Interfacing GPIB: A Generic Approach

Sharon Chao

Office of Science, Science Undergraduate Laboratory Internship (SULI)

University of California, Berkeley

Stanford Linear Accelerator Center

Menlo Park, California 94025

8 August 2003

Prepared in partial fulfillment of the requirements of the Office of Science, Department of Energy's Science Undergraduate Laboratory Internship under the direction of Linda Hendrickson in the Next Linear Collider Division at Stanford Linear Accelerator Center.

Participant: _____

Signature

Research Advisor: _____

Signature

ABSTRACT

Interfacing GPIB: The Generic Approach. SHARON CHAO (University of California, Berkeley, CA, 94720) LINDA HENDRICKSON (Stanford Linear Accelerator Center, Menlo Park, CA 94025).

With the capability of interfacing an extensive band of instruments, the General Purpose Interface Bus (GPIB) has become an integral part of computer systems. For the purpose of optimizing its universality, an interface between the user and the GPIB devices can be implemented to allow nonspecific commands to be used. In addition to writing a generic interface that can be applied in MATLAB, GPIB instruments can also be accessed through the Experimental Physics and Industrial Control System (EPICS). This option was initially explored when the EPICS support for GPIB was built, and then added to the Input Output Controller (IOC). Subsequently, hardware modules could be attached to the VME crate of the IOC, and configuration data could then be read to the database layer. This procedure, however, did not meet generality of the specifications, so the former option was investigated. When the interface was ready, the oscilloscope, arbitrary waveform generator, and various other GPIB devices were connected to the LAN-GPIB gateway for testing. This project would be useful as a diagnostic tool for several Next Linear Collider Research and Development (NLC R&D) applications such as the NLC Test Accelerator (NLCTA) and 8-pack, used to demonstrate beam acceleration; feedback on nanosecond timescales (FONT), used to demonstrate control of beam collisions; and the vibration project, used to demonstrate final focus vibration stabilization.

INTRODUCTION

In preparation for building a collider of unprecedented magnitude, the Next Linear Collider Research and Development (NLC R&D) team has employed diagnostic tools to facilitate the demonstration of beam acceleration, collision, and stabilization. Because the language used by these diagnostic devices is low-level and unnatural, a communication layer must be created to allow interfacing between users and the NLC Test Accelerator (NLCTA) diagnostic and control hardware systems. Often, these interfaces can be implemented with numerous different approaches, which should each be evaluated based on whether they meet specifications and whether they can be completed resourcefully.

More specifically, instruments can be connected to the network through the GPIB-LAN gateway, and interfacing these GPIB devices can be attempted from various directions. The first and most obvious method is with the support of the Experimental Physics and Industrial Control System (EPICS), a widely-distributed compilation of software applications initially constructed by Los Alamos National Laboratory and Argonne National Laboratory but now built collaboratively by users; this system is now used by several large organizations—Stanford Linear Accelerator Center (SLAC), Berlin Electron Synchrotron (BESSY II), Lawrence Berkeley National Laboratory (LBL), and KEK B-Factory—as well as a multitude of lesser known laboratories. Using EPICS would apply an extensive hardware controller that should only need to be configured appropriately to recognize specific GPIB devices. Although this procedure would require minimal effort, its suitability must be examined first.

Another approach is to write an interface that would extend upon the GPIB-LAN module's ability to connect a broad range of instruments and allow the user to feed the devices generic commands. A stand-alone program must be built first, and then another layer of code

must be written to allow the user to communicate with GPIB devices through MATLAB. The objective of this procedure would be to produce a universal interface that retains its modularity and levels of abstraction.

INVESTIGATING EPICS

At the start, the most straightforward path appeared to use a distribution of EPICS [1] because it would require only software configuration rather than actual coding, which can be time-consuming and result in inconsistencies; in addition, EPICS' broad spectrum of applications has been the source of a unifying backbone to various developments at SLAC including the NLC R&D projects NLCTA and 8-pack. The central constituents of the system include the operator interface (OPI), a workstation running on UNIX; the input output controller (IOC), a chassis running on VME/VXI that can be connected to I/O modules; and the local area network (LAN), the means by which the OPI and IOC communicate. Besides the hardware, EPICS provides a number of software components—core software to build EPICS, IOC software, OPI tools to preserve consistency across the system, and channel access, software for accessing databases from the IOC independent of the network. These components comprise the tool necessary for building a control system.

