Forth-Based Software for Real-Time Control of a Mechanically-Scanned Ultrasonic Imaging System

*E.T. Lynk† and H.E. Johnson, Jr.**

General Electric Company
Corporate Research and Development Center
Schenectady, New York 12345

Abstract

Forth was used as the control software for an ultrasonic imaging system developed at GE/CRD. The system is an "Ultrasonic Macroscope" and is used in industrial "Nondestructive Evaluation" applications. It forms an image by mechanically moving an ultra-sound beam over the surface of the material under test. The beam is tightly focused at a constant depth below the surface to reveal flaw locations within the sub-surface slice. Image acquisition and display functions are performed by two Direct-Memory-Access controllers, using extended memory. Data input, output, and computation all take place concurrently. The system is fully interrupt-driven to free the 11/23 bus for DMA transfers (word transfer rates on the order of tens of KHz).

The application software (which runs on an enhanced fig-FORTH-derived kernel) models the imaging system as a cyclic, sequential finite state machine. A state variable is derived from the image line-counter and governs the operations that the CPU must perform. Through the use of interrupts and flags, the output process (which synchronizes the entire operation) functions as if it were driven by an independent processor.

Routines were written to permit high-level access to extended memory and high-level interrupt handlers. Software debugging was facilitated by a "snapshot" routine which captured the "state" of the system. It provided a post-mortem display/dump of the sequence of events, state of flags, variables, etc., at the time each snapshot was taken.

This application was particularly well-suited to FORTH. Because of the heavy bus usage anticipated at high scan speeds, stand-alone operation is mandatory, regardless of the software environment.

Introduction

The software described in this article controls an instrument called a "Scanning Ultrasonic Macroscope" (SUM) [1]. It is used in the field of Non-Destructive Evaluation to assess the quality of a multitude of engineered materials, over a wide range of industries. The SUM produces magnified hard-copy images of defects such as voids, inclusions, and disbonds, which lie beneath the surface of materials. The images are then used to determine properties (such as size, spatial distribution, etc.) of the sub-surface features which were revealed by the ultrasonic probe. Details on the theory and applications of SUM may be found in the references cited below [1,2,3]. A brief description of our instrument will be given here so that our software may be understood.

†Present address: Dept. of Mechanical Engineering, The City College, CUNY, Convent Ave & 138th St., New York, NY 10031

*Present address: Advanced Computer Applications, Inc., 125 Pheasant Run, Newtown, PA 18940.

Typical Images

Figures 1 through 3 are examples of images made with the SUM described in this article. Figure 1 is an image of the soldered bond between two silicon wafers. The dark spots are voids in the solder layer. Figure 2 is an image of the bond region of a commercial "chip carrier", a package made of bonded ceramic sheets. The dark areas (except for the central square) are places where the bond is faulty. Figure 3 is an image of echoes from a penny, taken through the back side. This image, of little technical importance, is a dramatic demonstration of the excellent gray-scale capability of this system.

Figure 1. Defective Soldered Bond Between Si Wafer and Cu Heat Sink.

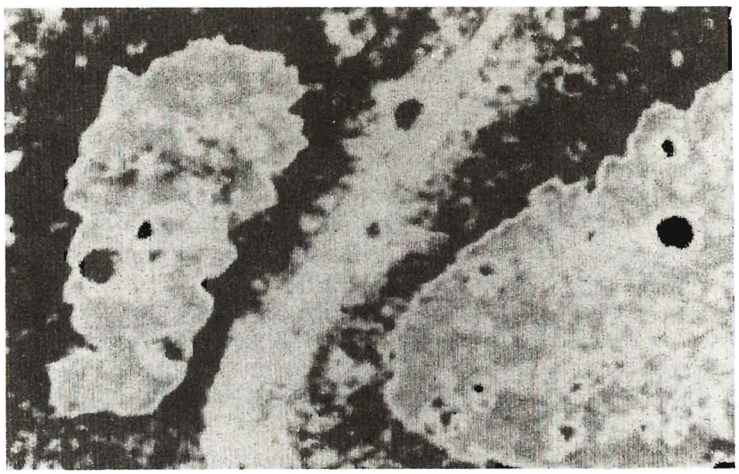


Figure 2. Defect in Commercial "Chip Carrier" Bond.

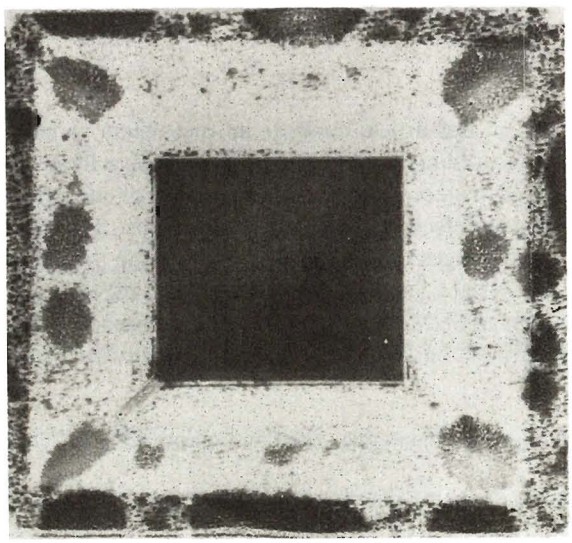


Figure 3. Gray-Scale Rendition of a Penny Taken from Back Side.

Software Environment

The application software was written using a version of Forth derived from PDP-11 fig-FORTH. It has been enhanced with numerous features including multi-tasking, support for several kinds of disks, an excellent screen editor, a string package, utilities, the Forth-79 standard and many others. There are (at least) three versions of "GE/CRD-fig-FORTH", all generated from Forth source code using the Cassady meta-compiler: A stand-alone system (used for the present application), one which runs as a task under RSX-11M (with interfaces to the RSX system), a similar version for the VAX under VMS which runs in compatibility mode, and a version which generates code for the 8085. The RSX and VAX versions allow program development for large projects to take place on timeshared systems. There are utilities for downloading applications to the target computer and vice-versa.

