

Fermi National Accelerator Laboratory

FERMILAB-Conf-91/320

System Software Design for the CDF Silicon Vertex Detector

S. Tkaczyk

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

M. Bailey

*Purdue University
West Lafayette, Indiana 47907*

November 1991

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



Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

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.

System Software Design for the CDF Silicon Vertex Detector

S. M. Tkaczyk

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

M. W. Bailey

Purdue University, West Lafayette, IN 47907

Abstract

An automated system for testing and performance evaluation of the CDF Silicon Vertex Detector (SVX) data acquisition electronics is described. The SVX data acquisition chain includes the Fastbus Sequencer and the Rabbit Crate Controller and Digitizers. The Sequencer is a programmable device for which we developed a high level assembly language. Diagnostic, calibration and data acquisition programs have been developed. A distributed software package was developed in order to operate the modules. The package includes programs written in assembly and Fortran languages that are executed concurrently on the SVX Sequencer modules and either a microvax or an SSP. Test software was included to assist technical personnel during the production and maintenance of the modules. Details of the design of different components of the package are reported.

I. INTRODUCTION

The development of the software package and assembly language for the tests and operation of the Silicon Vertex Detector readout electronics components being commissioned for the CDF92 run will be discussed. The basic components of the SVX DAQ are a Fastbus Sequencer, a Rabbit Crate Controller and a Digitizer module. The function and arrangement of these modules will be briefly summarized here, and are described in more detail in other contributions to this conference[1,2].

The SVX detector contains over 46,000 readout channels divided into 24 sections called wedges. Each wedge is connected to a separate Digitizer, which resides in a Rabbit crate mounted on the CDF detector. Four Rabbit crates hold 6 Digitizers each. The Digitizer is responsible for driving and receiving analog and digital signals to the detector. The digital signals it sends are used to control the operation of switches on the custom designed VLSI chips called SVX Rev D integrated circuit[3](SVXIC) that are connected to the detector. The analog signals it sends are used to inject charge into the detector. The analog signal it receives is the pulse height of a charged particle traversing the detector, which is sent with a digital description of the location of the channel that was hit. In each of the four Rabbit crates there is one Controller, an interface between the Digitizers and the Sequencer, which

resides in a Fastbus crate in the Control Room. There is one Sequencer connected to each Controller.

The Sequencer contains a 16k data memory and a 16k program memory, along with a microsequencer chip, conditional control logic, and a 100 MHz clock. It sends data, function codes, and pattern timing signals through cables that are directly connected to the Controller. The Sequencer's data space is divided into 8 blocks, 6 of which correspond directly to the Digitizers in the crate. In the current configuration, the seventh block is not used, and the eighth block holds data from the Controller. In other possible configurations, the seventh and eighth blocks could be dedicated to two more Digitizers placed in the Rabbit crate. Each of these blocks has associated with it an End Of Buffer(EOB) register which keeps track of the last location in its block to which data has been written. The Controller drives the pattern timing signals onto the backplane of the Rabbit crate, where they are picked up by the Digitizers and driven to the wedges. If a function code is present, the Controller interprets it as a write or read of one of its own registers, of a register in a Digitizer in a designated slot, or of a register in all the Digitizers in the crate, performs the appropriate function, and sends the data back to the Sequencer. Three of the top five bits of the data indicate the slot location of the module from which the data came. The Sequencer uses these bits to direct the data to one of its eight data blocks.

II. CONTROL AND TESTING OF DAQ MODULES

A. Operation and Control Utilities

The degree of complexity of the three modules residing on different backplane buses made it necessary to design a software package which made common operations easily accessible, and would also automate the testing procedures as much as possible. The package, called SVX_Control, is written in Fortran, since we wanted to use the CDF Fastbus subroutines that were programmed in Fortran. We use the Uipack user interface package[4] to make the program menu-driven, so that technical personnel and the designers of the devices can quickly learn to use the program.