For the purpose of exploring EPICS as well as configuring it to recognize the GPIB instruments, we built and examined EPICS version 3.13.2, the version previously used at SLAC. After this distribution was compiled with a copy of the EPICS setup file, we were provided with the core software including Channel Access and databases. A few example IOC applications were subsequently built with MakeBaseApp [2], a perl script supplied in the EPICS base. With these applications, we could see a sample of EPICS' considerable database system. The database

latent in the IOC's memory is comprised of database definitions and instances of these, called records; in the example application, we found database definitions in the file `exampleApp.dbd` and records in the file `dbExample1.db`. Whereas `exampleApp` consisted of a massive amount of menus and definitions of the record types' fields, `dbExample1` merely declared the record instances—`aiExample`, `calcExample`, and `xxxExample`—and specified their field values. These examples, however, represent only a fraction of the numerous record types that EPICS encompasses.

Through further examination into this example program, we were able to better understand channel access, the software EPICS provides for communication with the IOC on a client-server model. From Solaris, we could start up MATLAB or Probe and change fields in the example databases without logging onto vxWorks, the operating system that the node was running on. To better comprehend vxWorks, we then wrote a simple “hello world” program on Solaris and compiled and ran it on vxWorks.

With the intention of acquiring an interface to GPIB, we built the GPIB support produced by Benjamin Franksen from BESSY [3]. Unfortunately, while examining the driver support for the GPIB instruments, we found that the commands it accepted were specific to each interfaceable device; although continuing to use EPICS to communicate with the supported GPIB devices would be temporarily sufficient, its efficacy would decline in future projects as more software maintenance would be needed to add drivers to this collection of canned interfaces. For this reason, we determined that we would explore alternative paths.

In addition to this discrepancy with our project specification, we found a few other limitations in EPICS as well. The database and record support software did exhibit solid use of abstraction and modularity—a necessity in a system with substantial amounts of code. The

documentation, however, was minimal for the peripheral devices, and there were incongruities as a result of EPICS being a collaborative development of many disparate users. For example, the GPIB package we used was a relatively new version of a GPIB interface and thus was not part of the EPICS base. Even though its driver support for the GPIB we were using, the HPE2050A, had been tested, the module owner, Franksen, noted that the low-level drivers had not been thoroughly tested. In addition, this version did not have proper documentation but instead referred users to documentation from a previous version. After examining the low-level routines, we could see that with negligible commenting in the code, the process of understanding and debugging another programmer's code would be an arduous task. Despite the fact that we chose not to use EPICS altogether, this had been an informative demonstration of the significance of coherent software and adequate documentation.

A GENERIC INTERFACE

With the objective of having an interface that accepts nonspecific commands, we opted to make callable C functions derived from the stand-alone program, `gpib_test` [4], written by David Maden from the Swiss Light Source at the Paul Scherrer Institute. These C functions handled the basic commands including initialization, read, write, clear, and close. Even though their purpose for this project was to support a MATLAB interface, they were still callable directly from a main program; therefore, preserving distinct levels of abstraction was vital to maintaining its flexibility for future applications.

We started by building Maden's stand-alone program, which we were able to test on the GPIB devices. The LAN/GPIB gateway joined the network and the GPIB instruments, so we had to provide a TCP/IP address as well as the GPIB address the instrument occupied on the

gateway. After initialization, the program ran through an interactive loop during which the user could send information, send and receive information from the device, clear, or close the connection. When the oscilloscope or arbitrary waveform generator were connected to the gateway, we could verify that commands were being written by monitoring the response from the device. Using Maden's program as a foundation, we created five separate functions—`open_connection`, `read_string`, `write_string`, `clear`, and `close`. To retain simplicity, we had to make a distinction between what was user interface and what was functional, so the result was an unadorned collection of C functions that were now not being called from an interactive loop.

