Lucid and Vme Modules
(U of S Subatomic Physics Internal Report SPIR-122)

Anthony Del Frari

October 29, 2003

Contents

List of Figures

Chapter 1

Project Purpose and Description

Some researchers in subatomic physics at the University of Saskatchewan conducts particle physics experiments using gamma ray beams. This involves collecting and analyzing large quantities of data electronically. For this purpose the Lucid system was developed. The system incoporates both an online data acquisition system and an offline analysis system.

The data acquisition system currently uses a VMEbus processor to read data off of data acquisition modules (adc's and tdc's). These modules currently run on the camac and fastbus backplanes. These busses have relatively low data transfer rates. This results in large amounts of data being lost as the modules wait to be read out. The VMEbus backplane allows for significantly faster transfer of data between modules. For this reason it was decided to extend the Lucid system to allow the use of data acquisition modules on the VMEbus. This document outlines the results and details of that effort.

Chapter 2

Overview of Data Acquisition System and Lucid

The Lucid system allows for event-by-event real-time recording and analysis of data from an experiment. To do this reliably it is split into several sections. A data acquisition portion called the "reader" collects data from instruments and places it in memory for the other portions of Lucid to access. The data can be written to long term storage (hard disk, or tape) using the "writer" portion of the Lucid system. The analysis of data takes place in the "looker" portion of the Lucid system.

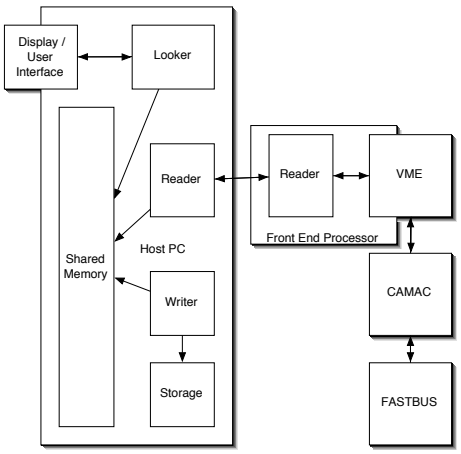

Figure 2.1: System Block Diagram for the Lucid System

In the operation of an experiment the user defines "reader", and "looker" files that define how the user wants to run those portions of the system. This system allows an experimenter to easily define an experiment without being bothered with all the details of how the individual pieces of equipment work. The main equipment that the Lucid system interacts with are a controlling/user PC, an online embedded processor sitting on the VME bus, and data acquisition modules (adc's, tdc's, etc) that sit on the VME or CAMAC busses.

The data acquisition system is comprised of three parts. A set of persistent daemons running on the online frontend system that allows for the controlling and testing of equipment without the rest of the Lucid system running. The other two portions are collectively called the "reader". The one part runs on the controlling PC and the other runs on the frontend processor. The PC portion passess data and commands between the frontend processor and the rest of the Lucid system. The frontend portion of the reader executes the commands that the experimenter defined in the "reader" file, and passes the accumulated data back to the controlling PC by ethernet.

The data acquisition boot sequence is a two step process. The first step occurs when the user boots up the frontend system. The operating system is then loaded remotely from the PC using bootp and tftp. When the system boots several daemons are started. They are the exception handler, the nethv daemon, the netcamac daemon, and the pdl downloader daemon. The exception handler allows the system to report error information to the user when the system crashes. Nethv is a daemon that allows for remotely controlling camac based high voltage controllers. The netcamac daemon allows the user to remotely send commands to camac modules and the view their response. The pdl downloader is used to load the data acquisition software.

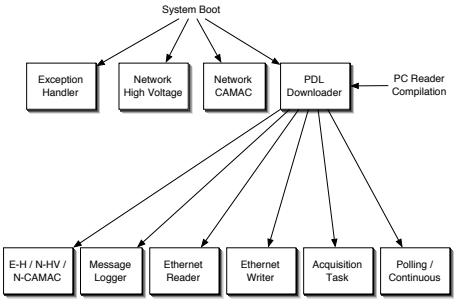

Figure 2.2: Sequence of task creation on the frontend processor

The second step of the process occurs when the user uses the Lucid programs to compile a frontend reader. After compiling it the Lucid software contacts of pdl downloader daemon and sends the data acquisition software to the frontend processor. The pdl daemon loads this into memory and causes the system to restart. The loaded data acquisition system then restarts the original daemons as well as starting the data acquisition specific daemons. The data acquisition

daemons are the message logger, the ethernet reader, the acquisition daemon, the ethernet writer, and the polling/continuous daemon.

