" program that appeared in *Super 99 Monthly*. While this was primarily just an interesting project, it will allow you to obtain numbers from text files that were created by other users. For numbers you generate, a better solution would be to modify "8ØSYLK" to support numerics instead of strings. I must warn you that the above spreadsheet is slow!

The real significance of the above spreadsheet is that it is an example of using loops, a CASE structure to be more precise, if you are familiar with CASE, which is used in many languages, including FORTH. And you thought Multiplan™ didn't support loops, eh? If you come up with an interesting CASE structure using ITERCNT(), let us hear from you!

Oops!

In the "XB MIRROR" article last month, I really blundered in stating that "XXB" AORG's into High Mem. The program uses the XBALT method to load into the normal Low Mem area, at >24F4. For those who wish to use both programs together, you might try AORG'ing XB MIRROR into High Mem.

In the August News section, I failed to disclose that the photographer who so kindly sent pictures of the German 8Ø-column card was Ken Davies. Many thanks, Ken.

Q&A

The Scratchpad Memory map in the August 1986 issue includes several references to temporary variables, etc. Will you be disclosing more information on those addresses?

Yes, I have several projects in the works that will clarify the use of many of the addresses in Scrathpad. We certainly intend to continue to provide, as space and time permits, all of the details that we are familiar with. Some addresses cannot be adequately explained within the format of a map, so future articles will include code and discussions that will shed much more light on the uses of many of the addresses. "XB MIRROR" has been well-received and between my own research and the suggestions of several experts, the enhancements and expanded capabilities that I'm working on for that program should prove enlightening.

Piracy Crackdown

The Software Publishers Association has announced that it is extending an open offer to all BBS users to join the fight against software piracy. The Association, which represents over 15Ø firms in the software industry, issued the offer of $1ØØ to the first individual who provides all of the following iformation about a pirate bulletin board in the U.S. or Canada:

1) The name of the bulletin board (if any), telephone number and necessary log-on information.

2) The full street address of the board and the full name of the System Operator (Sysop).

3) A disk containing copyrighted materials downloaded from the system.

4) A printout showing all other copyrighted materials available on the Board, with the date on which the printout was made.

Upon receipt of the above information, the SPA may, at its option, either contact the Board and request its immediate shut down or refer the evidence to the local office of the F.B.I. for an investigation into possible

criminal violations of the Federal Copyright Laws. Upon verification of the information provided, the SPA will send a $100 check to the individual who provided the information.

Ken Wasch, Executive Director of the SPA, emphasized that the industry appreciates the valuable service most electronic bulletin boards provide of facilitating the exchange of information among users. Wasch added, "The 'pirate' bulletin boards unfortunately are stealing from both users and the software industry, which depends on software sales to recover the enormous development costs of each software application." The SPA expects to initiate few prosecutions, since it anticipates that most offenders will comply with requests to remove copyrighted software from their boards.

In announcing the offer, the SPA added the following terms and conditions:

* The offer opened on September 1, 1986, and will continue until November 1, 1986, unless otherwise extended or modified.
* The SPA reserves the right to modify terms of the offer to ensure that the reward money is not claimed in a fraudulent manner.
* The determination of whether a claimant has complied with the terms of the offer shall be made exclusively by the Software Publishers Association.
* The identity of persons providing information in response to this offer will be held strictly confidential.

For more information about the offer, please contact the SPA at (202) 452-1600.

NEWS

Reports from Seattle indicate that the recent show there was a big success!

In Seattle, Craig Miller announced that he is engaged in a joint venture with a major U.S. firm (the firm has not previously been involved in 99/4A projects) to produce a device that will provide full hardware and software interface between the 99/4A and an IBM[1]-type microcomputer environment. An announcement with further details will be made in January, 1987. According to Craig Miller, the device is now substantially completed and will likely be available by early 1987.

Tape, Ltd.'s Franz Waggenbach was in Seattle and showed the 80-column card that we mentioned last month and also brought in an EPROM'er. Word has it that an EPROM'er was released for evaluation, so we'll try to have more on it in an upcoming issue.

MYARC, Inc.'s Lou Phillips provided information on the firm's Genéve computer card project to the Seattle show audience. According to Phillips, the Genéve, which utilizes a 9995 processor and a 9938 video chip, is nearly completed. Phillips also stated that MYARC will soon be releasing a new controller that will include hard disk controller capabilities, thereby substantially reducing the cost for 99/4A owners to get into hard disks.

Most of the attention at the show in Seattle apparently focused around hardware. With so many programmers working on the various hardware projects, there's sales to be had by any talented programmers who pursue the market! Go for it!

The fourth annual Chicago TI-99/4A Computer Faire, sponsored by the Chicago-Area TI 99/4A Users Group, will take place November 1, 1986, from 9 am to 6 pm. The show will again be held at the Ironwood Room, Triton College, 2000 North Fifth Avenue, River Grove, Illinois, about 15 minutes from O'Hare Airport. The show has consistently attracted about 2000 99'ers from about 15 states! Many of the folks you've heard about and corresponded with over the years will be there, so don't miss it! Barry Traver and I (Richard Mitchell) will be there and invite you to drop by our booths.