The top menu allows control of all system functions necessary during normal runtime operation. The most straightforward operation is writing and reading of the Sequencer's CSR and data registers; therefore we provide options that allow single writes and reads to CSR registers and block writes and reads to the data registers, since this is how these registers are normally accessed. In all write operations,

* Operated by the Universities Research Association under contract with the US Department of Energy..

a readback is also performed and any mismatches are reported. One can also access the Controller and Digitizer registers via addresses mapped into the Sequencer's CSR space but the procedure is different. In this case the desired function code is stored in the Front End Crate control Register in the Sequencer. Then a Fastbus write is made to the CSR address corresponding to the Digitizer or Controller register to be accessed, which causes the function code and data to be sent to the Controller. If the function is a read, the data word sent with the Fastbus write is ignored, and the data word read is sent to the appropriate data block in the Sequencer. One then has to perform a Fastbus read of that data space location to get the data. In the program, block reads of the EOB registers before and after the read code was executed are compared to determine the location of the data. These operations are transparent to the user, so that writes and reads of the Controller and Digitizer registers can be performed by selecting the same options used to write and read regular Sequencer CSR space registers. Other options are provided to read the supply voltages of the Rabbit crate, download the Sequencer program memory, and change the Fastbus primary address of the Sequencer to allow communication with any other Sequencers in the system.

Many functions, such as running the Sequencer, require only that the user write a certain value to a single CSR register. We did not want to have separate menu options for all these kinds of functions, but neither did we expect that the average user would know where to write which values. Therefore we made extensive use of the ability of Uipack to accept command files. These files are written in standard text format and are simply a list of commands in the order that they could be selected from the menus. Files were made for the simple kinds of functions mentioned above, so that the user will enter something like *@run* to run the Sequencer. Another benefit of using command files during development of the modules is that we can quickly update them if the address of any CSR registers or the value to write to them changes. We also set up more complicated command files that call some combination of the simpler command files. For instance, the command *@prgrun progname* resets the Sequencer, zeroes its Program Counter, downloads the program *progname*, zeroes the EOB registers, and runs the program. Use of the command files saves time, eliminates careless mistakes, and allows users to exercise the full utility of the DAQ modules without having to know any of the details.

B. Testing Utilities

The test portion of the package allows us to access and automatically pinpoint faults in the boards by running a series of automated tests. It was developed in parallel with the prototyping of the modules, so that when any error was found on a board that the test programs failed to detect, the programs were updated in an iterative process so that most of the errors that we have seen in the original DAQ modules will be detected and reported if they occur in newer modules.

Some errors are reported to the screen and all errors are written to a log file that is opened upon entering the test menu. The simplest tests are reads, writes and comparison of all the Sequencer's data and CSR space registers. The user can choose a pattern of sequential numbers, walking ones, walking zeroes, random data, or user defined data words to be written. The location of any mismatches, the value written and the value read are written to the log file. Another test checks if the Block Transfer (BT) mode of reading out the Sequencer is working correctly. In BT mode, when a Fastbus block read of one of the data blocks is executed, the Sequencer generates a Fastbus SS=2 code and stops sending data when the address being read exceeds the EOB register content. For the test, we perform such a read for all possible values of the EOB contents, and report to the log file if SS=2 is not generated, or if the number of words transferred does not agree with the EOB content. Still another test causes errors to occur and checks that the correct bits in the error register are set and that the error can be reset properly.

The correct functioning of the microsequencer chip and conditional control logic is tested by downloading a microcode program that executes all 16 of the instruction codes in the case that the condition was true or false. If everything works, the microcode program finishes at a particular location that the test program recognizes as meaning successful completion. If an error occurs, the microcode program finishes at a different location. The test program reports the error associated with this location to the log file, eliminates the microcode instructions that produced the error, and runs the microcode program again. This is repeated until the "successful completion" line is reached.

The VAX and QPI system used for most of these tests was in certain cases unable to provide a completely effective means for testing the Sequencer. For instance, the QPI temporarily drops the connection signal after a certain number of data words are transferred from any Fastbus device, thus breaking down the transfer of large data blocks into smaller pieces. Since the Sequencer interprets each connection to be a separate transaction, the SS=2 code was not issued for some BT mode reads that were made over two connection cycles. Also, the speed of data transactions between the VAX and QPI is much slower than between two devices in a Fastbus crate. One test we wanted to perform was to attempt to read all undefined CSR space registers in the Sequencer and make sure the SS=7 Fastbus code was issued, but the test would take several weeks since well over 4 billion locations allowed by the Fastbus 32-bit address field were undefined. During normal runtime operation, the Slac Scanner Processor (SSP) is the device that will be used to read out the Sequencer, so in each of these cases, we wrote programs that could be executed on the SSP. In the first case, the proper BT mode of Sequencer readout was verified, and in the latter case, the restricted test was able to be executed in less than two days. Tests such as these have not yet been absorbed into the SVX_Control testing environment, but it is anticipated that they soon will be.