The messsage logger is used send data acquisition messages to all the places a user could want it. These include the console, telnet port, and if present to an ethernet logging task. It is also responsible for restarting the system if a fatal error occurs. The ethernet reader daemon is responsible for reading user commands from the controlling PC and passing them on to the acquisition task. The ethernet writer task is responsible for taking the buffers the acquisition task fills with data end sending them to the controlling PC.

The acquisition task is responsible for reading data. It waits for an event to occur (timer, CAMAC Lam, etc.) it then executes the appropriate acquisition code. The polling / continuous process is responsible for having the acquisition task complete an operation as fast as the system can respond. The code that the acquisition task executes upon reciept of an event is user defined in the 'reader' file. This file is 'compiled' in a two step process. First, the program 'vmebuildreader' is called on it. This program parses the file and outputs a C program that will execute the required commands when called by the acquisition task.

Chapter 3

Results

3.1 The VME Modules Database

In the current Lucid system the properties of CAMAC modules are defined in a database. This allows the user to use a module without knowing the exact details of all the commands needed to use the module. This same functionality was wanted for VMEbus modules.

The database consists of a list of modules with the commands and aliases that are associated with them. To define a new module is simple. Each module definition starts with the keyword module followed by a long descriptive name for the module in quotes. Shorter names(aliases) for the module are given indented on the following line.

module “<descriptive name> ”
“ <short name> ”
“ <short name> ”
.
.

For example:

module “CAEN 792 32 channel charge ADC”
“792”
“caen792”

Following the aliases for the module are two important lines. The first is the readsize line. This line defines the size of the buffer used by Lucid when it does a ”read” operation on the module. The line says ’x by y’ this defines an array of length x with entires of size y in bits. The final portion of the line is optional, and specifies the default timeout for the module when a wait command is done on it. This number is in microseconds.

readsize is <array length> by <data size> [timeout <microseconds>]

For example:

readsize is 34 by 16 timeout 1000

will create an array of 34 shorts (16 bits). It also specifies the wait timeout as 1 millisecond.

After the readsize specification the user can specify a set of masks for use in the compression system. They are specified by the following lines.

compressionvalidmask <valid data mask>
compressionendmask <and mask> [<equals mask>]

When data compression is invoked the 'valid data mask' is bit-wise anded with the data and only non-zero values are saved. The compression can be ended before the readsize parameters by the use of the end masks. These masks are employed in a boolean expression that ends the reading of data as below.

(data && <and mask>)==<equals mask>

The VME modules database supports the use of three kinds of commands. Each command consists of the name of the command followed by the keyword **is** and the information that defines it.

The first is simply the definition of a register. The information needed is the location of the register as an offset into the modules address space, the address and data sizes, if it is read/write or both, and whether it can be accessed by geographic address or if supervisory data mode is to be used.

<command name> is <address< <address width> <data width> [r][w]
<addressing mode>

For example:

bitsetone is 0x1006 A32 D16 rw geo

The second type of command allows for the reading or setting of a single bit in a type one command. The name of the type one command must be given and the number of the bit to be set. The main use of this type of command is to provide for meaningful names in reader files.

<command name> is <register name> bit <bit number>

For example:

seladdron is bitsetone bit 4

If seladdron is the register at address 0x1006 then if this command is called on a module with a base address of 0x040000 it will generate code that looks like the following:

$*$(unsigned short*)0x041006 |= (1 << 4)

The third type of command is used when a complex operation is to be performed. This type gives the name of a function to be called when the command is used. At the end of the line are optional parameters in quotes. This string will be passed to the function at run time.