CONSOLE CRU BIT MAP (9901)

All of the Data for the 9901 on the 99/4A is inverted.
On or Set = 0 and Off or Reset = 1

Cru Base Address	Bit No.	Description
>0000	0	0 = Internal 9901 Control 1 = Clock Control
>0002	1	Set by an External Interrupt (Peripheral Device)
>0004	2	Set by TMS 9918A on Vertical Retrace Interrupt
>0006	3	Set by Clock Interrupt for Cassette read/write routines Also used for Keyboard Matrix Row 7

KEYBOARD 8 x 8 MATRIX

	Bit No.		Column 0	1	2	3	4	5	6	7	
>0006	3	Row 7	=	.	,	M	N	/	Joy1	Joy2	Fire
>0008	4	Row 6	SPACE	L	K	J	H	;	Joy1	Joy2	Left
>000A	5	Row 5	ENTER	O	I	U	Y	P	Joy1	Joy2	Right
>000C	6	Row 4		9	8	7	6	0	Joy1	Joy2	Down
>000E	7	Row 3	FCTN	2	3	4	5	1	Joy1	Joy2	Up
>0010	8	Row 2	SHIFT	S	D	F	G	A			
>0012	9	Row 1	CTRL	W	E	R	T	Q			
>0014	10	Row 0		X	C	V	B	Z			

Cru Base Address	Bit No.	Description
>0016	11	Not Used
>0018	12	Reserved - High Level
>001A	13	Not Used
>001C	14	Not Used
>001E	15	Not Used
>0020	16	Reserved
>0022	17	Reserved
>0024	18	Bit 2 of Keyboard Matrix Column select (8x8 matrix)
>0026	19	Bit 1 of Keyboard Matrix Column select
>0028	20	Bit 0 of Keyboard Matrix Column select (MSB) (set up the column to read - R1 = 00xx thru 07xx) (LI R12,>0024 LDCR R1,3) (and read the row bits (3-10) with) (LI R12,>0006 STCR R4,8 INV R4)
>002A	21	Set Alpha Lock
>002C	22	Cassette CS1 motor control On/Off
>002E	23	Cassette CS2 motor control On/Off
>0030	24	Audio Gate enable/disable
>0032	25	Cassette Tape Out
>0034	26	Reserved
>0036	27	Cassette Tape In
>0038	28	*
		* Not Used - causes lock up.
>00FE	128	*

BYTEMASTER ORDER FORM

The Smart Programmer

SP1	$18.00	U.S. AND CANADA FIRST CLASS
SP2	$15.00	U.S. THIRD CLASS (no back issues)
SP3	$20.00	FOREIGN SURFACE (no back issues)
SP4	$32.00	FOREIGN AIRMAIL
SP5A-D	$ 1.75	U.S. JUNE - SEPT. 1986, ea.
SP6A-D	$ 2.75	FOREIGN JUNE - SEPT. 1986, ea.

Super 99 Monthly

SM1	$18.00	Complete set of 18 back issues
SM2A-R	$ 1.00	Back issues - ea. (U.S. Third Class)
SM3A-R	$ 1.50	Back issues - ea. (Canada and U.S. First Class)
SM6A-R	$ 2.50	Back issues - ea. (Foreign Air Mail)
SM4	$12.00	Programs on disk (non-FORTH)
SM5	$15.00	Super 99 Handicapper (req. XB, 32K, Disk, Printer)

ITEM #	QTY	EACH	AMOUNT	New	Renewal
				☐	☐
				☐	☐

NAME_____

ADDRESS_____

CITY_____

STATE_____

ZIP CODE_____

COUNTRY_____

Payments accepted by check or money order in U.S. funds, coded for processing through the U.S. Federal Reserve Banking System. No billings or credit sales. Dealer inquiries invited. Discounts available on quantity orders.

Louisiana residents must add 4% sales tax. Calcasieu 1%. Sulphur 2%.

The Smart Programmer is published monthly by Bytemaster Computer Services, 171 Mustang Street, Sulphur, LA 70663.

Editor Richard M. Mitchell

Staff Craig Miller, Steven J. Szymkiewicz, MD, Charles M. Robertson, Barry A. Traver, Mariusz Stanczak, D.C. Warren

DISkASSEMBLER is a trademark of Millers Graphics
Multiplan is a trademark of Microsoft Corp.
1) IBM is a registered trademark of International Business Machines

Bytemaster Computer Services
171 Mustang Street
Sulphur, LA 70663-6724
U.S.A.

Bulk Rate
U.S. Postage
PAID
Sulphur, LA 70663
Permit No. 141

POSTMASTER: ADDRESS CORRECTION REQUESTED
RUSH -- TIME DATED MATERIAL