

Fermi National Accelerator Laboratory

FERMILAB-Conf-91/316

Software Development Tools for the CDF MX Scanner

W. Stuermer, K. Turner and S. Littleton-Sestini

*Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510*

November 1991

* Presented at the *IEEE Nuclear Science Symposium*, Santa Fe, New Mexico, November 2-9, 1991.



Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

SOFTWARE DEVELOPMENT TOOLS FOR THE CDF MX SCANNER

W. Stuermer, K. J. Turner, S. E. Littleton-Sestini,

Fermi National Accelerator Laboratory¹

Abstract

This paper discusses the design of the high level assembler and diagnostic control program developed for the MX, a high speed, custom designed computer used in the CDF data acquisition system at Fermilab. These programs provide a friendly, productive environment for the development of software on the MX. Details of their implementation and special features, and some of the lessons learned during their development are included.

I. INTRODUCTION

The CDF experiment at Fermilab uses a programmable processor called the MX in its data acquisition system. Because of the custom design of the MX, off the shelf software could not be used for program development, software debugging or hardware diagnostic functions. Writing these utilities afforded an opportunity to incorporate some new and interesting features, as well as learning the degree of effort required to develop custom software to support a machine like the MX.

A. MX Architecture

The design of the MX has several unique or unusual features. These features are of interest here because they are reflected in the design of the tools developed to support the MX. As shown in figure 1 the MX has 5 internal memories in addition to the instruction memory (IM). The Ewe Memory (UM) is used to store a list of specially encoded words which tell the ADC card (called a "Ewe") how to convert analog channels and to read (digital) PROM data. There are three independent data memories called DMA, DMB and DMC, whose data is used to perform arithmetic corrections on the ADC data and such housekeeping functions as counting loops. The Event Memory (EM) is where the MX writes its output. Much of the power of the MX comes from the fact that it can fetch an operand from three different memories, perform a three way operation such as $A+B*C$ and store the result in a

fourth memory, in a single 120ns instruction.

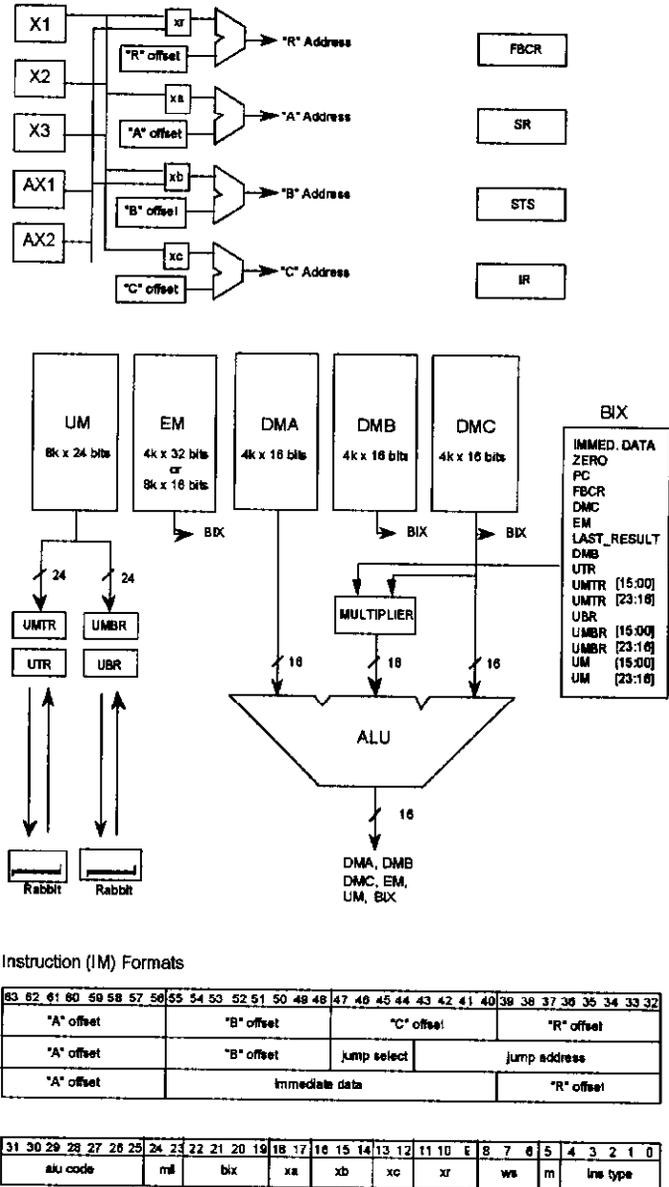


Figure 1 - Architecture of the MX Computer

¹Fermilab is operated by the University Research Association, inc. under contract with the U.S. Department of Energy. This work was supported by the U.S. Department of Energy under contract No. DE-AC02-76CHO3000.