<command name> is function <function name> [“<function paramaters>”]

For example:

read is function caen_read “read params”

A read function command can also be defined that directly supports the lucid compression format. Such a read function definition would include the keyword **compression** before the keyword **function**.

3.1.1 The Caen Functions

At the time of writing there were six function commands defined for the caen qdc and tdc modules. These commands are: **caen_compress_read**, **caen_read**, **caen_read_sort**, **caen_init**, **caen_clear**, and **caen_thresholds**.

The function **caen_compress_read** is a compression-aware function that can be used on caen tdc and qdc modules to directly read data out of them and into lucid compressed format. Using this function in a compressed read removes the necessity of decoding the data in the looker. The **caen_read** is just a straight forward function that reads the data out of the caen modules. This data must then be decoded in the looker. This last read function, **caen_read_sort**, reads the data out of the modules and decodes the data for storage. Use of this function does not require the data to be decoded in the looker. To clear all data from a caen module without reading it into memory the function **caen_clear**.

The caen modules can be initialized using the function **caen_init**. This will set the internal address registers to the address given to the cards, so that even if the address wheels on the card have not been set the cards can be used for data retrieval. This function will also set the cards lam level and vector if they have been given.

The **caen_thresholds** function reads thresholds from the provided buffer and writes them to the module. This function is also used to turn off any undesired channels. The input is just a sequential array of the thresholds that are to be applied to the module. The maximum size of this array is then 32, i.e: the number of channels in the module. If a channel is to be turned off then its entry in the array should be 0x100. A channel can be skipped by using the value 0xffff.

3.2 Netvmed

The Netvmed program and daemon fulfill the analogous purpose of the Netcamacd program and daemon do for CAMAC. They both allow the user to access

the modules from the command line regardless of whether Lucid is running or not. Netvmed uses the vme database to map command names into operations as well as allowing the user to use a generic memory access of the VMEbus modules. Lucid and Netvmed use the same vme module database.

The Netvmed program is used by inputting a command as the arguments to the program. The program will then save these values as long as it is running. The program will prompt the user for any information that is not provided on the command line. When it has all the information it needs it will send the requested command to the online daemon. The daemon will send the requested command to the module. When a response is returned it will be passed on to the user. Then Netvmed will loop around and ask for the information not given on the command line again.

Note: The Netvme command and daemon do not support the use of function type commands.

3.3 Lucid

3.3.1 Defining a vme module

The definition of a vme module in the reader is much the same as the definition of a CAMAC module. After the keyword define comes the name to be associated with the module. Then either a slot, an address, or both must be defined for the module. Last one can optionally give a lam (interrupt) vector for the module and the vme interrupt level for the module. If the interrupt level is not given it will default to 3 the same level that the CAMAC modules use.

define <variable name> "<module name>" slot(<number>)
address(<number>) [lam(<number>) level(<number>)]

For example:

define adc "792" slot(5) address(0x00050000) lam(0x45) level(4)

3.3.2 Using VMEbus Modules

VMEbus modules in Lucid are handled in almost exactly the same fashion as CAMAC modules. All VMEbus modules must be defined in the appropriate database to be used in Lucid in exactly the same way that CAMAC modules must be defined in the appropriate database. When that is done VMEbus modules can be used identically to CAMAC modules with the use of read, clear, save, and wait commands.

For CAMAC modules there is a generic CAMAC command that allows the use and operation of a CAMAC module without having to define the module in the reader. For VMEbus modules there are two versions of this command. There first is much the same as the CAMAC command in that the module need not be defined. This command simply takes a memory address, a operation

type (i.e. read or write), address and data sizes, and what VMEbus address modifier is to be used for the operation.

vme <address> A<address width> D<data width> <read/write> [data <variable>]

For example:

vme 0x051006 A32 D16 read data variable

The second version of the generic VME command requires that the module be defined in the reader and in the appropriate VMEbus database. This command takes a module name and a command name and executes the command from the database that fits that information.

vme <module variable> <command name> <read/write> data <variable>

For example:

vme adc bitsetone write data 0x10

Both versions of the generic VME command can use variables and arrays for both reads and writes. In addition constants can be used in place of a variable when writing to a module.