The SUM

Figure 4 is a schematic diagram of the current version of SUM. A photograph of the instrument is shown in Figure 5. The SUM makes an image by mechanically scanning a focused ultrasound beam over the material to be examined. We use broadband transducers with center frequencies typically between two and 100 MHz, yielding minimum spot sizes of about 5 mils. The scanned area on the objects of interest might range in size from one-half to several inches on a side.

The two-axis translation stage (to which the ultrasound transducer is attached) is driven by stepping motors and a dedicated controller. Front-panel dials specify the distance to be travelled along each axis. The LSI-11/23 computer initiates axis moves, and the controller generates an interrupt when the move is finished. The fast axis typically moves at about 4 inches/second. A pulse, derived from the instantaneous position of the stage, fires the transducer. The resulting echo signals are digitized with a gated peak-detector and stored in the 11/23 memory via a DRV11-B DMA controller. The gated echo signals come from a constant depth within the material.

A second DMA controller (Data Translation DT-2771) sends the image data from memory to the plotter, after conversion to an analog signal. The image grid size is made larger than that used for scanning, in order to achieve magnification. The image can optionally be filled with repeated or interpolated lines in order to make a correctly formatted image when the object is spatially under-

*Originally written and placed in the public domain by John James.

sampled. This can be done without loss of resolution: because of the finite spot size of the ultrasound beam, it is permissible to insonify the object on a sparse grid, as long as the spots from adjacent samples overlap.

Figure 4. Block Diagram of Scanning Ultrasound Macroscope.

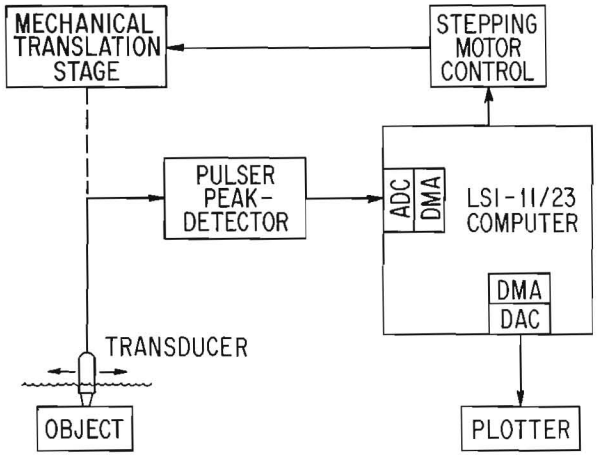
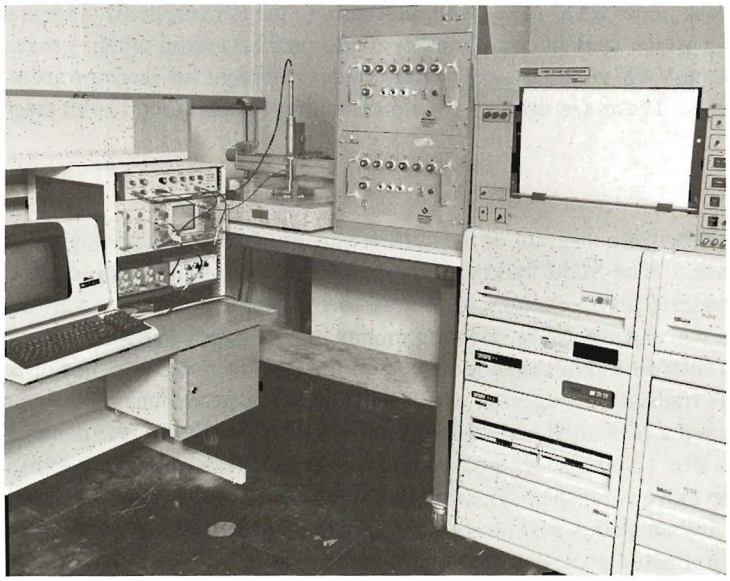


Figure 5. Photograph of Instrumentation.



The plotter is a Raytheon "Line Scan Recorder" (LSR) which makes the best hardcopy gray-scale images for this application. The LSR has three metal styli, spaced equi-distantly about a flexible belt which moves at constant speed (which is set from the 11/23). The styli electrically burn away layers of a conductive paper and exposed underlying carbon powder. In this way, a smoothly-varying gray-scale image is formed when the magnitude of the exciting voltage corresponds to the

magnitude of the echo signals from the object under investigation. A maximum magnification of 40 can be achieved with the present instrument, but the largest image that can be plotted is 18 inches across, the width of the recorder paper. The long dimension of the image can be several feet long, since the paper comes on a continuous roll.

Fast SUM System Concept

The mechanical scanner must traverse the object in two directions to make a raster pattern, but the electro-mechanical plotter can plot in only one direction. Thus, the plotters of earlier versions of SUM are idle as the scanner retraces its path to the beginning of a line. The primary objective of the SUM described here was to examine objects faster than previous instruments, by plotting data acquired in both scan directions. We used the computer to reverse the order of those data which came from a reverse scan. The original system design called for dual-port memories (with their associated controllers) on both input and output, to free the computer busses. As an approximation which lets us test the system concept, two (less expensive) DMA controllers were used to permit simultaneous data input, output and reversal or interpolation. It should be noted that it would not be necessary for the CPU to perform a reversal operation if commercial DMA controllers were capable of accessing data at decreasing memory locations as well as at increasing ones.

Model for Control Software

In this section we show how the main control routine for the application was derived from a consideration of its data flow graph. Figure 6 is a schematic representation of the physical paths of the moving elements of the SUM, while Figure 7 shows the path of data flow through the three processors and data buffers. In Figure 6 small arrows which point to descriptive names denote interrupts, which will be discussed later. The top part of Figure 6 shows a single cycle of motion of the mechanical scanner; the heavy lines represent the transducer path. It repeats that cycle until an X-Y raster has covered the object; an `END-SCAN` interrupt is generated after completion of each short step in the Y-direction.

The diagram at the bottom of the figure depicts the mechanical motion of the plotter. It runs continuously to drive the styli at constant speed in the indicated direction. Each stylus on the moving belt generates a `START-PLOT` interrupt as it reaches the beginning of a new line to be plotted. The `END-PLOT` interrupt is generated each time the output DMA processor completes its task.