Other menu options can be selected to test the Controller and Digitizers. These modules are accessed by the Sequencer either through the Fastbus procedures described above, or by running a microcode program in which the 32 bits devoted to sending the data and function codes to the crate are properly specified. One test performs writes and reads to all the Digitizer and Controller registers using Fastbus, compares the results, and reports any mismatches. It also checks that the data are being sent to the right block in the Sequencer's data space. One can select whether to test a digitizer in a particular slot or to scan all the digitizers in the crate. Or one can opt to download a microcode program that performs all the writes and reads to the Controller and Digitizers at the maximum transfer rates allowed by the design.

Besides its normal operation of digitizing the analog signal from the detector or the voltage of the Rabbit crate, the Digitizer also allows one to select one of its three DACs as input to its ADC for diagnostic purposes. Two test programs make use of this feature. The first downloads a microcode program that sets all possible values for each of the DACs and causes the ADC to digitize the signals for each value. In one mode, a comparison is made between the DAC input and ADC output, and an error is reported only if the difference exceeds a specified value. In another mode, the microcode program is executed multiple times, the mean and the standard deviation for each value are calculated, and the results are written to a histogram. The second program allows one to select the DAC to use as input to the ADC and the value to write to the DAC, then digitizes the signal multiple times. The result can be printed on screen or used to fill a histogram. The user can interrupt the program at any time to change the DAC selection or value.

Some more options available in the SVX_Control package, such as its program and data space display facilities and its ability to act as a file server and manager of the microcode, will be discussed later. Other functions that can be performed, such as interfacing with other Fastbus or Camac modules, are beyond the scope of this paper.

III. SYSTEM MICROCODE STRUCTURE

A. Description of the Meta-Assembler

The Sequencer's microsequencer chip executes 96-bit wide instruction words, divided into 3 equal parts that provide pattern timing and data latching signals, crate data and control codes, and microsequencer conditional control logic and branching instructions. Use of the HiLevel Assembly Language Environment[5](Hale), a relocatable macro meta-assembler program, allowed us to define source program definitions of instruction formats for our application. The objective is for any user to be able to write a program using only logical mnemonic terms for the desired functions, without having to know anything about the format of the Sequencer control word. The necessary file structures naturally divide into three forms: the definition files, that specify the format of the microcode work and the value of any useful symbols; the

source files, that contain the sets of macro calls used to write the actual programs; and the format files, that store the compiled microcode in a format that can be easily downloaded in to the device(in this case, ASCII).

The Hale definition file for SVX programming is built on four levels. The lowest level is to assign mnemonic names to valid entries in all logical fields of the control memory formats. For example, the microsequencer instruction codes were assigned the standard AM2910 names, the crate control function codes were assigned names such as WRITE or BROAD to execute writes to a single digitizer or broadcasts to the crate, and the digitizer registers were given names such as OFFDAC for the offset DAC register.

The next level is to set up formats corresponding to each logical field of the control memory. Typically, each format specifies only one logical field, with the remaining bits being treated as "don't care" bits that can be overlaid by other instructions. We also assigned default values to each of these fields so that if they aren't specified in a source code program they will be compiled correctly anyway.

At the next level, pipeline macros are set up. There are two kinds of macros in Hale. In both kinds, the macro is specified with some number of call parameters. Within the definition of the macro, these call parameters can be placed in the field of format instructions, or used as call parameters for other macros. When an ordinary macro is called, all microcode words specified within its definition are generated. But when a pipeline macro is called, its instructions are activated, but no code is immediately generated. Most of the pipeline macros consisted of a call to a single format statement, with the call parameters specified in a natural order for the user, even though the order may differ from the actual bit arrangement. Thus many pipeline macros can be called before generating a line of microcode. When a command called *PADPIPE* is executed, all the pipeline macros that have been called generate a line of microcode that is overlaid with the code from the other pipeline macros to create one 96-bit microcode word.

At the highest level, we set up system control macros to control the Analog and Digital SWitches of the SVXIC chip called *ADSWINST*, where *INST* is replaced by one of the 16 mnemonic names of the microsequencer instructions. These macros have twelve parameter fields that allow the specification of the entire instruction word. We arranged the parameter fields so that the four that are most commonly used are specified first. The remaining fields can be left unspecified and the corresponding bits will be set to default values. The macro itself calls all of the pipeline macros with the call parameters that are specified, then executes the *PADPIPE* instruction to overlay the code. Depending on the characteristics of *INST*, some extra logic may be applied to handle unspecified fields. For example, if no condition is specified for a conditional instruction, the bit corresponding to the unconditional line of the microsequencer is set, to cause the instruction to be executed as if it were always true. We also introduced warning messages that are printed to the screen during compile time if illegal chip operations are

specified in the source code. The microcode is still generated in this case, but we have advised users against running programs that result in these warnings.