The MX also has a set of 13 registers, but only the index registers will be discussed here. Memory addresses in the MX are formed by adding the contents of a 16-bit index register with an 8-bit offset encoded in the instruction word. The in-

struction format allows for 4 independent addresses (3 operands and 1 result), but there are some restrictions on which index registers can access a given memory. The "A" and "C" addresses are used to read operands from the DMA and DMC memories respectively, and can use the X1, X2 or X3 index registers. The "B" address is used to read an operand from the UM, EM, or DMB memories, and can use the full set of index registers (X1, X2, X3, AX1, AX2). The "R" address is used for writing an arithmetic result and can also use any index register. The result can be written to any memory except the IM.

B. Role of the MX in the CDF Data Acquisition System

Figure 2 shows a simplified view of the CDF data acquisition system. When the CDF experiment is running, there is a proton-antiproton "bunch crossing" every 3.5 μ s. Some of these crossings result in high-energy collisions. The result of a collision is called an "event". The purpose of the data acquisition system is to digitize, process and collect the signals coming from the electronic detectors used to instrument the CDF experiment, and write the data to tape for further analysis offline.

The trigger system uses special "fast out" signals from the front-end electronics to determine if the event is interesting. If the event satisfies the level 1 and level 2 triggers, a start scan message is broadcast to the MXs.

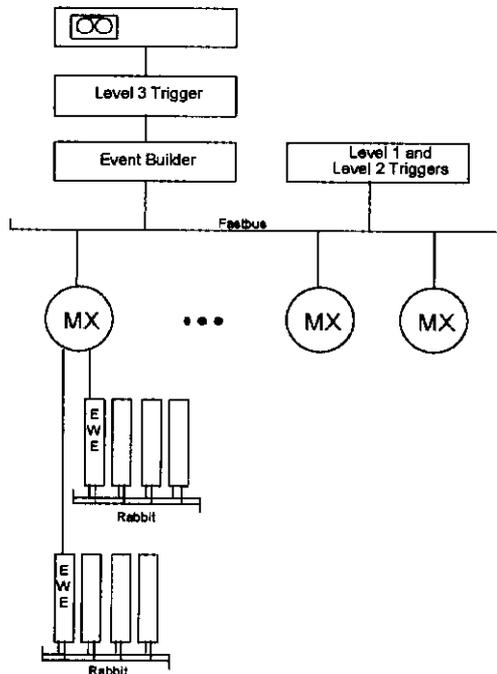


Figure 2 - Simplified Diagram of the CDF Data Acquisition System

The purpose of the MX is to digitize a set of analog channels, perform certain local processing on the data and format it

in the EM. It starts this process when a start scan message is written to its FBCR (Fastbus control register) register by a Fastbus broadcast, and signals completion by setting the DONE bit in the SR (status register). When all of the MXs and other "scanners" have finished, the Event Builder reads the data from each one and builds "event record". The "Level 3" trigger uses physics analysis programs to reconstruct the event and apply a final cut to which events are written to tape.

Each MX controls two ADC cards, called Ewes; and each Ewe converts approximately 500 analog channels coming from the CDF detectors. The MX tells each Ewe what to do by sending it specially encoded 24-bit words which are written to the UM at the same time that the MX program is downloaded to the IM. After starting each conversion, the MX polls the Ewe status until DONE goes high. Then, the digital data is read from the Ewe.

If the MX is running the data acquisition code, a pedestal value is subtracted from the data, it is multiplied by a linear correction, and it is compared to a threshold value. If the MX is running the pedestal calibration code, 256 values are collected for each channel. Each value has a base value subtracted (to prevent overflow) and is added to a 32 bit sum(x) and sum(x²). Later, these sums are used to calculate the mean and sigma for each channel pedestal.