Figure 6. Paths of Motion of Scanner (Top) and Plotter. Interrupts Used for Synchronization.

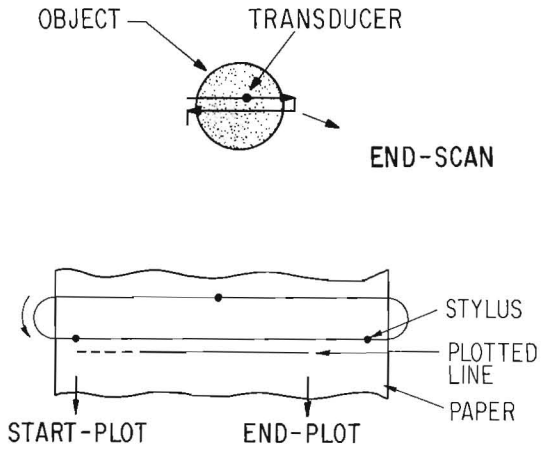
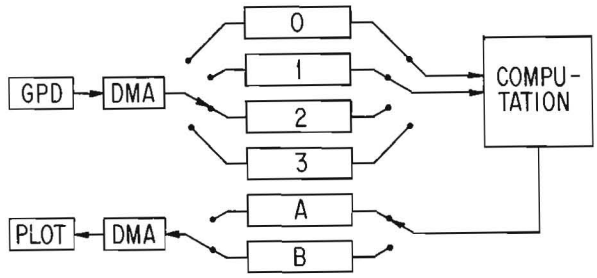


Figure 7. Data Flow and Buffer Structure. Data Buffers Reside in Extended Memory.



Coordinated Movements

Now, the mechanical motion of Figure 6 is intimately connected with the filling and emptying of the buffers of Figure 7. In order to assure positional accuracy the interfaces were designed such that data was clocked in and out of the DMA controllers by signals derived from the moving parts of the system.

All I/O was "double-buffered": one output buffer could be filled by the CPU while the other was emptied by the DMA controller. Four input buffers were used so that two at a time could be used for interpolation into one of the idle output buffers, while a third was being filled by the input DMA controller. Pairs of buffers were used so that future variants of SUM could take advantage of commercially available dual-port memories to further increase imaging speed by taking all I/O off the Q-bus.

In parallel with the input and output operations, the CPU performed its reversal or interpolation computations using two input buffers and one output buffer. We disabled the multi-tasking feature of our Forth system (eliminated the `PAUSE` word which calls the round-robin scheduler) so that the CPU was dedicated to a single task. The CPU was actually stopped (via a `WAIT` instruction) while waiting for an interrupt, thereby freeing the bus for DMA operations.

Main Control Routine

A central problem encountered with multi-tasking and multiprocessing systems is synchronization of tasks. Another is how to grant exclusive access to critical resources. Thus, the movements and concomitant I/O discussed above must be properly synchronized with each other and with the CPU, since all three independent processes run concurrently. In addition, none of the buffers must ever be accessed simultaneously by more than one processor. To control access to the buffers, we associated with each one of them a flag which denotes whether it is "empty" or "full."

We simplified the access and synchronization problems for our situation by making it a three-processor, single-task-per-processor system, and by using a master clock pulse to govern the operation of the system. Synchronization was accomplished by causing the two mechanical cycles to operate at multiples of the same frequency by using the `START-PLOT` interrupt as a master clock. What might have been asynchronous operation was transformed into synchronous operation. As part of the initialization procedure, we used the Line Time Clock to measure a half-cycle of the scanner motion, then set the speed of the LSR styli such that the time between `START-PLOT` interrupts (or a fixed number of them, for interpolation) was just long enough to complete the half-cycle. All processors were thereby caused to finish their tasks within the time period between an integral number of `START-PLOT` interrupts.

The main control routine can easily be designed by first constructing a timing diagram showing the sequence of operations. Figure 8 is such a diagram for the case of imaging without interpolation or filling. A more complicated one must be devised for the case of interpolation between lines, but it is not necessary for filling lines, since the `END-PLOT` interrupt handler can simply force the same data to be plotted over and over.

Figure 8. Timing and "State" Diagram of Imaging Process.

	0	1	2	3	0	1
0	I-F	M-A			I-F	M-A
1		I-R	R,M-B			I-R
2			I-F	M-A		
3	R,M-B			I-R	R,M-B	
A	0		0		0	
B		0		0		0

I-F = INPUT FORWARD SCAN
I-R = INPUT REVERSE SCAN
R = REVERSE DATA IN-PLACE
M-X = MOVE DATA TO OUTPUT BUFFER X
0 = OUTPUT DATA TO PLOTTER

In Figure 8 each vertical column represents a time slot, during which all operations must be completed. The top four lines represent operations that take place on the four input buffers, while the bottom two lines represent the action of the output DMA controller on the output buffers. Once the system has been properly initialized, it may be seen to repetitively cycle through a finite number of "states", within which the same sets of operations are performed on the same buffers. The timing diagram may then be treated as a state table; the state variable is the number of the input buffer being filled. (In this case, there are only 4 unique states.)

The control software reduces to the task of ensuring that for each state variable, the correct CPU and DMA processes get started. It is the responsibility of the interrupt routines (and terminating CPU routines) to ensure that the transitions between states take place smoothly by operating on the buffer flags. For the situation of Figure 8, the control routine can be written as a `CASE` statement, in which the state variable selects the branch path. Written this way, the action to be taken in each state can be shown explicitly, making the routine very easy to understand.

The control program is shown in Screen 1. The `CASE` selector (state variable) is derived from a count of the number of lines plotted (mis-named `PLOT-STATE`).

Some words and phrases used as debugging aids are shown in `BDSCAN` as examples for later reference: The vectored `EXECUTE` terminates operation if a non-zero "abort flag" was, at some earlier time, loaded by the CPU when executing elsewhere in the system. When debugging, unique abort flags can reveal which word was being executed if the system crashed within a word which contained one. The & is a word which captures the system conditions, and will be discussed below.