Because the first use of the functions was to access GPIB instruments from MATLAB, we had to write a gateway routine called a MEX function, C code that would allow C functions to be called as built-in functions [5]. The C functions, known as the computational routines, and their MEX layer yield a MEX file, which is an object file that can be dynamically linked before the MATLAB interpreter loads and executes it. MEX files are most commonly employed when the code can be optimized with C code or when a program has already been written in C; the practicality of writing a MEX wrapper should be assessed in respect to simply writing an equivalent M-function, however.

In MATLAB, all variables can be represented by the `mxArray`, so the gateway routine had to be a MEX function that accepted as arguments two `mxArrays`—one input and one output—along with the number of corresponding arguments to each array. Using the predefined `mxGet` functions, we could convert the data from the `mxArray` into valid C datatypes. For example, in our MEX function, we obtained the first argument—expected to be a command name—by using `mxGetString`. Because MATLAB functions are responsible for checking validity of the argument form, rather than the compiler used by C functions, we used the

mxIsChar function to verify the user had passed in a string. This string was then systematically compared to each expected command—e.g. `read_string`—so the proper operations could be performed. Additional arguments were extracted with the same method and subsequently used in a call to the correct C function in a separate file. When the C function returned, the output argument could be inserted back into an mxArray using one of Matlab's mxGet functions. The computational routines of this file had no knowledge of the MEX layer in the other file, and this MEX wrapper was, in turn, void of all functionality.

DEBUGGING THE INTERFACE

With the initial version of the computational and gateway routine complete, we attempted to build a MEX file. As predicted, we encountered a bulk of errors after compilation. Most of these were syntax errors, though, and could be corrected without complication. When these minute errors were resolved, however, we were confronted with a more obscure problem—link errors. We received a message revealing relocation errors, and a list of variables from the `gpib_test` program's library that were causing the setback. Using the debugger on MEX files is difficult and linking occurs after the process of compiling and assembling, so we first examined the code for any apparent logical errors. When this method failed, we referred back to the original stand-alone program, `gpib_test`, and inspected the differences. We then looked at the makefile used to build the `gpib_test` library and noted that it lacked a compile option, `-fPIC`, that would handle relocation. After this option was added, the two files were linked and loaded properly, producing a mexsol file.

From Matlab, we could call the C functions in the same manner M-functions are called. To initialize a connection, we could make a function call by entering `hook=GPIB('init'`,

'123.45.67.890, '2'), where hook is a variable holding the index that is returned corresponding to the GPIB device; the second input argument is the IP address, and '2' is the address on the GPIB. Following initialization, we could feed it commands to send and receive strings in the form of `status = GPIB('write_string', hook, "<string>")` or `[<return_string>,status]=GPIB('read_string',hook, "<string>")`, where <string> is the instruction sent to the device, and return_string represents the output variable that is assigned the read value from the device. After communication is finished, the connection can be closed by calling GPIB with the hook index returned after opening the connection: `status=GPIB('close, hook)`.

Although we were able to execute these commands, we soon became aware that a proper connection with the device was not being established. The primary indication was the lack of response from the arbitrary waveform generator, but the empty string returned from the read_string function denoted a flaw as well. To locate and resolve the bug, we attempted a variety of approaches, the most straightforward being a simple trace through the code looking for obvious oversights. We began by inserting print statements to pinpoint the exact line where the error occurred. We found that the error occurred when accessing a structure's members, and the function that was performing the initialization was not being completed successfully.