The resulting data is written to the Event Memory in a format which is organized by detector component, and logical channel ID, called "scanner bank format."

Currently, there are 60 MXs used in the CDF data acquisition system and about one half of the data from the CDF detectors are processed by these MXs. The rest of the data is processed by other devices in the data acquisition system.

III. HIGH LEVEL ASSEMBLER

Arguably, a processor's assembly language is the most important interface between the hardware and the intended users. This was especially true in the case of the MX, since the development of a high level programming language such as Pascal or C was not anticipated. A custom assembler language incorporating features from high level languages such as do while loops, if/then/else structure, local storage and branch symbols has been designed and implemented to serve as a vehicle for software development on the MX.

There are several possible pitfalls in the design of an assembler for a device such as the MX. For example, an arithmetic expression for the MX may involve 4 operands including the result. Does LDAM A,B,C,D mean $A = B + C * D$, or $D = A + B * C$, or does it mean $A = B * C + D$? The MX assembler (called ASM/MX) borrows it's syntax for such arithmetic expressions from more familiar grammars like Fortran. It is hoped that the meaning of a statement in ASM/MX like $LOAD A = B + C * D$ require no explanation (the rules of precedence are the same as those in Fortran). In this example A, B, C and D could be any memory or register, limited only by the architecture of the MX.

A. Familiar Cast of Characters

ASM/MX refers to the MX registers by the same names as those found on the original schematics and engineering documentation used in the development of the MX (and shown in figure 1). This is important because a significant group of users of the assembler are the engineers and technicians who developed the MX and keep it running today. The same is true of the MX memories, except the DMA is referred to as "A", DMB as "B" and DMC as "C", for the sake of brevity. In this way, the memories and registers of the MX form a familiar cast of characters for those being introduced to MX programming for the first time.

B. Memory References

As stated above, MX memory references are formed by adding the contents of a 16-bit register to an 8 bit offset and such references may apply to the DMA,DMB,DMC,EM or UM memories. The MX assembler provides several distinct representations for such references, in order to facilitate the simplest and most easily understood structure for each use.

memory	syntax	
dma	A.name	A (constant)
dmb	B.name	B (constant)
dmc	C.name	C (constant)
em	EM.name	EM(constant)
um[23:16]	UM_HIGH.name	UM_HIGH(constant)
um[15:00]	UM_LOW.name	UM_LOW (constant)
dma	A.name (constant)	
dmb	B.name (constant)	
dmc	C.name (constant)	
em	EM.name(constant)	
um[23:16]	UM_HIGH.name(constant)	
um[15:00]	UM_LOW.name (constant)	

Figure 3 - Direct Addressing Mode in ASM/MX

memory	syntax	
dma	A (xa)	A.name (xa)
dmb	B (xb)	B.name (xb)
dmc	C (xc)	C.name (xc)
em	EM(xb)	EM.name(xb)
um[23:16]	UM_HIGH(xb)	UM_HIGH.name(xb)
um[15:00]	UM_LOW (xb)	UM_LOW.name (xb)
dma	A.name (xa + constant)	
dmb	B.name (xb + constant)	
dmc	C.name (xc + constant)	
em	EM.name(xb + constant)	
um[23:16]	UM_HIGH.name(xb + constant)	
um[15:00]	UM_LOW.name (xb + constant)	

Figure 4 - Indirect Addressing Mode in ASM/MX

For example, the first 256 locations in each of the memories are treated as a special "scratch memory" by MX programmers. The reason that this is so is that these locations can be accessed without the use of an unused index register, which can be a scarce commodity. In ASM/MX, the programmer can assign symbolic names, like a.module_no, to one or more consecutive memory locations. These names can be used alone like a 16 bit integer in Fortran, or in combination with an index register, like an array of 16 bit integers. In some situations, it is convenient to specify an explicit offset in addition to the offset implied by the symbolic name. The syntax for direct (no index register) and indirect (w/ index register) are shown in figures 3 and 4, respectively. Notice that all symbolic names begin with the name of the memory in which it resides.

