LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR -85-1817

RECEIVED BY OSTI JUN 07 1985

CONF-850513--4

LA-UR--85-1817

DE85 012736

TITLE: USING AN LSI-11/23 AND RT-11 TO DIGITIZE ANALOG TAPES

AUTHOR(S) John N. Stewart, Geophysics, ESS-3

SUBMITTED TO 1985 Spring DECUS U.S. Symposium
New Orleans Hilton,
May 27-31, 1985

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

USING AN LSI-11/23 AND RT-11 TO DIGITIZE ANALOG TAPES

John N. Stewart
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

I describe in this paper the design and implementation of a system on an LSI-11/23 under RT-11 that digitizes 14-channel analog tapes at given intervals and writes the digitized data to a magnetic tape. The analog tapes contain 13 channels of data and one channel of IRIG-B time code. A given interval is selected by entering a time and an interval. The IRIG-B time code is read from the analog tapes and used to start the digitizing. A programmable clock board is used to control the digitizer and to count the interval length. In order to achieve the through-put rates needed, I wrote the code in MACRO and addressed the magnetic tape controller directly instead of using programmed requests.

THE PROJECT

This project started in the spring of 1984 when I was asked if there were any codes to digitize 14-track analog tapes. My reply was that there were none, but that I could probably write one if they were not in a big hurry. I had several LSI systems digitizing 24 hours a day and storing the data on magnetic tape, so I was familiar with the equipment and could program it. I asked a few questions to pinpoint exactly what was wanted. Essentially, there were several 14-track analog data tapes containing seismic events at various times. Of the 14 tracks, 13 were data and 1 was IRIG-B time code. The time and length of the events were known. The desired digitizing rate was 500 samples per second per channel. The digitized data would be written to magnetic tape.

The equipment situation looked good. I had another LSI system running RT-11 (SJ) which was used for development work and also included a magnetic tape unit, a digitizer board, and a programmable clock board. There was an IRIG-B time code reader available, and an analog tape playback unit could be found. Figure 1 shows how the system looked. I decided to write the software in MACRO to gain speed, simplicity, and memory over writing it in a high level language.

The 16-channel digitizer that would be used could handle +/-10 volts, +/-5 volts, 0-10 volts, or 0-5 volts. It could do a digitization in 25 microseconds and had an automatic channel sequencer. I felt the actual digitizer loop in the code would take less than 100 microseconds which would allow a digitizing rate of better than 10,000 samples per second. Since I only needed 500 samples per second per channel for 14 channels or 7,000 samples per second I seemed in good shape. I would have the code digitize the selected channels of data as quickly as possible and then wait for the next clock interrupt rather than doing an equal interval type of digitizing. This would provide enough time between sampling intervals to switch buffers and start the write tape process when a buffer filled.

There were no apparent problems with memory. The code would use two output buffers, each big enough to hold one second of data, which would give me plenty of time to write a buffer to magnetic tape while putting data into the other buffer. The size of each buffer would be 7,000 words (14,000 bytes), slightly large for a standard magnetic tape record but acceptable.

I did not see any problems writing the magnetic tape. The tape unit had a density of 1600 bytes per inch and moved at 75 inches per second. The code should be able to easily write 14,000 bytes in less than a second.

THE FIRST PROBLEM

The only analog tape playback unit that could be found forced playback at twice real time which

Figure 1. The Equipment.

meant the code had to digitize at twice the desired rate or 1000 samples per second per channel. That is a total of 14,000 samples per second (28,000 bytes). This meant the digitizer had to take a sample in less than 71 microseconds and also make available an extra slot of time every second to switch buffers and start writing out the old buffer to magnetic tape. Again, considering the 25 microsecond digitizer conversion cycle and needing only a few instructions to digitize a channel, I felt the speed up could be handled.

The amount of memory for the two 1 second buffers would double to 28,000 words (56,000 bytes). This would not leave enough memory for the code even though I would be programming in MACRO. I needed to go to smaller buffers and, hence, records; but I did not know the amount of time the tape unit used to start and stop when writing a record (record overhead time). I was also starting to worry about the volume of data the tape unit could handle in one second. I searched through the tape manual and found the information as shown in Table 1.