Finding the location of the bug was possible with this simple method; understanding it well enough to fix it, though, entailed a more systematic method. One of the debugging methods we used was to modify our faulty code to function more like the program that was working, gpib_test. On the surface, the static structure used in the C subroutines appeared to be one of the main dissimilarities that was causing errors. After removing the static structure from this file, we still had the same erroneous results. Therefore, we restored the static structure and continued to make the programs even more similar by temporarily disregarding the MEX file. We created

one file with a main function that simply called each function in sequence. Because the MEX wrapper was removed, we could then use the debugger to trace through the program. Moreover, we could employ the debugger to scrutinize the behavior of `gpib_test` and compare their performances. Through this process, we were able to see that the format of the GPIB address argument sent to the initialization function was represented differently in `gpib_test`. The initialization function worked successfully, although the read and write functions still resulted in segmentation faults. By printing out the source of the error, we found that the calls were being timed out. Furthermore, we were able to see through the debugger that a variable controlling the duration of time allowed before a timeout error was being assigned zero, whereas `gpib_test` was assigning it a more reasonable number, 500. With these changes, our stand-alone program ran without complication, and we applied these changes to our C subroutines.

TESTING, REVISION, AND DOCUMENTATION

From Matlab, we tested our MEX file by feeding commands that called the functions `init`, `write_string`, `read_string`, `clear`, and `close` with various instructions selected from the waveform generator's manual. In addition, we called the functions with incorrect arguments and also tested multiple initializations to ensure that multiple connections could be created, with each connection independent and transparent from the others. With the completed program, we created a webpage [6] with an explanation on how to build and use the interface, so users would not have to contend with the obscurities of using a poorly documented application. In addition to documentation, the webpage also includes the source code, the executable binary, a link to the `gpib_test` webpage, and sample MATLAB scripts.

CONFLICTS AND FUTURE APPLICATIONS

Using a running program as a base for our application was advantageous in the coding process; this was nonetheless a challenging project because using and adapting the existing code required understanding of the program, and several inconsistencies with our application caused debugging to be a protracted process. Many programmers work on pre-existing code rather than create their own program from scratch; this is common because the user will often want to use a program, but with slight variations for their own application. Making these modifications would necessitate being familiar with the code, which can be difficult, especially when select portions of the code are hidden from users. For example, while trying to comprehend the reason for the errors, we tried to view some of the low-level functions, `device_read_1` and `device_write_1`, that were producing errors. Their definitions, however, were not available for users to see or alter. Thus although having a working program was convenient and definitely essential, our familiarity with the pre-existing code was not comparable to familiarity we would have had with a program built ourselves.

After testing out commands on a variety of GPIB devices, we began to appreciate the universality of the program. Hopefully, EPICS will incorporate a similar generic program in their system and optimize their application. For now, this project can be used to control the frequency synthesizer for the vibration stabilization project, and later it can be used to communicate with a signal generator to straighten bunchtrains for the FONT project. Although the program was much more modest compared to the massive EPICS, its broad band of applications makes it even more appropriate for current and future projects.

ACKNOWLEDGEMENTS

I would like to thank the U.S. Department of Energy letting me participate in this SULI program. I also thank the NLC department at SLAC for being kind and accommodating.

Finally, thanks to my mentor, Linda Hendrickson, who was generous with sharing her time and knowledge.

REFERENCES

- [1] Advanced Photon Source, “Experimental Physics and Industrial Control System,” <http://www.aps.anl.gov/epics/>, June 23, 2003.
- [2] “Creating <top> Applications,” <http://epics.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/iocScm-3.13.2/creatingTops.html#ApplicationTemplatesSuppliedWithBase>, July 9, 2003.
- [3] B. Franksen, “LanGpib @ BESSY –Information and Download,” <http://www-csr.bessy.de/control/SoftDist/gpib/lanGpib.shtml>, July 9, 2003.
- [4] D. Maden, “HP E2050A LAN/HP-IP Gateway Diagnostic Program,” http://www.sls.psi.ch/controls/help/howto/gpib_test.html, July 10, 2003.
- [5] “External Interfaces/API,” http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/matlab_external.shtml, July 21, 2003.
- [6] L. Hendrickson and S. Chao, “GPiB,” <http://www.slac.stanford.edu/accel/nlc/local/AccelPhysics/codes/gpib/>, August 11, 2003.