C. Program Structure

An ASM/MX program consists of a list of functions and "storage sections." A storage section declares a related set of symbols like those just described and fulfill much the same purpose as common blocks in Fortran. One storage section may be designated as the "GLOBAL STORAGE" section. Storage symbols declared in this storage section are automatically imported by every function. Every other storage section is given a name and functions that need to refer to the storage declared there need to import them by referring to the name of the storage section. Storage sections differ from Fortran common blocks in that the symbols (variables) are only declared in one place and it is impossible to accidentally overlay symbols have different names or associated storage.

syntax	example
GLOBAL STORAGE	STORAGE constants
<storage_declaration>	a.const : 8 words = 0,1,2,3,4,5,6,7
<storage_declaration>	a.const_stop : 1 word = 128
• • •	a.const_rd : 1 word = 64
END	b.hex_FFFF : 1 word = (hex) FFFF
	b.hex_8000 : 1 word = (hex) 8000
STORAGE <section_name>	b.flag_error : 1 word = 1
<storage_declaration>	c.const_16 : 1 word = 16
<storage_declaration>	c.const_32 : 1 word = 32
• • •	c.hex_4000 : 1 word = (hex) 4000
END	END

Figure 5 - Storage Sections in ASM/MX

Functions are the basic executable unit of an MX program. One function is designated the "MAIN FUNCTION" and serves as the primary entry point for the program. An ASM/MX function is divided into two parts. The first is used to declare local symbols for storage and importing storage symbols from storage sections. The second part contains all of the executable code in the function.

syntax	example
<pre> MAIN FUNCTION IMPORT STORAGE FROM <section_name>, <section_name>, . . . STORAGE <storage_declaration> <storage_declaration> . . . BEGIN <executable_statement> <executable_statement> . . . END FUNCTION <function_name> IMPORT STORAGE FROM <section_name>, <section_name>, . . . STORAGE <storage_declaration> <storage_declaration> . . . BEGIN <executable_statement> <executable_statement> . . . END </pre>	<pre> MAIN FUNCTION import storage from constants storage c.stack : 20 words c.top_stack : 0 words BEGIN load SR = a.sr.done load X1 = addr(c.top_stack) /* Main event loop. */ %loop /* Wait for a start-scan. */ %until ss /* Wait. */ %end until /* Save the start scan message and the EOS bit in the status register. */ load a.fbr = fbr load sr = a.sr.start /* Find out what type of start-scan. */ eval a.fbr and (hex) 0C00 load X2 = last_result (* high) c.const_64 %case X2 of 0: call datsa 1: /* do nothing */ 2: call inita call set_chadrs 3: /* do nothing */ %end case /* Set EOS (end-of-scan) bit in the status register. */ load sr = sr or a.sr.done %end loop END </pre>

Figure 6 - Functions in ASM/MX

This separation between declarative and executable statements is unusual for an assembly language, but introduces an element of predictability for the programmer. Like any reasonable programming language, ASM/MX will flag the use of any storage symbol which has not been explicitly declared as a syntax error. This behavior helps prevent simple spelling errors from introducing bugs in the program. By putting all of the storage declarations at the beginning of the function, the programmer knows where to look to see if a storage symbol has already been defined and, if so, how it is spelled.

Transfer of control within an ASM/MX function is accomplished by the "GOTO" statement. Every GOTO statement must have a branch label as its target. Branch labels look like those found in PASCAL and consist of an alphabetic character, one or more alphanumeric characters and a colon. All branch labels are local to the function in which they are declared, so jumping into the middle of another function is not possible. This feature is intended to discourage the unstructured, "spaghetti code" often found in assembly language programs. Branch labels are not required at all, in most cases, because of the availability of structured, "meta-statements." (Please refer to section D, Meta-Statements.)