TABLE 1

TAPE TIMING (for 9 track, 75 IPS tape)

Ramp time (getting up to speed)	=	5.000 milliseconds
Write gap delay	=	1.000 millisecond
End of record (7 blank frames, CRC, 8 blank frames, LRCC)	=	.067 milliseconds
Write stop delay	=	.300 milliseconds
Ramp time (slowing to a stop)	=	5.000 milliseconds
Record overhead time	=	11.367 milliseconds
Time to write 1000 bytes (1 inch)	=	13.333 milliseconds

Therefore, a 28,000 byte (1 second) record would take:

(28,000/1000) * 13.333 + 11.367 = 373.333 + 11.367 = 384.700 milliseconds.

I had thought that tape operations were much slower and so I was pleasantly surprised with the above numbers. With the record overhead time being so short, I could go to a more reasonable record size of approximately 4000 bytes and the buffers would only total to approximately 8000 bytes. This would solve the memory problem.

A 4000 byte record would take 64.7 milliseconds to write. I decided to have the code calculate a record size based on the digitizing rate such that the record would contain more than 100 wall clock milliseconds of data. This would allow time to switch buffers and start the tape writing between sampling intervals.

THE TIME CODE READER

The next consideration was how to start digitizing. At first I had planned to take the easy way out. I would have the user position the analog tape close to the desired time by looking at the IRIG-B time code reader display, and then have the user start the analog tape reading and press a key on the input terminal to signal the code to start digitizing. The IRIG-B time channel could be digitized to enable the user find the real time when the digital tape was processed. For one or two events this would do, but not for hundreds of events. I remembered someone saying there was a BCD output on the time code reader. I checked the manual and found that the time code reader had a BCD time output of 32 bits giving seconds through days. I only had one 16-bit parallel interface board so settled for 7 bits of seconds, 7 bits of minutes, and 2 bits of hours. The 16-bit BCD time code is shown here:

HOURS	MINUTES	SECONDS
-- --xx	xxx xxxx	xxx xxxx

where x represents 1 bit.

For example: time 16:58:21 would be received as the following 16 bits

10 101 1000 010 0001
(2 5 8 2 1).

Allowing only 2 bits for hours meant the user would have to keep track of the hours. Table 2 shows the correspondence between the 2 bit hour and the actual hour.

TABLE 2

HOUR BITS	00	01	10	11
	0	1	2	3
HOUR	4	5	6	7
	8	9		
COULD	10	11	12	13
	14	15	16	17
BE	18	19		
	20	21	22	23

Since the code would now have access to tape time, I decided to have the user set up a list of event times. The code would compare the event time with the analog tape time and, when found equal, would start digitizing. I would have the code convert the inputted time from ASCII to a BCD time so only one test needed to be done to check for the correct time.

Another problem occurred in that if the code was continuously looking for a time when the user was positioning the analog tape, a false time match could occur when changing analog tape speed to

fast forward and back to regular speed. To get around this I added a flag to each event time that if set to "P" would cause the program to pause until a character was entered at the terminal, otherwise the code would continue checking for the next time. This would allow the user to manipulate the analog tape without worrying about causing false time matches.

PUTTING IT TOGETHER

Having thought the project through, it was time to put the code together. The code was named FASDIG for the obvious reason that the code would be digitizing much faster than any of the other codes. What follows mostly reflects the existing state of FASDIG.

FASDIG uses the .GTLIN programmed request to read the input file. This was the only routine I could find to read a line of ASCII characters from a file. Therefore, the user is forced to run FASDIG from an indirect command file. The first line of the input file has to be "RUN FASDIG". The input data follows that. The first line of data contains 4 numbers: the number of channels, the number of .1 milliseconds (wall clock) per digitizing interval, the numerator and the denominator of the ratio of wall clock time to data tape time. Following that are up to seventeen 60-character lines for describing the tape and each of the channels being digitized. Then there is one line for each event consisting of the starting day, hour, minute, second of the event, number of seconds to keep, and the pause flag. The event time is in terms of analog tape time, the number of seconds to keep is in terms of wall clock time. I did not like mixing time bases but did not want to convert number of seconds from analog tape time to wall clock time which was needed for the digitizer. An example of an input file follows:

```
RUN FASDIG
 4,10,2,1,      4CHAN,1000S/SEC,RATIO=2
 EXPERIMENT # 2052  SS MEGA PUMP
CH1- EE-1 VERT (4V)  PRECAMBRIAN
CH2- CENT      (4V)  PRECAMBRIAN
CH3- BANC      (4V)  SURFACE
CH4- LAFK      (4V)  SURFACE

 341,04,24,36,10,P,
 341,04,34,22,10, ,
 341,04,34,50,10,P,
```

The first action of FASDIG is to execute the SETUP routine. The routine prints a startup message as follows:

```
MUST RUN FROM INDIRECT FILE.  WANT EXAMPLE (Y,N) ?
```

If the answer is a Y the routine then prints the startup message as shown in Figure 2.