The Role of Interrupts

The `STARTPLOT-SYNC` word, shown in Screen 2, is used in the main control routine to force the 11/23 to enter the loop only on every `START-PLOT` interrupt, thereby synchronizing the system. It works by using `EI-HALT` to put the CPU in a `WAIT` state with interrupts enabled (in order not to load the system busses). Any interrupt will wake the CPU, which will immediately execute the appropriate interrupt handler. After the handler terminates, execution will resume just after the `WAIT` instruction to cause another cycle of the `BEGIN...WHILE...REPEAT` loop to take place. If `START-PLOT-FLAG` is set, (on `START-PLOT` interrupt) the CPU will exit the loop, permitting the next cycle of the control routine to proceed.

The primary function of the other two interrupts was to operate on the buffer flags. For example, the handler for `END-PLOT` declared the "current" output buffer empty and switched to the other buffer for output. It also advanced the paper for the next line. The `END-SCAN` handler declared the current input buffer "full" and also switched to the next input buffer. The details of each of the handlers may be found in Screens 4 through 7 where the comments will add further explication.

Part 2: Software Elements for Real-Time Control

In the remaining sections we describe some of the support software we developed in order to complete this project:

High-level interrupt handler
Extended memory access words
Debugging aids

High-level Interrupt Handler

It was conservative of CPU and bus resources to use interrupts to signal when tasks were finished. Because of the number and complexity of things that had to be done we wanted to execute high-level words within the handlers. It was especially important to have that capability for initial development and debugging.

Before we wrote our routines, we were unaware of earlier work in this area: Keck and Forsley [4] described a method for incorporating high-level handlers into a multi-tasking Forth system running on the LSI-11. Melvin [5] presented some general considerations on high-level interrupt handling, along with a specific case implemented for the 8080.

Our approach was different from either of the previous ones, although the end results obviously contain similar elements. We chose simply to extend the conventional assembly-language method in a very straightforward way, to produce an "in-line" handler within which high-level and `CODE` words would be permissible. In contrast to [4] and [5], we set up our words to be analogous to `:` and `;`. Thus, the word `:INT` defines the beginning of an interrupt region, and `INT;` terminates it. We didn't find it useful to vector the execution of the interrupt routine as in [5], since our handlers always performed the same function and there were several interrupts, each with its own handler. When necessary (debugging, for example), a different handler could be attached to a given interrupt by executing the `INTERRUPT-ON` word which loads the handler starting address into the interrupt vector. Since the SUM application was run as the only task in a multi-tasking environment (the multitasking aspects of which were disabled to free the bus), it was normally sufficient to load the interrupt vectors when the application was loaded.

Our interrupt handler words are shown in Screen 3, along with examples of their use as discussed above. Our approach was to create, for each interrupt, a headerless region in the dictionary which would contain the handler code. The interrupt vectors were then loaded with the starting addresses of the handler code.

Operation of :INT

Assume that the definition for `:INT` has been compiled into the dictionary. We will now trace its operation as it executes. (Although we are compiling the definition of the interrupt handler into the dictionary, the text interpreter is in the execution mode when `:INT` and `INT;` are encountered.) First, `HERE` gets executed and places the address of the next available memory location on the data stack. Then, the words from the `ASSEMBLER` vocabulary are invoked. When these words execute, they actually store into memory (the dictionary) the desired machine instructions. Thus, most of the word is essentially a macro which stores machine instructions into memory, beginning at the address that `HERE` produced. The `DO` loop stores instructions to push registers R0 through R5 onto the system stack. Next, a `MOV #STATUS,R3` instruction is compiled into the dictionary. `STATUS` contains a pointer to the current user's task control block. R3 is the PDP-11 register specified to contain the pointer to the user area (U) in our implementation of fig-FORTH. The next two lines of code prepare the 11/23 to start executing high-level Forth words: instructions are compiled to initialize the Interpretive Pointer to begin execution just after the code for `NEXT,`. Then the code for `NEXT,` itself is compiled into memory. The real work of `:INT` is essentially finished at this point.

It is instructive to digress and see what happens when the code that was just compiled actually executes: upon receipt of an interrupt, the CPU pushes the PC and PSW onto the stack, and reloads them with values gotten from the interrupt vectors. Thus execution begins at the interrupt handler prefix we just created. Registers 0 thru 5 are pushed onto the stack, the address of `STATUS` is stored in Register 3, the IP is initialized to begin interpreting high-level Forth code at the location just beyond `NEXT,` then the code for `NEXT,` is executed. Thus, we simulate the termination of a `CODE` word as `NEXT,` goes to the next word to be executed.

We return to the actual creation of the interrupt handler. At this point, it is only necessary for `:INT` to compile the CFA's for the high-level words that are to be part of the interrupt handler. After the code for `NEXT,` is compiled into the dictionary, `]` executes and switches the text interpreter back to compilation mode. Thus, when `:INT` finishes executing and releases control, the interpreter is left in the compile state, and begins compiling the words that follow it into the dictionary. Thus, the CFA's of all high-level Forth words which follow `:INT` on the screen will be compiled into memory as if they were part of a colon definition.

Operation of INT;

Execution of `:INT` essentially results in an "unfinished colon definition" in the sense that `NEXT` was not compiled as the last CFA in the string. The word `INT;` terminates the handler region; its job is to place code at the tail of the definition being built so that, at run time, the machine gets restored to the original conditions before the interrupt.

When `INT;` is defined, it is flagged as `IMMEDIATE` so that, when it is encountered at compile-time (compiling the interrupt handler definition into the dictionary) it gets executed, as did `:INT`. It operates by building what is essentially a `CODE` definition into the dictionary. The first two lines of executable code store two successive addresses in the dictionary just after the last CFA from the high-level part of the interrupt handler. The first address will be interpreted as a CFA by the address interpreter. The word that it points to just happens to reside two bytes higher in the dictionary. At that location is stored the second address (which just happens to point two bytes above), which points to the beginning of the actual code that the inner interpreter will execute. Thus, this part of the handler makes the transition from the address interpreter to the execution of machine code. We have essentially simulated a code word. The instructions that will be executed are pops of R0 thru R5, then execution of an RTI instruction. After that Forth can resume, since all the registers will have been restored to their original condition. The last line of the `INT;` definition forces the text interpreter to enter the execution state after `INT;` finishes compiling the machine instructions into the dictionary. This terminates compilation of the high-level interrupt handler.