Transfer of program control from one function to another is done with a conventional "CALL" statement that enters the function at the top, and "RETURN" statement which causes

the program to resume execution where it left off in the original function. The only provision for passing arguments to the called function or returning results is by the use of storage symbols which are shared by both functions. This is consistent with the requirement for MX programs to operate as fast as possible.

syntax	examples
<pre> %WHILE <condition> <executable_statement> <executable_statement> . . . %END WHILE </pre>	<pre> /* Load ewe for next channel. */ %while a.const_xqt and um_high(ax1) = 0 load umbr = um(ax1), ax1++, strobe %end while </pre>
<pre> %LOOP <executable_statement> <executable_statement> . . . %END LOOP </pre>	<pre> /* Free running counter. */ %loop load a.count = a.count - 1 %end loop </pre>
<pre> %UNTIL <condition> <executable_statement> <executable_statement> . . . %END UNTIL </pre>	<pre> /* Wait for the ewe to finish with this channel. */ %until done(bottom) load ubr %end until </pre>
<pre> %CASE <index_reg> OF 0: <executable_statement> <executable_statement> . . . 1: <executable_statement> <executable_statement> . . . n-1: <executable_statement> <executable_statement> . . . %END CASE </pre>	<pre> /* ----- 3 cases: o Both top and bottom lists are empty. o Only top list is empty. o Only bottom list is empty ----- */ load x2 = c.state %case x2 of 0: goto end_loop 1: call state_a 2: call state_b 3: call state_c %end case </pre>
<pre> %IF <condition> THEN <executable_statement> <executable_statement> . . . %ELSE IF <condition> THEN <executable_statement> <executable_statement> . . . %ELSE IF <condition> THEN <executable_statement> <executable_statement> . . . %END IF </pre>	<pre> /* Move to an even word boundary. */ eval x3 and a.const(1) %if last_result != 0 then load em(x3) = 0, x3++ %end if </pre>

Figure 7 - Meta Statements in ASM/MX

D. Meta-statements

One of the most important goals of ASM/MX programming is for the resulting program to execute as efficiently as possible. This is because, in the case of the data acquisition code, the execution time of the MX adds directly to the "front-end deadtime" of the system. During this time, the data acquisition system is blind to the proton-antiproton collisions that are occurring every 3.5μs. Since operation the accelerator

at Fermilab costs many thousands of dollars per hour, these lost collisions have a significant value.

For this reason, ASM/MX maintains a one-to-one relationship between executable statements and generated machine instructions. This gives the programmer the control he/she needs to write the fastest possible program within the available instruction set.

There are certain constructs of these atomic statements, however, which are repeated time and again in MX programs. These constructs correspond to the classical elements of structured programming, including 3 types of loop, chained if/then/else and the case statement. Special "meta-statements" have been added to ASM/MX which implement these constructs by generating several MX machine instructions. Each of these constructs begin with a keyword with a percent sign, like %WHILE, in order to distinguish between meta-statements and the ordinary, atomic statements. The syntax of the available meta-statements is listed in figure 7.

IV. MX ONLINE EXECUTIVE INTERFACE

The MX functions as an embedded processor, in that it has no controls or displays on its front panel and has no direct terminal connection. All communication between it and the outside world is done over the Fastbus local area network. A program has been written to serve as a combination control panel, software development debugger and hardware diagnostic utility. This program is called the MX Online Executive Interface, or "Moxi." Moxi implements several features which may be useful for applications written to support custom processors besides the MX.

MAIN MENU		STATUS	
1 [DO]	Execute command script.	MOXI VERSION	8.14
2 [SET]	Set device and MOXI paramet...	ulpeak:	6.00
3 [RESET]	Reset MOXI parameters to de...	mx scanner:	UNKNOWN
4 [SHOW]	Show MOXI parameters.	address:	00000003
5 [EDIT]	Edit file with LSE or EDT.	ewe:	NONE
6 [HELP]	Request help on some command.		
7 [DCL]	Execute VMS command(s).	log file:	DISABLED
8 [QUIT]	Exit MOXI.	mask mode:	ENABLED
9 [READ]	Read from MX or other devices	spy mode:	DISABLED
10 [DECODE]	Read register and interpret.	verify mode:	ENABLED
11 [WRITE]	Write to MX or other devices.	step mode:	DISABLED
12 [MODIFY]	Modify MX or EWE register.		
Press RETURN for next Page			
MOXI >			

Figure 8 - Menu and Status Viewports in Moxi

A. User Interface Issues

Every useful program has a user interface and Moxi is no exception. A significant effort has been devoted to developing a consistent, powerful and easy to use tool.