After the startup message the routine calculates the address of the second buffer by adding half of the available memory to the first buffer address. Care is taken to keep the address at a word boundary.

The first data line is read in. Using the digitizing rate and the number of channels, a record size is calculated such that it contains more than 100 milliseconds of data and the size is about 4000 bytes and less than the buffer size. For 1000 samples per second and 13 channels of data the size comes out as 4026 bytes and 143 milliseconds. Each record contains an 11 word header and each sampling interval contains the 16-bit BCD time from the time code reader at the start of this sampling interval in addition to the digitized data.

Various other quantities are calculated such as sample intervals per record and time per record. The number of .1 millisecond ticks per sampling interval is put into the clock buffer. When the clock is started, an interrupt will occur after that many ticks. Next, the comment or ID lines are read into a temporary array. The array will be part of the file header.

Finally, the event time lines are read and put into four arrays for the digitizing part of the code. Array 1 contains the ASCII day, hour, minute, and second for each event to search for. This ASCII time is put into the header of each event of the data output tape. Array 2 contains the BCD time to look for, converted from the times in Array 1. Array 3 contains the number of records to form and write for each event. This number is formed from the calculated record size and the number of seconds desired for an event. When digitizing, it is this number which is used to determine the end of an event rather than actually counting the seconds. Array 4 contains the pause flag: 0 for no pause or 1 to pause.

Once the input file has been completely read, a 177777 octal is placed in the BCD array after the last time. The SETUP routine then calls a tape initializing routine. This routine brings in the hardware tape driver and asks if the tape is a new or old tape. If it is an old tape, the routine will search for a double end-of-file and position the tape between the end-of-files; otherwise, it will rewind the tape. Then, it will write out the file header record. Next, the SETUP routine prints "READY TO DIGITIZE. START ANALOG. ENTER LETTER,CR ?" and now we return to the main program. A flow diagram of the SETUP routine is shown in Figure 3.

The rest of the code consists of 4 nested loops as follows:

```
Loop 1: innermost - digitize all channels.
Loop 2: next      - to number of sample intervals
                    in a record.
Loop 3: next      - to number of records in an
                    event.
Loop 4: outermost - to events until BCD time word
                    is 177777 octal.
```

At the start of loop 4 the code continuously reads the 16 bit parallel interface board for the BCD analog tape time and compares it with the BCD event time. When a match occurs the clock, loops 1, 2, and 3 and the digitizer are started and the clock interrupt flag check is skipped. Thereafter, before starting loop 1, the clock interrupt flag is checked. If it is not, then the code is too slow for this digitizing rate and the code stops; otherwise, the code waits for the clock

```
USING AN EDITOR BUILD A FILE, FILENAME.COM AS FOLLOWS:
1."RUN FASDIG"  (DONOT INCLUDE ")
2.NUMBER CHANNELS,NUMBER .1MILLISEC/SAMPLE,NUMERATOR,DENOMINATOR OF RATIO
    WHERE RATIO=PLAYBACK SPEED TO RECORD SPEED
3.UP TO 17 LINES OF ID (60 CHAR EACH) IF < 17 LINES, END WITH CR ONLY LINE.
    LINE 1=GENERAL INFO, THEN ONE/CHAN WHERE-1ST NONBLANK CHAR
    AFTER = SIGN IS CHAN ID
4.UP TO 100 PICK TIME LINES (DAY,HR,MIN,SEC,# SEC TO KEEP, P OR NOTHING)
    NOTE: P IS FOR PAUSE AT END OF THIS EVENT
    NOTE: SEC TO KEEP ARE WALL CLOCK SECONDS.
5.END FILE WITH A CR ONLY LINE

TO RUN CODE - ENTER "@FILENAME" (DO NOT INCLUDE ")

DO YOU WANT TO CONTINUE (Y,N) ?
```