Using Extended Memory

This application required much more memory than that available within the standard Forth memory space. The Line Scan Recorder has a resolution of 4000 pixels, so 4096 words were reserved for each scan line buffer. The software model requires 6 data buffers per scan line, so a total of 24k words was necessary for data storage alone. We used an extra 32k words of extended memory to provide these large data buffers.

The DMA processors were programmed to transfer data directly to and from extended memory using their extended addressing bits. However, the CPU was obliged to perform some calculations on that data. It was necessary to reverse the input data from backward scans in real time. Also, as part of the initialization procedure, the CPU measured the backlash present in the drive screws and calculated a correction factor. These operations had to be performed in local memory, so we wrote Forth words to store and fetch from extended memory, and to dump extended memory for debugging.

Leary and Winkler [6] described three memory-management schemes for the Forth environment. We became aware of their earlier work after we had written our routines, and the approach

we took was similar to one of the methods they described. However, there were sufficiently important differences in detail between our implementations that we briefly describe ours here. In particular, the "fetch" and "store" words described here should execute slightly faster than those in [6].

The method we developed independently corresponds to the "method 3" of Reference [6], to which the reader may refer for further details. In contrast to [6], we set up the mapping registers and processor status word to keep the kernel/user spaces and memory mapping permanently enabled; our application was run with the CPU in kernel mode.

Our ideograms for accessing extended memory are shown in Screen 8. The "long-fetch" word `L@` takes a 16-bit address from the data stack and leaves the 16-bit word that resided at that address in extended memory. It uses the PDP-11 MFPI instruction, which pushes a 16-bit word from the "previous space" or memory space of the previous mode (extended memory) onto the current stack. The MFPI instruction gets the address, from which to retrieve the data, by popping it off the data stack. After MFPI pushes the desired data onto the system stack, it must be moved to the Forth parameter stack since in our implementation of Forth the 11/23 stack pointer R6 functions as the Forth return stack RP. The system stack is denoted by SP in the source code. Thus, the data fetched from extended memory is popped from the Return Stack and pushed onto the data stack S. Since we permanently keep the address registers enabled and the modes enabled, this word uses two fewer `MOV` instructions, and should execute faster than those in [6].

The long-store word `L!` operates in a manner like that of `L@`. Here, a 16-bit data value n on the parameter stack is stored at an address (also on the stack) in extended memory. In order to use the MTPI instruction, the desired data must first be pushed onto the system stack (our return pointer SP). This is done by the first `MOV` instruction, which takes the second value on the stack and pushes it onto the system stack. The next `MOV` instruction pops the intended address off the parameter stack, then places it over the item on top of the stack. The reason for this operation is to simply clean up the parameter stack for the return. The MTPI instruction pops its address off the parameter stack (leaving it empty), pops the data off the system stack (SP) and stores it in extended memory at the specified address. Again this word should execute somewhat faster than the corresponding one in [6] since two less `MOV` instructions were used here.

We avoided setting up the PSW on every invocation of `L@` and `L!` by using the scheme set forth in the `INIT-MAP-MODE` word, shown in Screen 9. Once executed, it permanently set up the kernel and user mode parameters. We took advantage of the fact that the Current Mode and Previous Mode bits in the PSW get loaded from the current stack when an RTI instruction is executed [7].

The sequence of operations in `INIT-MAP-MODE` is:

- Disable memory management hardware
- Push a copy of the PSW onto the system stack
- Clear the upper 8 bits of the PSW (leaving the lower 8 bits unaffected)
- Set bits 12:13 in the PSW, denoting that the Previous Mode was User Mode, and that the Current Mode is Kernel Mode
- Push the address just after the RTI instruction onto the system stack
- Execute an RTI instruction.

The RTI pops the return address and the previous PSW off the stack into the PC and PSW, respectively. Recall that the stack was set so that execution would resume just after the RTI instruction, with the PSW indicating that the Previous Mode was the User Mode, and that Current Mode is Kernel Mode. Thus, the CPU starts executing in Kernel Mode at the instruction which enables memory mapping. Prior to the execution of `INIT-MAP-MODE`, the memory-management registers must be set up with `INIT-MAPPING`, shown on the same Screen. This word is executed only once when the application is `LOAD`ed, so it is written in high-level Forth. Using `L@` and `L!` one can rewrite standard Forth words just by substituting for `@` and `!`.

Debugging Real-Time Systems

In this section, we would like to comment on ways to debug a real-time system, point out some difficulties peculiar to our system and share some methods and tools we found helpful. Finally, we will present details of a useful debugging tool we developed.

The natural way to implement a well-designed Forth program is from the "bottom up": The most primitive words are written and fully debugged before incorporation into other words. An analogous approach should be followed when integrating parts of a complex software system which must respond to interrupts in real-time. One should break the system up into isolated parts, each of which can be individually debugged, starting from the lowest level of system complexity. After all program units are separately debugged they can be tested together, building to the complete application step-by-step. Following such a procedure may not guarantee that all bugs are found, since there are often unexpected interactions between different parts of the system. Nevertheless, it seems to be the best approach.

Debugging an interrupt-driven real-time application can be frustrating, to say the least, and nearly impossible without the proper tools. One can distinguish at least three failure modes:

(a) Application fails with Forth (and high-level debug tools) available (soft crash).
(b) System crashes, Forth not available; only ODT-like debugger available, Forth registers and memory intact (hard crash).
(c) System crashes into ODT, registers and memory corrupted, must re-boot.

Case (c) must be avoided. Once one has progressed to the point where all system failures fall into categories (a) or (b), tools can be developed to assist debugging. In the following, only failures of type (a) or (b) will be considered.