3.3.3 VME Module Interrupts

At the time of writing Lucid was not able to respond to interrupts raised by a VME based module.

Chapter 4

Details

4.1 The VMEbus

The VMEbus system consists of a backplane bus, and a set of insertable modules that communicate with each other across the backplane. The modules can be for data acquisition (ADC's, TDC's, etc), processors (MVME167, etc), etc. The address space can be 16, 24, 32, or 64-bits and data can be sent in the same bit ranges. Address modifiers are used to control access to the bus and how the modules interact. A VME crate can have up to 21 slots for modules. Newer crates and modules allow the slot number to be used in addressing the modules, this is called geographic addressing. The module in slot 0 of the crate is the crate controller and controls access to the bus.

Interrupts can be placed onto the bus interrupt lines by any module. The VMEbus has seven interrupt lines with 1 having the highest priority. When an interrupt is seen by the crate controller is it causes a signal to propagate through the modules in search of the module that raised the interrupt. When the correct module get this signal it places its interrupt vector on the data lines so that the crate controller can read it.

For more information on the VMEbus visit 'www.vita.org'.

4.2 The MVME167 Processor Board

The current frontend processor for the Lucid vme system is the motorola MVME167. The processor is a motorola 68k microprocessor which talks to the VMEbus through a VMEchip2 chip on the board. The VMEchip2 maps the VMEbus into the 68k's memory space and turns VMEbus interrupts into 68k interrupts.

The VMEchip2's internal registers are mapped into the 68k's memory space at 0xFFF40000 through to 0xFFF4008C. The registers for mapping 68k memory to VMEbus memory is addresses 0 through 10 of that range. Mapping VMEbus to 68k uses 14 through 28. Managing interrupts uses the range 68 through 88.

The VMEchip2 sets up three default mappings. A 24 bit address range from 0xf0000000 to 0xf1000000, 32 bit address range from 0xf1000000 to 0xff7fffff, the last region is a 16 bit address region from 0xffff0000 to 0xffffffff. All of these regions can run in 16 or 32 bit data mode, and can operate with several different address modifiers.

In addition to these Netvmed sets up two more regions. The address range 0x0f000000 to 0x0fffffff is setup for 24 bit addresses with 16 bit data. This regions address modifier is set to use the geographic addressing scheme. Geographical addressing only supports 24 bit addresses. The MVME167 only has options for 16 or 32 bit addresses, so this range is setup to think that it is using 24 bit addresses. The second region set up is 0x04000000 through to 0x04ffffff which is also setup for 32 bit addresses but with 32 bit data. Its address modifier is setup for supervisory data access.

Starting Address	Ending Address	Address Mode	Data Mode	Address Modifier
0xf0000000	0xf1000000	24	16	UDM
0xf1000000	0xff7fffff	32	16	UDM
0xffff0000	0xffffffff	16	16	UDM
0x0f000000	0x0fffffff	24	16	GEO
0x04000000	0x04ffffff	32	32	SDM

To enable the 68k to see a VMEbus interrupt several things need to be done. First, the appropriate VMEbus interrupt needs to be unmasked. Then the 68k interrupt level that it will map to has to be set. Finally the master interrupt enable on the VMEchip2 has to be set.

For more details on the mvme167 processor board go to 'mcg.motorola.com' and search for MVME167.

4.3 pSOS

pSOS is the real time operating system that the data acquisition system uses on the MVME167. It supplies all the necessary task handling and interaction functionality that the system needs. It also implements BSD networking and thus allows the data acquisition system to transfer data over a network using standard UNIX networking functions. pSOS also allows for the setup of custom interrupt handlers.

4.4 The VME Modules Database

The database consists of a list of module names and the commands that are associated with them. To be able to use this information it needs to be parsed and stored in data structures during the operation of Netvmed and the vmebuildreader portion of the Lucid system. The parsing is done with the compiler generation language YACC (a.k.a. Bison), and the parsing tool Lex (a.k.a. Flex).