Moxi features a "dual user interface" and can be operated as a menu driven or command driven program. When Moxi starts up, it presents the user with a menu viewport, a status

viewport and a workspace. The menu viewport gives the user a list of verbs, the user types one and Moxi presents the user with a different menu, until there are no more options and the command is executed. This mode can be useful for people who are just becoming familiar with Moxi's commands, but leaves a relatively small part of the screen available for the workspace, where the user types in a command and reviews the response. As the user becomes more proficient in using Moxi, the user can dismiss the menu viewport with the "SET VIEWPORT/NOMENU" command. This makes Moxi more responsive because it no longer has to change the menu, and leaves more room to review the results of a command which dumps more than a few lines of data, and to review the results of more than one command. This can be quite useful when tracking down a bug in hardware or software.

A verb-noun syntax is used to organize the commands in Moxi and make them easier to remember. There are currently over 100 commands in Moxi. Rather than attempt to assign a unique verb to each one, Moxi uses a Verb-Noun combination wherever possible. Because the same verbs and nouns are used in many commands, the user only has to remember M+N items rather than M*N items. For example, the DECODE verb can be applied to the command register in the Ewe (called EWE/COMMAND), or the FBCR ("Fastbus Control Register"), or a word in the instruction memory. Similarly, the noun FBCR can be used in the READ and WRITE, as well as the DECODE, commands.

CONTINUE	MODIFY FBCR	RESET TRAP	SHOW LOGFILE
DCL	MODIFY IM	RESTORE	SHOWMODE
DECODE EWE	MODIFY memory	RUN	SHOWMX
DECODE EWE/register	MODIFY PC	SAVE	SHOW TRACE
DECODE IM	MODIFY umxr	SCAN/D_MODE	SHOW TRAP
DECODE MEP	QUIT	SCANX_MODE	SHOW VERSION
DECODE MEP/register	READ BAT	SCANDATA	SPAWN
DECODE register	READ CARD	SCANINIT	STEP
DISABLE CS	READ CSR_SPACE	SCANTERM	STOP
DISPLAY	READ Composite	SET BREAK	TEST EWE
DO	READ DATA_SPACE	SET DISPLAY	TEST EWE/AUTODEC
EDIT	READ EWE	SET EDITOR	TEST EWE/REGISTER
EXIT	READ EWE/register	SET EWE	TEST memory
HELP	READ EVENT	SET LOGFILE	TEST MX
HISTOGRAM ADC	READ Memory	SET LOOP	TEST register
HISTOGRAM EM	READ MX	SET MODE/NAACK	VERIFY
INIT BAT	READ Register	SET MODE/SPY	WAIT
INIT EWE	READ TMXD	SET MODE/STEP	WRITE BAT
INIT MX	RESET BREAK	SET MODE/VERIFY	WRITE Composite
LOAD	RESET DISPLAY	SET MX	WRITE EWE
LOOP	RESET EWE	SET TRACE	WRITE Memory
MAP	RESET LOGFILE	SET VIEWPORT	WRITE MEP
MODIFY EWE/CHADRS	RESET LOOP	SET EWE	WRITE MX
MODIFY EWE/COMMAND	RESET MX	SHOW BREAK	WRITE Register
MODIFY EWE/PEDESTAL	RESET TRACE	SHOW EDITOR	
MODIFY EWE/WRITE		SHOW EWE	

Figure 9 - Summary of Commands in Moxi

Another principle in Moxi's design is to avoid "command modes", where a command is only available in some restricted context, or the same commands have different meaning in different modes. This is a common problem with menu driven programs, which can make them tedious to use. All of Moxi's commands are always available, unless they require a device which has not been selected, and the user can escape any input prompt (for the WRITE PC command, for example) by entering CNTRL-Z.

Positive feedback is used to let the user know what state Moxi is in and to put the data displayed in the workspace in a meaningful context. The status viewport tells the user what

version of Moxi is currently executing, what devices (such as the MX) have been selected, and what modes (ie, is there an open log file, is Moxi currently reporting NACK errors) are active.

The standard names are used for the memories and registers in the MX, PC and DMA, for example. Registers located in other devices, such as the Mep, Ewe, and Bat, use names like MEP/SCID and EWE/COMMAND.