Hard Crashes

Debugging a hard crash can be especially intimidating for the novice user of Forth: Besides having to know the machine language and architecture of the CPU in use, one must know the details of the Forth "computer" itself. In this case one is usually forced to trace the paths of the Forth inner interpreters through the dictionary, in order to find the problem. A useful debugging tool would leave data in known memory locations so that they could be examined with the built-in debugger/monitor.

Soft Crashes

When the application fails but it is possible to regain keyboard access to Forth, then Forth itself can be used as a debugging tool, as with conventional applications. In addition, special tools can be developed which exploit the nature of the application and make it easier to debug.

In addition to the standard debugging tools and lore (change only one thing at a time), there were a few specific tools and rules that seemed to be well-suited to real-time applications:

1. Use vectored abort routines so that Forth can regain control after a "soft" crash (no need to re-boot).
2. Place unique "abort flags" in selected regions of code to help determine the cause of the abort.
3. Use "pass counters" to record the number of times a particular section of code was executed.

Occasionally one encounters problems that are the fault of the computer manufacturer. We had a particularly troublesome problem that we could reproduce with such simple code that the solution must lie elsewhere: The 11/23 would sometimes take several milliseconds to respond to a `START-PLOT` interrupt. Upon calling DEC, we learned that the revision level of our CPU card indeed could not reliably handle simultaneous interrupt and DMA requests! The problem disappeared after we bought the upgraded board.

Other lessons learned: Don't be too concerned if the actual operating performance speed is lower than that calculated using published specifications. Performance of the computer may be critically influenced by what operations are taking place (or trying to take place) on the bus. It may be difficult

for manufacturers to measure and specify bus acquisition/relinquishment times when multiple DMA or interrupt requests are pending.

Bus contention is a major problem with multiprocessor systems which share a bus. The data and address busses are themselves critical system resources in these cases. That is why we tried to off-load the busses as much as possible in our system. Recall that in the optimum system design dual-port memory would be used to completely free the computer busses.

Snapshots of System Crash

When debugging our system, it was particularly helpful for us to have a sequential record of the buffer states (and other variables) as the CPU traced a selected path through the code. We wrote a "snapshot" routine which produced such a record. It stored the buffer states, counters, etc. in a large array (snapshot buffer). If the system crashed in a way such that Forth was still available, one could immediately print out the contents of the snapshot buffer, trace the execution path, and soon find the problem. The routine was even helpful when a hard crash occurred, because one could examine the snapshot array with ODT.

Listing 1 shows a typical snapshot dump, while Screen 1 shows how the snapshot word & was used. We inserted it into the load screens, followed by a unique decimal digit string, at places where we wanted a "snapshot" of the system. When the snapshot word executed, it stored the number corresponding to the string in a memory buffer, followed by the states (empty/full) of the buffers and flags, and the values of the selected variables. When the snapshot buffer was dumped, it revealed the sequence of events along the selected execution path, including high-level interrupt handlers.

The snapshot word is shown in Screen 14. It works in the following way: The ideogram & is flagged as immediate, so when a screen containing it is loaded, & is executed. After a test to make sure that we are compiling, the current number base is stored on the return stack and the base switched to decimal. The next line of code converts the ascii number in the input stream (decimal digits) into its binary equivalent, which is left on the stack. `LITERAL` then compiles this stack value into the dictionary as 16-bit literal. The CFA of `(&)` is then compiled into the dictionary, so the number gotten above and the run-time behavior of & are both compiled as part of the definition of the word which contains &.

Run-Time Behavior of &

When the word containing & executes, the literal number gets put on the stack, and the code for `(&)` is executed. As may be seen in Screen 3, all it does is take a number off the stack, store it in the snapshot buffer, and then increment the buffer pointer. It then stores the values of the selected variables. The other related words are self-explanatory from the comments.

Concluding Remarks

This application falls into the classical motion-control/data-acquisition category that has proven the worth of Forth so often in the past. In this article we have shown how we used the language, with simple extensions, to control a real-time imaging system built with three active data processors. Although the two DMA processors performed simple data movement tasks, the application was nevertheless a good introduction to treatments of some of the classical problems encountered with multiprocessor systems. This was especially true of the debug phase.

Acknowledgements

We are very grateful to Robert Gilmore for sharing his expertise and insights about ultrasonic macroscopes and their applications. The excellent image quality of the present SUM is due in large measure to the gated peak-detector which was designed by John Young. He also helped interface it to the 11/23, and helped with other electronics-related problems, for which we are grateful. In addition, we would like to thank them for the many suggestions they made during discussions held

in the early stages of the overall system. We would also like to thank Bruce Bernstein and Nelson Corby for helpful discussions regarding the PDP-11, Rudy Koegl for critically reviewing our software design, and Lee Clark for help in bringing up Forth on our 11/23. Finally, we would like to thank Ray DeLarosa for expert technical assistance with building and debugging the several pieces of interface hardware.

References

1. Gilmore, R.S., Viertl, J.R., and Halberg, L.I. "Materials Characterization by Acoustic Microscopy", *Materials Technology*, Spring 1981.

2. Gilmore, R.S. "Materials Characterization by Acoustic Microscopy", Abstract of invited paper in *Bull. Am. Phys. Soc.*: 27 (1982), p. 881.

3. Gilmore, R.S., Miller, R.G., Tam, K.C., Wood, D.E., and Young, J.E. "Computer-Assisted Ultrasonic Microscopy: Materials Characterization, Industrial Applications", *Proceedings of 14th Symposium on NDE*, San Antonio, Texas, 1983. Published by Southwest Research Institute, San Antonio, Texas 78284.

4. Keck, R.L. and Forsley, L.P. "A High Level Interrupt Handler in Forth", *FORTH Dimensions*,Vol. 3, No. 4:116-7, 1981.

5. Melvin, Stephen, "Handling Interrupts in FORTH", *FORTH Dimensions*, Vol. 4, No. 2:17-18, 1982.

6. Leary, Rosemary C. and Winkler, Carole A. "Mapped Memory Management Techniques in FORTH", *FORTH Dimensions*, Vol. 3, No. 4:113-115, 1981.

7. *Microcomputers and Memories*, Digital Equipment Corporation, Maynard, Massachusetts, 1982, p. 274.