B. Logical Division of the System Code

At any given time, a user is interested in only a few of the chip operations when modifying the system microcode; therefore we did not want all the other chip operations to have to be continually respecified. HALE has a relocatable linker utility that allows separate modules to be compiled, then linked together later in any order desired. So we divided the entire program structure into several separate functional units, wrote the source files for each, compiled them, and placed them in an area accessible by all the members of the group. A rough description of the units necessary for all calibration and DAQ programs is as follows:

CHIP INITIALIZE: Perform a dummy readout to reset the chip.

WRITE CHIP ID: Write a 4-bit identification number to each of the fifteen chips in a wedge, and set the mode of operation.

RESET AND INTEGRATE: Put the chip switches into a Reset state, then select the sampling of a new event and store it on capacitors.

LATCH: Latch the data stored on the capacitor.

READOUT: Depending on readout mode selected, scan all the channels and either read out all channels, or just those with hits above an internal threshold.

In addition to the operations on the SVXIC chip, there are additional sets of instructions used to control registers in the Digitizers. These are as follows:

INITIALIZE: Set the gain register, charge injection and offset DACs in the Digitizer to desired initial values; read these values back into the data stream.

INJQ: Vary the value of the Vcal DACs on an event-by-event basis (this is used for calibration only).

READ DIGITIZER: Read out the current value of all the Digitizers' registers into the data stream.

One last set of instructions is needed to control the program flow in the Sequencer based on the signals from higher level systems:

SYNCHRONIZE: Select Reset and Integrate cycle if a signal called Clear and Strobe is received, or select Latch and Readout if a signal called Start Scan is received (for a description of these signals, please consult reference 1). The chip is held in a Partial Reset state during this time, meaning that the charge collected on the capacitors in the previous Reset and Integrate cycle is held in place.

Most of these sets of instructions can be set up to work in all readout modes, and no further changes can be made that enhance performance of the chip. However, some others require precision tuning. In order to allow such tuning of parts of the acquisition cycle without requiring re-specification of the other parts, we have designed each of the sets of instructions listed above to be fully modular by providing labels designating entry and exit points at the beginning and end of each set. After fine-tuning and compiling one set of

instructions, the user then links with the other pre-compiled modules to regenerate the complete acquisition cycle.

The particular implementation of this utility is dependent on the chip function to be performed. The most general state transition diagram is shown in figure 1. By design, the most straightforward application is for real event acquisition, in which case all the modules listed above are linked. The only restriction on the order is that the Initialize module be first.

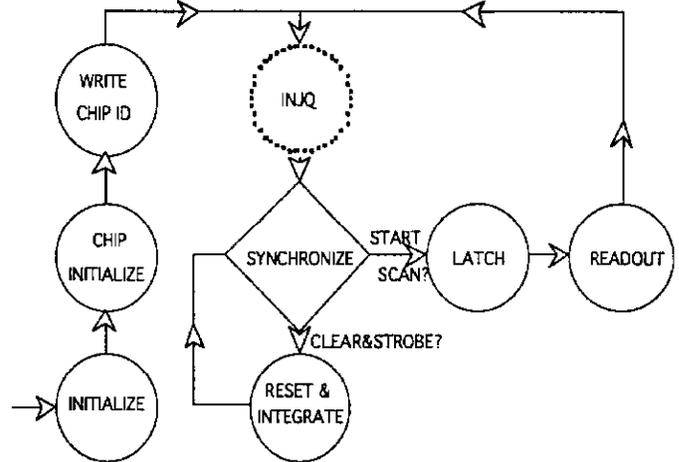


Figure 1. Typical order in which the Hale relocatable modules are executed.

For calibration programs, slightly more sophisticated means must be employed. For example, a leakage current calibration is performed by varying the integration time. Assuming the leakage current is constant, its magnitude is simply the change in charge accumulated on the capacitor divided by the change in the integration time. One can just modify the integration module to produce several different integration times, then link it with the other modules in the package.

Gain calibration is performed by computing the change in the readout pulse height divided by the amount of charge injected. This is a quality measurement for all channels, so every channel is read out. Threshold calibration consists in determining the amount of charge that must be injected in order to cause a given channel to be above the chip's internal threshold 50% of the time. In both cases, the variable quantity is the charge injection. All that is necessary is to set up an INJQ module for each of these, and link the other modules together, with latch-all readout for gain; latch sparse, for threshold.