Moxi provides quick INIT commands for the Mep, Ewe, Bat and MX; and quick TEST commands for the MX and Ewe. For example, TEST EWE reads out the Ewe's ID prom, reads certain reference voltages on the Ewe, performs a "barber pole" bit test for all the registers on the Ewe, and tests the Ewe's "autodec" feature of the channel address register. The INIT command sets the specified device to some predefined state, then reports to the user what state that is.

The Mep, Ewe, Bat and MX can also be used as the target for the READ and DECODE command. For example, READ EWE displays the contents of all of the registers of the currently selected Ewe. This is a lot easier than typing 7 commands to read these registers out one at a time.

B. General Facilities

The READ command provides a mechanism for displaying the contents of any register or memory in the MX, Mep or Ewe. The output of the READ command can be directed to a file with the /OUTPUT qualifier. The DECODE command provides roughly the same function as READ, except the target of the command (a memory, register, or device) decoded into fields, like the module and submodule addresses in the EWE/CHADRS register; or disassembled into the ASM/MX syntax, in the case of the IR (instruction register) and IM (instruction memory).

Similarly, the WRITE command provides a mechanism to change the value of any writable memory or register in the MX, Mep or Ewe. (Some registers are read-only.) Moxi will perform a readback and verify operation if "verify mode" is enabled. The MODIFY command allows the user to modify one or more fields in a register by performing read-modify-write operation. Naturally, the register fields are specified by name.

Even with the menu and status viewports disabled, the user only has 24 lines for the workspace on a standard ASCII terminal. Moxi allows the user to get around this by opening up a log file. Everything written to the workspace: command prompts, user responses, data, and any error messages, are written to the log file. This can be particularly useful in two situations. In the first, someone is running an overnight hardware test and wants to be able to review the results even if the terminal is accidentally turned off. The second situation is where the user is executing a program trace, where the output may potentially be thousands of lines long.

C. Software Facilities

The basic commands for controlling the execution of an MX program are RUN, STOP, and STEP. As the name implies, the STEP command uses the single-instruction feature of the MX to execute one instruction. It can only be used when the MX clock is in a halted state. The RUN command either releases the MX clock, or causes the MX program to execute in a series of single-instruction steps. This is controlled by the state of "step mode" in Moxi.

Breakpoints are IM addresses defined by the user and kept on a list by Moxi. The PC is read and checked against this list whenever the user executes the STEP command, or after every instruction is executed in step mode. If the current PC is a breakpoint, program execution is stopped, a list of read/write commands called the display list is executed, and Moxi returns control to the user via the command prompt. Because breakpoints require reading the PC after every instruction is executed, they are inactive when the MX clock is running (non-step mode).

Trappoints are user defined addresses where a special jump-to-self instruction has been written to the IM by Moxi. Trappoints behave like breakpoints, except they will stop program execution regardless of whether Moxi is in step mode or non-step mode. Because the PC is not read out after every instruction is executed, the display list is not automatically executed, but the user can use the WAIT command to poll the PC and execute the display list when the MX program reaches one of the trappoints. When the user sets a trappoint, Moxi reads the original contents of specified address, so the trappoint can be reset and the original instruction restored just as easily as it was set.

A special "CONTINUE" command is provided in Moxi to allow the user to "step through" a trappoint. When the user enters this command, Moxi first stops the MX clock and checks that the current PC is on the trappoint list. Assuming we are at a trappoint, Moxi temporarily restores the original instruction and executes a single step. The breakpoint is restored and the clock restarted.

Moxi allows the user to build a "display list" consisting of a list of read/write commands. The display list can be executed explicitly, with the DISPLAY command, or implicitly, after one instruction has been executed with the STEP command, or when the MX program hits a user defined breakpoint.

Rather than defining a set of commands to add and remove operations from the display list, Moxi uses the LSE or EDT editor on the host machine (a Vax) for the user interface. When the user enters "SET DISPLAY", Moxi opens a temporary file using the editor's callable interface. Once the user is in the editor, he/she can use the familiar keypad commands to insert, remove or modify read/write operations. The same syntax is used as the READ and WRITE commands. When the user is satisfied, the editor is exited in the normal way and control returns to Moxi. Moxi opens the temporary file, parses its contents and finally deletes it. This method provides the user with a familiar and flexible interface, and may be of use in other control programs like Moxi.