Manuscript Received January 1985.

```
0001

( BDSCAN                                                12 AUG 83 )
: BDSCAN   ( [ n1 n2 n3 ==> ] n1 Ymil, n2 Xmil, n3 plot/scan )
   DUP #PLOT/SCAN !  /  SWAP  2*  ( [ #lines #pix ==>  )
   4 #STATES !      INIT-SCAN
   BEGIN        STARTPLOT-SYNC
    ABO-FLAG @    IF  ABO-VEC @  CFA EXECUTE  ENDIF
    PLOT-STATE @  PREV-PLOT-STATE @  <>
    IF ( not filling or interpolating )
      PLOT-STATE @  DUP  PREV-PLOT-STATE !   #STATES @  MOD
     <CASE 0   & 1    0 START-SCAN    3 RE-VERSE    3 B PUT
      CASE 1   & 2    1 START-SCAN                  0 A PUT
      CASE 2   & 3    2 START-SCAN    1 RE-VERSE    1 B PUT
      CASE 3   & 4    3 START-SCAN                  2 A PUT
      CASE>
    ENDIF  PLOT-STATE @  LINECOUNT @  >=  ( done? )
   UNTIL  DMACSR READY?   ABO-VEC @  CFA  EXECUTE ;
```

```
0002

( DI  EI  EI-HALT  STARTPLOT-SYNC                    03 JAN 85 )
OCTAL

CODE DI   ( Save old intp mask, disable interrupts )
   SP -) MFPS,
   340 # MTPS,   NEXT,   C;

CODE EI   ( Restore old interrupt mask )
   SP )+ MTPS,   NEXT,   C;

CODE EI-HALT   ( Restore old interrupt mask and halt )
   SP )+ MTPS,   WAIT,   NEXT,   C;

: STARTPLOT-SYNC   ( Wait to synchronize with startplot )
   BEGIN  DI  STARTPLOT-FLAG @  0=
   WHILE  EI-HALT  REPEAT  0 STARTPLOT-FLAG !  EI ;

0003

( :INT  INT;                                          22 APR 83 )
DECIMAL

: :INT ( [ ==> addr ] Define high-level interrupt handler )
   ASSEMBLER HERE    ( return intpt hndlr entry point addr )
   6 0 DO  I  SP -) MOV,  LOOP ( Save registers )
   STATUS # U MOV,  HERE 8 + # IP MOV,    NEXT,
   FORTH  ] ( Start compiling ) ;

: INT; ( Exit high-level interrupt handler )
   ASSEMBLER
   HERE 2+ , ( Do the following ) HERE 2+ , ( code word )
   0 6 DO  SP )+  I 1- MOV,  -1 +LOOP  ( Restore registers )
   RTI, ( Return from interrupt )
   FORTH  [COMPILE] [ ( Stop compiling ) ; IMMEDIATE

0004

( X-AXIS-HANDLER                                      04 MAY 83 )
OCTAL

: X-AXIS-HANDLER
   XAHP# 1+!
   1 YFLAG !     ( Y ready for another move )
   1 XFLAG !     ( X also ready to move again )
   XLEN 2@   DMINUS   XLEN 2! ( Change X- direction )
   XAHP# @  YAHP# @  <>
   IF  ( X and Y axes out of phase )   1 ABO-FLAG !  ENDIF ;

                                                        -->
```

```
0005

( STARTPLOT Interrupt Handler                          04 JUN 83 )

:INT      SPP# 1+!   OBUFFSTATE OUTBUFF @ 2* + @
    IF ( buffer full )  10107 LSRDMA !  ( plot the line )
     FILL-COUNT @ 0=
     IF ( done filling )  PLOT-STATE 1+!  ( bump line counter )
      #PLOT/SCAN @  MINUS  FILL-COUNT ! ( Reset fill counter )
     ENDIF
    ENDIF   CLRST2FLG ( clear interrupt flip-flop )
    1 STARTPLOT-FLAG !    INT;
:  STARTPLOT-ON ( Use STARTPLOT-OFF to disable )
    LITERAL 444 ! 340 446 ! ( Disable interrupts during intpts )
    [ 2 CSP +! ]     ( Adjust CSP for literal pulled off stack )
    CLRST2FLG LSRCLK @ 40000 OR LSRCLK ! ; ( Enable ST2 intpts )
:  STARTPLOT-OFF
    LSRCLK @ 37777 AND LSRCLK ! ;                          -->

0006

( ENDPLOT Interrupt Handler                            04 JUN 83 )

:INT   ( load lsrdma registers for next )
       ( transfer and change outbuff pointer for next cycle. )
    EPP# 1+!   FILL-COUNT 1+!   FILL-COUNT @
    IF ( inserting fill lines, just reload address register )
      OUTBUFF @ 2*  OBUFFLIST  +  @  LSRDMA  6  + !
     ELSE  ( FILL-COUNT = 0, so finished with line. )
      DI  0 OBUFFSTATE OUTBUFF @ 2* + !  EI ( Set buffer empty )
      SWCH-OUTBUFF  LSRDMA 6 + ! ( Swch bffr,pntr,load register)
    ENDIF  ADVLIN  ( advance paper )
    LDLSRPCR  ( load point count register )   INT;
    350 !   0 352 !      ( Let this routine be interrupted )
                          ( Disable with ENDPLOT-OFF )
:  ENDPLOT-OFF
    LSRDMA @   177677 AND   LSRDMA ! ;                     -->
```

```
0007

( END-SCAN Interrupt Handler                              20 JUN 83 )
( Interrupts are enabled by the constant sent to the MOVE word)
( and disabled by clearing bits in the CSR.
0 VARIABLE GPDWCR ( Gated-Peak Detector word count register )

:INT
   ESP# 1+!   0 SENDOUTBUF  ( Clear interrupt flip-flop )
   YFLAG @   0=
   IF ( Y-axis in motion )      DMACSR @  200 AND
     IF ( Gated Peak Detector finished ==> no error )
       1 IBUFFSTATE INBUFF @ 2* + ! ( set input buffer full )
       X-AXIS-HANDLER
      ELSE   PKDDMA @  GPDWCR ! ( Save for debug )
     ENDIF
   ENDIF
   INT; ( Handler addr now on stack )  364 ! 340 366 !

0008

( L@  L!  Memory management                               22 APR 83 )