Type one and two commands are handled entirely with the Lucid software and the Netvme software. Type three commands require someone to write the function associated with the command. This function is written in the functions subdirectory of the vmebuildreader/vmelibs source directory. Place your code in a subdirectory with a name ending in '.fn'. For example the code for the functions used by the caen adc's and tdc's are in the directory 'caen.fn'. This will allow the automatic inclusion of your code in the functions library. The function must be compiled using the 68k cross-compiler into an object file that can be joined into a library.

There are two types of functions that can be created. Those that take a buffer of shorts and those that take a buffer of longs. To declare a function that takes one of these use the appropriate macro from lucid/include/vme.h. For example:

```
#define VME_FUNC_S(name) void (name)(unsigned short *data, int size,
                    struct runtime_data* rt, char* params)
```

could be used to declare a function named quickread that takes a short buffer as follows

```
VME_FUNC_S(quickread)
{
...
}
```

The variable data is the buffer that is passed to the function, and it must of atleast size 'size'. The variable rt contains the runtime information about the module that the function is being called upon, including slot number, address, interrupt vector and level. The params variable is defined in the module file. At the end of the declaration of the function command in the database any text in quotes will be passed to the function through the parameter 'params'.

To create a read function that can be used to do custom decoding of data into lucid compressed format one of the two compression aware function prototypes must be used. These are **COMPRESS_VME_FUNC_S** or **COMPRESS_VME_FUNC_L**. Either of these two will give the function access to the lucid data pointer through the variable dataPtr.

```
#define COMPRESS_VME_FUNC_S(name) void (name)(unsigned short
*dataPtr, int size, struct runtime_data* rt, char* params,unsigned long*
                    bytesWritten)
```

The function should write data into the dataPtr and advance it as it goes. When it is done it should write the number of bytes it has written into the parameter bytesWritten. This value will then be used by lucid to set the compression offsets for the read. **Note:**compression-aware functions should only be used as read commands.

4.4.1 Extending the Parser

There are two steps to extending the parser. The first is to add any new symbols to the lex parser file 'vmeparser.l'. The second step is to include any grammar changes in the file 'vmegrammar.y' so that when the parser sees your extensions the appropriate code will be called. These files can be found in the directory '.../vmebuildreader/vmelibs/' in the Lucid source.

Adding a keyword requires changes to both parser and grammar files. In the parser file the keyword is identified to the parser along with what it should return to the grammar file when it is found. For example the following line is for the keyword 'function'.

```
function                { return (FUNCTION); }
```

In the grammar file the keyword has to be declared as a valid token. This done by adding a line that begins with the '%token' at the beginning of the line. For example to add the token 'FUNCTION' one would add the line

%token FUNCTION

Once this has been done any extension to the grammar must be done. A basic example of this is as follows.

```
command:    STRING IS FUNCTION STRING
                {
                        printf("Command found\n")
                        executeCFuncion($1,$4);
                }
                ;
```

A grammar rule consists of a name (command) and a set of tokens. If the input to the parser matches the set of tokens then the C code in the scroll braces is executed. The values of the tokens can be referred to as $n where n is the number for the token in the set starting at 1. The semicolon indicates the end of the grammer rule.

For more information on yacc and lex see there respective man pages. More extensive tutorials and documentation are available from the GNU Foundation 'http://www.gnu.org'.

4.5 Netvmed

Netvmed consists of two components. The offline program that interacts with the user, and the online daemon that sends requests from the program to the VMEbus modules. The online daemon operates very simply. It takes the command sent to it by the program and determines what operation the user wants it to perform and how many times it is to be performed. The options are 16 or 32 bit reads or writes. The daemon then performs the operation the requested number of times and sends the appropriate response back to the program.

The first thing that the program does is load in the vme modules database. Then it sets the variables that are passed into it, prompting the user to enter any additional information that it requires. It searches the database for the entry of the entered command. It then builds the request to be sent to the daemon and sends it. When the daemon responds Netvmed parses the returned message and prints out the results.

4.6 Lucid

This project was concentrated on the vmebuildreader portion of the Lucid system. For this reason this portion of the document will concentrate on this portion of the system.