Alternatively, if the user specifies a file name with the SET DISPLAY command, Moxi will look for an existing file and

build the display list with the commands in this file. In this case, the file is not deleted. This allows the user to build a set of predefined display lists. Towards this end, Moxi first looks for the display list in the current directory, then in a special directory defined by a logical name.

D. Hardware Facilities

Facilitating the job of tracking down and repairing MX failures is an important function in Moxi. One tool Moxi provides for this is the "scope loop." A scope loop is a list of read and write commands which are executed to generate some signal in the MX hardware. A oscilloscope or logic analyzer can then be used to look at the behavior of the MX's logic under this stimulus.

Moxi uses the same user interface for setting up a scope loop and setting up a display list. When the user enters the "SET DISPLAY" command, he/she is put into the LSE or EDT editor (this is a user defined option), where the desired read and write commands can be added, removed, or modified.

When the user starts the scope loop (via the "LOOP" command), the commands on the scope loop are executed again and again, until the user presses CNTRL-C. Unlike the display list, the read commands on the scope loop do not display any data. This is in order to maximize the speed with which the loop executes

Another command which is useful in maintaining the MX hardware is the "TEST" command. The TEST command works with any of the memories and registers in the MX; or any of the registers in the Ewe; or all of the registers and memories in the MX; or all of the registers in the Ewe.

Which test is performed depends upon the memory or register under examination. For example, the registers in the Ewe are checked using a barber-pole pattern that looks like 1, 10, 100, 1000.... First, the pattern is written to all of the registers, then the registers are read back and compared with the original data. Then the whole pattern is rotated with a circular shift operation and it is written to the registers again. This process only requires 16 I/O operations on the Vax and since a different value is written to each register, problems with the register decoding in the Ewe are checked at the same time.

Although some hardware failures may show themselves every time it is tested, others happen only intermittently. The OVERNIGHT command repeats one or more tests over and over again in a loop until the user terminates the test by pressing CNTRL-C. The overnight command accepts the same targets as TEST and, in fact, calls the same test functions.

V. CONCLUSIONS

The ASM/MX assembler and Moxi are sophisticated programs which provide a good environment for software development and hardware maintenance. These programs demonstrate certain features which may be applicable in the design of programs of related function. ASM/MX demonstrates that an assembler featuring good, context dependent error messages and combining some of the best features from high level and

assembly level languages can be implemented using a simple, recursive-descent approach.

Moxi demonstrates the concept of providing the user with complementary user interfaces, in this case, menu driven and command driven interfaces; it shows one way a consistent command language can be designed, by adopting a verb-noun framework; and uses an interesting user interface for creating a command list, by using the callable interface to an editor..

But these capabilities were not achieved without cost. ASM/MX and Moxi are large programs, with almost 12k and 53k lines of code respectively, and took approximately 6 programmer-years to develop. Off the shelf software could not be used because of the custom design of the MX, so this cost may be viewed as a consequence of choosing a custom design for the MX. This investment was justified in the case of CDF since there were no processors available in 1981, when the CDF data acquisition system was designed, to match the MX's capabilities.

Developing custom software will always be a costly proposition, and must be taken into account when custom designed hardware is being considered as part of a system's design.

VI. ACKNOWLEDGEMENTS

We wish to gratefully acknowledge the efforts of Greg Schuweiler, who is currently maintaining Moxi; and of Steve Hahn and Daniel Frei, who have made major contributions to Amanda, the MX data acquisition and calibration code.

VII. REFERENCES

- [1] F. Abe, et al., "The CDF Detector: An Overview," *Nuclear Instrumentation and Methods*, vol A271, pp. 387-403, 1988.
- [2] G. Drake, et al., "The RABBIT System," *Nuclear Instrumentation and Methods*, vol A269, pp. 68-81, 1988.
- [3] T. F. Droege, I. Gaines, and K. J. Turner, "The M7- A High Speed Digital Processor For Second Level Trigger Selections," *IEEE Transactions on Nuclear Science*, Vol. NS-25. The name of the MX was derived from the Magnificent Multi-Muon Mass and Momentum Monitoring Machine, or M7. M7 was promoted to M8, translated to M10 (octal), and transliterated to MX (Roman numerals).
- [4] A.J.T. Davie and R. Morrison, *Recursive Descent Compiling*, New York: John Wiley and Sons