OCTAL  ASSEMBLER DEFINITIONS
006500 1OP MFPI,   006600 1OP MTPI,
FORTH DEFINITIONS

172300 CONSTANT KPDR   KPDR 40 + CONSTANT KPAR
177600 CONSTANT UPDR   UPDR 40 + CONSTANT UPAR
177572 CONSTANT SR0

CODE L@   ( [ addr ==> n ] Long-fetch from previous space )
   S @)+ MFPI,  SP )+ S -) MOV,   NEXT,   C;

CODE L!   ( [ n addr ==> ] Long-store to previous space )
   2 S) SP -) MOV,   S )+ S () MOV,   S @)+ MTPI,   NEXT,   C;
                                                          -->

0009

( INIT-MAP-MODE  INIT-MAPPING                             03 JAN 85 )

CODE  INIT-MAP-MODE  ( Set PSW and enable mapping )
   SR0 @# CLR,   ( disable memory management )
   SP -) MFPS,   ( get status )
   177400 # SP () BIC,  30000 # SP () BIS,   ( PS: kernel/user)
   HERE 6 + # SP -) MOV,   ( PC )  RTI,   ( set new PSW and PC)
   1 # SR0 @# MOV,   ( enable memory management )   NEXT,   C;

: INIT-MAPPING   ( Initialize memory management regs. )
   DI  10 0 DO   ( 0 thru 7 )
      I 7 = IF  7600  ELSE  I 200 *  ENDIF  KPAR  I 2* + !
      77406 KPDR  I 2* + !
      I 10 + 200 *  UPAR  I 2* + !   77406  UPDR  I 2* + !
   LOOP  INIT-MAP-MODE  EI ;
```

```
0010

( SNAPSHOT words )                                    04 JUN 83 )
DECIMAL
200 CONSTANT SNAPS  ( for maximum of 200 times )
26 CONSTANT TB-L    ( capture 13 variables )
SNAPS TB-L * CONSTANT TB-LEN
0 VARIABLE TB-PTR
0 VARIABLE TB-TOP   TB-LEN  2 - ALLOT          TB-TOP TB-PTR !
: 2SP 2 SPACES ;
: 4SP 4 SPACES ;
: 8SP 4SP 4SP ;
: S ( [ n ==> ] Print n spaces ) SPACES ;

                                                          -->

0011

( SNAPSHOT words                                      04 JUN 83 )

: .TB   ( [ addr ==> ] Print snapshot at addr )
      DUP CR @ 4 .R ( Print label )
   2+ DUP @ 4SP  5 .R
   2+ DUP @ 8SP .
   2+ DUP @ 2SP .
   2+ DUP @ 2SP .
   2+ DUP @ 2SP .
   2+ DUP @ 4SP .
   2+ DUP @ 2SP .
   2+ DUP @ 8SP .
   2+ DUP @ 4SP .
   2+ DUP @ 4SP .
   2+     @ 8SP .  ;                                    -->

0012

( Snapshot routines continued                         04 JUN 83 )

: 2+! ( [ addr ==> ] Increment memory by 2 at addr )
   2 SWAP +! ;
: GRAB ( [ n ==> ] Store n in current snapshot bffr loc )
   TB-PTR @ ! ;
: BUMP ( [ ==> ] Point to next snapshop buffer loc )
   TB-PTR 2+! ;
: .SNAP
   CR 18 SPACES ." INPUT BUFFER STATES OUTPUT BFFRS "
   CR ." Label  PLOT-STATE    (0) (1) (2) (3)  "
   ."   (0) (1)   OUTBUFF INBUFF XFLAG YFLAG    FILL-COUNT "
   TB-PTR @  TB-TOP  DO  I .TB  TB-L +LOOP ;
                                                          -->
```

```
0013

( Snapshot routines continued                          04 JUN 83 )
: (&) ( [ n ==> ]   Take snapshot for trace )
DI   ( Decimal value of string in input stream now on stack )
   ( Label )          GRAB   BUMP
   PLOT-STATE      @  GRAB   BUMP
   IBUFFSTATE      @  GRAB   BUMP
   IBUFFSTATE 2+   @  GRAB   BUMP
   IBUFFSTATE 4 +  @  GRAB   BUMP
   IBUFFSTATE 6 +  @  GRAB   BUMP
   OBUFFSTATE      @  GRAB   BUMP
   OBUFFSTATE 2+   @  GRAB   BUMP
   OUTBUFF         @  GRAB   BUMP
   INBUFF          @  GRAB   BUMP
   XFLAG           @  GRAB   BUMP
   YFLAG           @  GRAB   BUMP
   FILL-COUNT      @  GRAB   BUMP
                                                       -->

0014

( Snapshot routine continued                           03 MAY 83 )

  TB-PTR @  TB-TOP - TB-LEN >=
  IF  ( buffer full )
     TB-L MINUS TB-PTR +! ENDIF   EI ;

: &     ( Take snapshot for trace )
   ?COMP
   BASE @ >R DECIMAL
   32 WORD HERE NUMBER DROP
   [COMPILE] LITERAL  COMPILE  (&)
   R> BASE ! ; IMMEDIATE
```

Dr. Lynk is currently engaged in teaching and the establishment of a laboratory for ultrasonic materials characterization at CCNY. He was a staff member at General Electric's Corporate Research and Development Center from 1974 to May, 1985. While there, he helped develop diagnostic imaging systems for ultrasound non-destructive evaluation, medical ultrasound and x-ray computed tomography.

Mr. Johnson is currently Director of Research and Development at Advanced Computer Applications, Inc. in Newtown PA, responsible for the development of integrated software and hardware turnkey systems for industrial process control. These advanced systems are 32-bit multi-processor based and utilize leading edge software and hardware technologies. He was a staff member at General Electric's Corporate Research and Development Center from 1980 to 1983. His primary activities focused on the development of threaded interpretive languages for use in knowledge-based "expert" systems and interactive real-time control of robotic systems.