4.6.1 The reader parser and grammar

The vmebuildreader parser is based on the same yacc / lex system as the vme database parser. The lex portion looks for keywords in the inputted code, which it passes to the yacc grammar to be matched against the rules for the reader file. The files involved in this are scanner.l and grammar.y. Adding keywords to this parser is quite simple. First define them as valid tokens in the yacc file then add them to the array kwords in the lex file. Then one can add any rules associated with them into the grammar file.

4.6.2 Definitions and The Symbol Table

As the reader parser and grammar dig through the reader file they build up a symbol table of the symbols that they see. This includes module names, variables, trigger names, and event names. The symbol table allows vmebuildreader to keep track of the variables that have been declared and the properties associated with them. For CAMAC and VME modules the symbol table entry includes both information from the appropriate database about the module as well as information defined for the module at run time. The symbol table entry for a module represents both the module and the variable for it by the same name.

The VME symbol table entry is constructed in the file makevmemodule.c by the function makevmemodule(). This function first looks up the module in the vme database to obtain the general information about the module like how big its read buffer should be. It then tests this module against all previously defined modules (including the CAMAC branch drivers) for conflicts. The possible conflicts are with slot numbers, addresses, or lam vectors. If a conflict is found a warning is displayed to the user but the code generation will continue. The symbol table entry created includes the address and/or slot of the module and optionally the lam vector and level that the module will use. This is the runtime data that the system needs to be able to identify the module at runtime. From the database definition of the module the readsize of the module is obtained.

This includes the size of the data and the amount of it that will be read from the module. This information is used to determine the size of the buffer to be used when the user does a 'read' command on the module. Vmebuildreader also obtains from the database entry the default timeout for the module if one is defined. If it is not defined the value will default to 0.

4.6.3 How a reader command is turned into C code

This section will track the progress of the reader command below through the system.

read, save and clear adc1

Let us assume that 'adc1' has already been defined as a vme module. When this command is seen by the grammar it calls the function code_attrib() with the flags ATTR_READ, ATTR_SAVE, and ATTR_CLEAR. There are also flags for wait and compress commands. The function code_attrib() is responsible for turning these types of commands into into C code and it is defined in the file 'code_attrib.c'. This function starts with a sanity check on the flags. For example, one can not save and clear without also doing a read. It also changes a read and a clear into a readclear.

After testing the flags the system then calls specific functions to handle each data type. In the case of a vme module the command gets sent on to the function attr_vme(). Here wait commands are sent of to be dealt with separately from other commands. If the command is to save data the first thing to be done is tell the system that it must reserve space for it in the event structure. To write the C code the modules database is queried to see if there is a definition for the command that is to be executed. If there isn't one then the system will print out an error message and stop execution. When the commands information has been obtained it is passed to the vme code writing subsystem. This is done through the function code_vme_command() in the file 'code_vme.c'. Here the vme commands are broken up into the different types and each is turned into code. If necessary there are loops placed around the code that actually accesses the module.

4.6.4 Wait commands

The code that handles wait commands on vme modules is in the function vme_wait(). The function is in the file 'code_triggers.c'. Wait commands use the modules database defined command dataready. If a module does not have such a command a wait command can not be used on it. The way it works it that the acquisition task will use the dataready command to poll the module until it gets a non-zero return. To keep acquisition from polling for ever a timer is used with a module specific timeout to break the loop.

Appendix A

Making ‘init’ a keyword

This chapter outlines the method by which the word ‘init’ was made a keyword in the reader language. Doing this required two steps, 1) Modify the lex parser, 2) Modify the yacc grammar. The following sections outline how this was done.

A.1 Lex

The lex parser for the reader language is contained in the file:

‘ lucid/src/vmeaq/vmebuildreader/scanner.l’

The first step in this process is too add the ‘init’ to the array kwords. This array is used by the parser to lookup keywords and return the appropriate value to the yacc grammar. This array must be in alphabetical order so it is important to insert the word in the correct place. The entry for init looks like:

```
{ "inhibit",INHIBIT,0 },
{ "init",INIT,0 },
{ "initialize",INITIALIZE,0 },
```

The second parameter is the value that will be returned to the grammar. The last parameter if set gives the user a warning that they are using a looker reserved keyword as a variable and will not be able to access it. In this case it is not necessary.