IV. FORTRAN UTILITIES FOR MICROCODE MANAGEMENT

A. File server in SVX_Control

In order for all users to be able to compile and link the programs, we set up command files and defined symbols to execute them. A set of all the modules listed in the previous sections was kept in a restricted area on disk. Members of the

group had read access only to this area. If a user wanted to modify one of the modules, he could copy it to a separate work area, change it, and compile and relink a new microcode program that would also be placed in the work area. There were two problems with this procedure. The first is that the user would have to know which modules to link and the correct order in which to link them to generate the appropriate program. The second is that several users may be modifying the same programs in the same directory. We solved these problems by implementing a file server facility from a separate menu in the SVX_Control environment. Upon entering the menu for the first time, the user executes a "new user" option. This option copies a complete set of the modules to the work area, but appends the username to the end of each filename. Files that contain the list of modules to be linked in order to generate a given calibration program are also created. Then the user chooses the kind of calibration program to be modified and is presented with a list of the modules that can be modified. After selecting which modules to modify, an edit session is created for each selection. The user is asked whether to compile the modified modules and link a new microcode program. If so, the appropriate command files are called. The name of the microcode file is requested, and the file is written to a library area where it can immediately be downloaded and executed by SVX_Control. In this way the user doesn't have to know anything about the names of individual files or the order in which to link them; it is all done automatically for him.

B. Display utilities in SVX_Control

Prior to the introduction of the Fastbus DAQ modules, there was a simpler system which used the Camac system. All the microcode for the Camac sequencer was done in a binary, column-oriented text file. The functionality of that module was less than the Fastbus Sequencer, so the number of bits to specify were fewer, but to the average user the file was just a meaningless collage of zeroes and ones. There was no means to generate these files from higher level source code, so any changes had to be done directly in the microcode file. Nevertheless, several people in the group became quite adept at recognizing the chip operations that were performed by each binary pattern. Therefore we added an option in SVX_Control that would display the parts of the microsequencer program that had analogous functions in the Camac sequencer in the familiar binary pattern to which some people had become accustomed. Other display options are available that break the 96-bit words into the logical fields, displayed according to the field width in binary, octal, or hexadecimal format.

C. Pattern plotter

A standalone Fortran program was written that can read the formatted microcode files, interpret the microsequencer instruction codes, and generate timeline flow charts showing the pattern of timing signals that would be sent to the SVXIC chip when the program is run on the Sequencer. This not only

allows one to immediately spot logical flaws in the program structure, but also allows users with more experience with operating the SVXIC chip to quickly determine if the program will deliver a pattern that insures optimal operation of the chip. At the moment, the program generates the plots using the assumption that all condition codes are true. It is anticipated that in the near future the program can be interfaced with a file that specifies the timing of external signals sent to the Sequencer, so that conditions can be properly evaluated. This will result in a fairly comprehensive simulation of the actual execution of the programs in the Sequencer.

V. CONCLUSIONS

The idea of a design of one software package to operate and maintain three different hardware devices has already proven effective. We observed good error coverage, limited supervision during the test run and short repair time due to automatic identification of problem areas. The package helped to save a lot of debugging time in a highly complex data acquisition environment.

The management and structure of the assembly language to generate the microcode programs for operation of the SVXIC chip is such that any user who knows what switch timing to select is able to write, compile, and link such programs without knowing any of the details of the DAQ chain. This allowed for more attention to the operation of the detector without many people having to spend a lot of time understanding the Sequencer, Controller, and Digitizer modules. The same method could be applied to any other similar systems, and the existing assembly language could be easily modified to work with future versions of the Sequencer, without any change noticeable to the user writing the system source programs.

VI. REFERENCES

- [1] S.M. Tkaczyk, *et al.*, "Commissioning of the DAQ Control and Data Acquisition for the CDF Silicon Vertex Detector", these proceedings.
- [2] K.J. Turner, *et al.*, "Control and Data Acquisition for the CDF Silicon Vertex Detector", these proceedings.
- [3] S. A. Kleinfelder *et al.*, "A Flexible 128 Channel Silicon Strip Detector Instrumentation Integrated Circuit with Sparse Data Readout", *IEEE Trans. Nucl. Sci.* NS-35 171 (1988).
- [4] D. R. Quarrie, "Uipack: The CDF User Interface Package Programmers Reference Manual", CDF internal note #372.
- [5] HiLevel Assembly Language Environment User's Manual, Irvine, CA, November 1987.