Figure 2. Startup Message

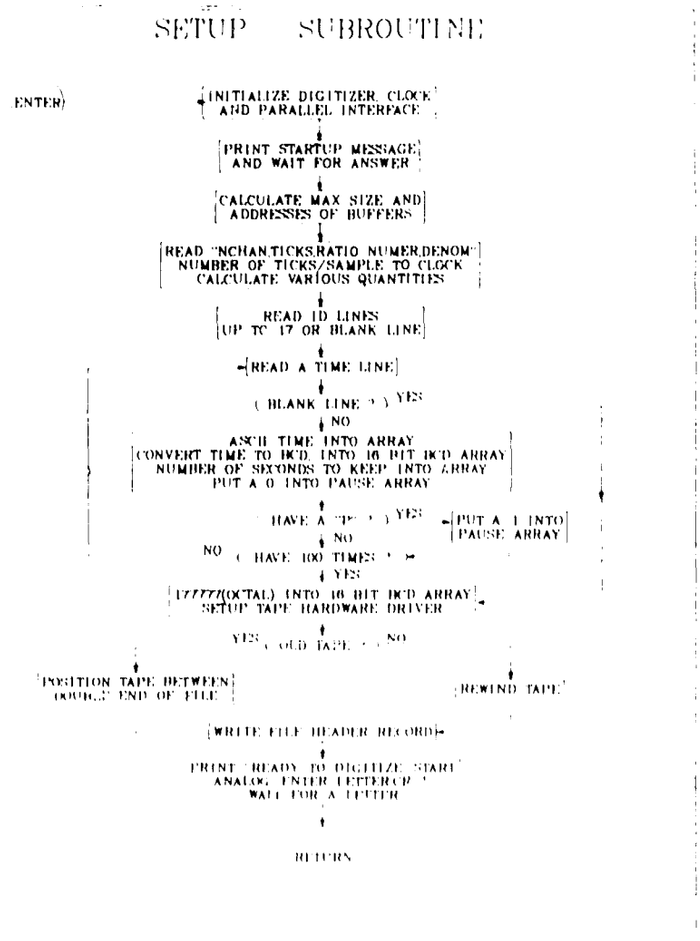

Figure 3. Flow Diagram of SETUP.

interrupt and then starts loop 1. After loop 1 finishes a check is made to determine if the buffer (record) full flag is set. If so, a write of buffer 2 to tape is started and the flag cleared. Loop 2 continues until the correct number of sampling intervals to make a record are in the buffer. After loop 2 finishes the buffer addresses are switched and the buffer full flag is set. Loop 3 continues until the desired number of records have been digitized. After the end of loop 3 the code stops the clock, writes the last record to tape, writes an end-of-file, and backspaces over the end-of-file. Next the code checks the pause flag. If the flag is set, the comment "AM IN PAUSE. ENTER LETTER,CR TO CONTINUE." is printed and the code waits for input from the terminal. If the flag is not set or input has been received, then loop 4 continues until an event time equal to 177777 (octal) is found which ends loop 4. The code does a little clean up and finishes. Figure 4 shows a flow diagram of FASDIG.

The clock interrupt handling routine decrements the clock interrupt flag, clears the interrupt bit, and returns from interrupt. The flag is reset by setting it to a 1. If the flag is ever negative, then more than one interrupt has occurred since the last time the flag was reset. Although not mentioned previously, the clock interrupt flag is checked for +, 0, and -.

MORE PROBLEMS

After putting the above code together and debugging it, I started to run the real case of 1000 sampling intervals per second (i.e. 10 ticks or ten .1 milliseconds per sampling interval). Lo and behold I got the message "CODE IS TOO SLOW." I tried several other values and found the code would digitize at 20 ticks but not at 18 ticks. I studied the listing of the code and squeezed out every unnecessary instruction I could find in the inner loops. I also rearranged the order of some instructions. I did everything imaginable to speed up the code, but to no avail.