A.2 Yacc

The yacc grammar for the reader language is contained in the file:

‘ lucid/src/vmeaq/vmebuildreader/grammar.y’

The first step to get the keyword ‘init’ to be recognized is too add to tell yacc that it is a valid token (word). This is done with a statement like the following:

%token <s_token> VME INIT

Make sure that the name of the token here matches the value returned by the lex parser when the word is seen. The keyword 'init' will be used in a reader file in a line like below:

vme adc1 init

Lines like this one are called statements in the grammar. To implement the use of the keyword 'init' the following lines are added under 'statement':

```
|       VME NAME INIT
                {
                struct Stab *sptr;
                if((sptr = lookup("init")) == NULL)
                        sptr = install("init",SYM_UNDEF);

                code_vme_cmd($2,sptr,VME_OP_NONE,NULL);
                }
```

The first line says that when the grammar sees the tokens: VME,NAME, and INIT it should execute the C code that follows it. It then checks to see if the word 'init' is in the symbol table and if its not it will create it. Once the word is in the symbol table the code that will actually execute the command in the frontend code is generated.

Appendix B

Writing a custom function command

Custom vme function commands allow the user a vme module to gain direct access to the abilities of a module. The main use seen for this feature is to encapsulate length sequences of commands that are used often in the reader file.

Custom commands are stored in the '.fn' subdirectories of the directory

' lucid/src/vmeaq/vmebuildreader/vmelibs/functions'

From these directories object files and header files are combined to create a library of the functions that are available to the lucid system. Each directory should include at the least one header file, a source file, and a Makefile to compile the code into object code. Note that the code must be compiled using the appropriate cross-compiler.

B.1 Writing the function

As discussed in chapter 4 there are a series of macros that are used to define the prototypes for custom functions. These are defined in the file vme.h in the runtime subdirectory of the functions directory. These macros must be used as they tell the automatic make system the names to the valid functions. If you have a module that will read data a 16bit or less values use the short functions, otherwise use the long functions. If you are writing a function that will do custom compression of data use one of the compress macros. Once this is decided it is time to write your function.

In this section we will study the following function from the 'caen.fn/caen.c' set of functions.

```
VME_FUNC_L(caen_read)
{
```

```
        int i=0;
        unsigned long address;
        unsigned long value;

        address = 0;

        if(rt->address != -1)
                address += rt->address;
        else
                LogFatal("caen_read: must give an
                         address to read module\n");

        i = 0;
        do {
                value = VME_READ_32(
                        VME_ADDR_A24_D32_SDM(address));
                data[i] = value;
        }while(!INVALIDDATA(value) && ++i < size);
}
```

This defines a function names 'caen_read' that will accept a 32bit buffer. The first thing the function does is check if the address of the module has been defined. This module requires a valid address for data to be read from it. This module also supports the use of geographical addressing and if the user has not defined the modules address this function can not succeed. The call 'LogFatal' will print out the error message in the Lucid messages window and then the frontend code will stop. To print a message but not kill the frontend use the command 'LogMessage'.

Once assured of a valid address the function starts to read data from the module. Reading is done with the macro 'VME_READ_32'. Note that the address of the module is mapped to a different value using a macro. Both of these macros as well as related ones are defined in the previously mentioned vme.h. This function exits when it reads invalid data from the module or it has filled the buffer. Always ensure that you do not read or write outside of the buffer.

B.2 Getting it included

Once the function is written there are three steps to getting it into the system:

- 1) In the directory ' lucid/src/vmeaq/vmebuildreader' execute the following commands:

  make clean
  make
  make install

This will rebuild libvmeparser.a and incorporate it into vmebuildreader.

- 2) In the directory ‘ lucid/src/util’ execute the same three commands. This incorporte the new libvmeparser.a into the netvme command.
- 3) Create a command entry for the function in the vme.modules file.