At that time I was using the .WRITE programmed request to write the data out to tape. Since nothing else I had done speeded up the code, I decided to look at the tape controller manual to find out how easy it would be to control the tape unit directly instead of using the .WRITE programmed request. I found that the tape controller had several registers and only the command,

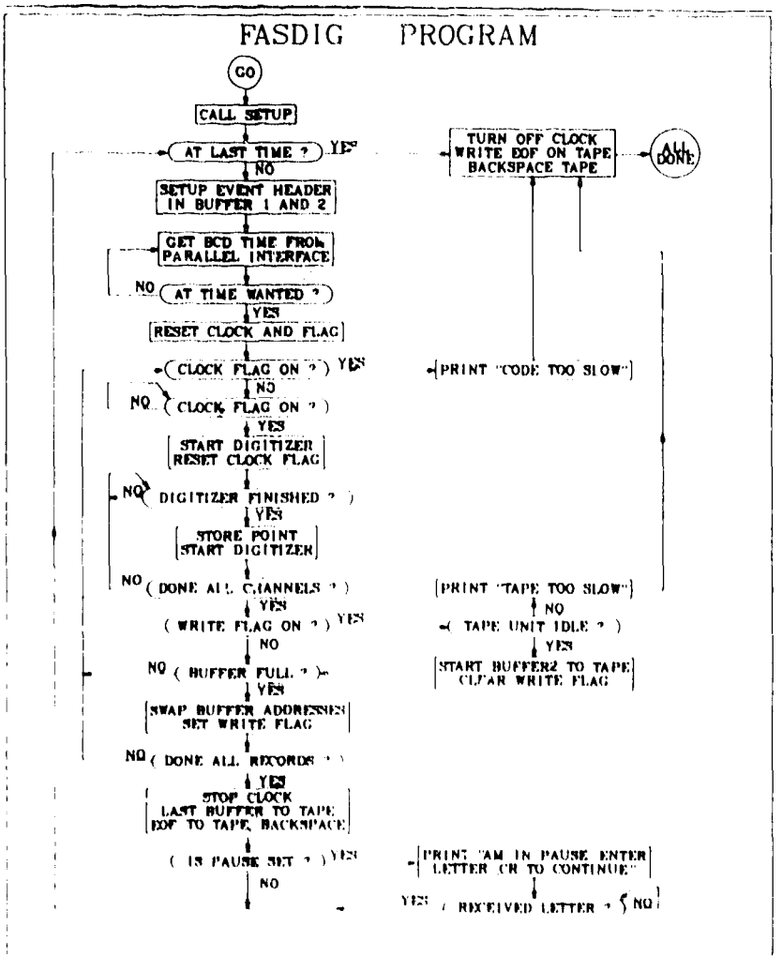

Figure 4. Flow Diagram of FASDIG.

address, and buffer count registers needed to be accessed in order to check activity on the tape drive and to have the controller start writing data to the tape. It seemed simple enough, so I replaced the following piece of coding

```
        .WRITE   #AREA,#1,@#BUF2,@#WDPREC
        BCC      57$
        .PRINT   #WFAIL
        JMP      EXIT
57$:    Reset write flag
```

with

```
        TSTB     @#MTCMD
        BMI      57$
        .PRINT   #NOCTRL
        JMP      EXIT
57$:    MOV      @#BUF2,@#MTADR
        MOV      @#NEGBYT,@#MTBRC
        MOVB     #5,@#MTCMD
        Reset write flag
```

where BUF2 contains the address of the data buffer to write out, WDPREC is the number of words to write out, NEGBYT is the negative number of bytes to write out, and MTCMD, MTADR, and MTBRC are the magnetic tape controller registers: command, address, and buffer count. WFAIL and NOCTRL point to two error comments. Thus, the .WRITE programmed request is replaced by three MOVE commands: two to tell what data to use and one to signal the controller to start writing. The error checking is slightly different in the two cases; in the old case the carry bit was set to denote that the .WRITE was unable to queue the write request, in the new case the byte sign bit of the command register is cleared when the controller is busy. In either case, if the carry is set or the sign bit is cleared, the tape can not be written and the code is stopped. With this new change, the code had no trouble digitising at a total rate of 13,000 samples per second and writing the [illegible] byte tape records every 143 milliseconds. Figure 5 shows a sample of the data that was digitized.

CONCLUSION

In the above project, I almost didn't try any method other than the .WRITE programmed request. I was worried about the conflicts that might occur with the system and the amount of effort [illegible] the system would take. However, I [illegible] method I used did work with very little effort. My first conclusion is, therefore, don't give up before you try.

Secondly, I want to point out that the ease with which this change was made was largely do to running under the RT-11 operating system. I have tried to do similar projects under PSX-11M and usually have to spend months reading up on what data bases and packets must be built to accomplish what I want. Even then I would guess that the system will have a few "gotchas" that I overlooked. The simplicity of running under RT-11 is wonderful.