VIRTUAL INSTRUMENT BUS USING NETWORK PROGRAMMING

By
Daryl J. Rawnsley

B.S. University of Maine, 1995

A THESIS
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
(in Computer Engineering)

The Graduate School
University of Maine
May 1997

Advisory Committee:

Donald M. Hummels, Associate Professor of Electrical and Computer Engineering, Thesis Advisor

Fred H. Irons, Castle Professor of Electrical and Computer Engineering,

Bruce Segee, Assistant Professor of Electrical and Computer Engineering.

LIBRARY RIGHTS STATEMENT

VIRTUAL INSTRUMENT BUS USING NETWORK PROGRAMMING

By Daryl J. Rawnsley

Thesis Advisor: Donald M. Hummels, Ph.D.

An Abstract of the Thesis Presented

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

(in Computer Engineering)

May 1997

A fundamental problem in a networked computing environment is the orchestrated control of distributed hardware devices. One example is an experiment that uses serveral devices controlled by different host computers. Such a situation can make software development difficult. This thesis addresses the development of a virtual instrument bus that makes the physical hardware configuration transparent to the user. An extensible object oriented library has been developed using instrument buses in the Communications Lab controlled by separate UNIX hosts. This library will greatly facilitate future development of complex experiments requiring multiple bus instrument coordination.

ACKNOWLEDGMENTS

This work has been supported in part by the ARPA HBT/ADC program under a contract administered by the Office of Naval Research Grant N000149311007, and the Army Research Office Grant DAAH04-94-G-0387.

I would like to thank my family and the Communications Lab for their support through this entire project. The Network staff of the Electrical and Computer Engineering Department including Andy Sheaff have been instrumental to the success of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

This thesis presents the development and implementation of instrument control software for use in a networked computer environment. The project was motivated by ongoing research in the Communication Laboratory at the University of Maine. The Communications Lab, among other things, analyzes Analog to Digital (A/D) converter output to provide a means of compensation for the error introduced by the device. Software to support automated tests for data acquisition from A/D test boards has become difficult to maintain as the number of test boards and test instruments grows. A variety of test instruments such as logic analyzers, signal generators, and data caches connect and communicate to workstations using a General Purpose Interface Bus (GPIB). Software to control test instruments that are physically located on separate workstations within the lab, as illustrated in Figure 1.1 is extremely time consuming or impossible to configure. Moving instruments from one workstation to another required reconfiguring software and recompiling an extensive software package.

The software maintenance and network support issues encountered on the Communications Lab are typical of those encountered when instruments are controlled over a computer network. While users have become accustomed to distributed network resources (shared file systems, transparent access to printers, etc) instrument control software has not supported the capabilities of most networks. For a networked computing environment, instrument control software should support the following features:

- Instruments should be portable to any machine on the local network without recompiling test software.
- Test software should not be platform dependent. Tests should operate cor-

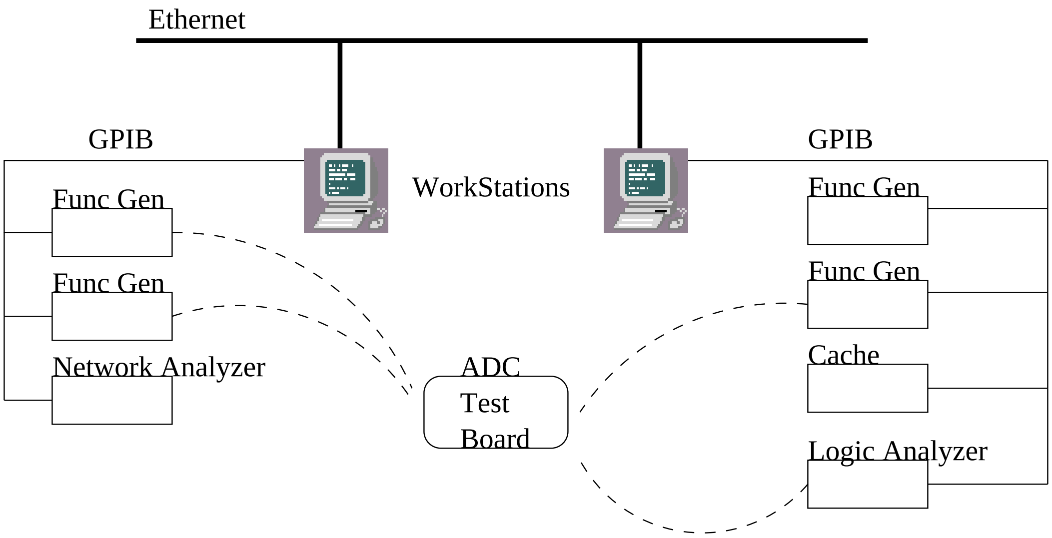

Figure 1.1: Ideal Lab Configuration

rectly regardless of the platform that the test is run from.

- Development of test software should not be platform dependent. Once the network instrument control libraries are compiled for a particular architecture, the test software should be supported for any machine using that architecture.

- The software interface should be consistent regardless of the physical instrument bus interface.

- A common software interface should be provided for instruments with common functionality. For example, all function generators should respond to a common set of amplitude/frequency configuration commands.

1.1 Existing Software

Existing software used in the Lab for data acquisition and controlling instruments is written in the C programming language. The physical addresses of the test instruments and the names of the machine hosts that they are connected to are hard-coded into the software. In order to move instruments from one machine to another, or to change their addresses, the existing software package has to be recompiled for the changes to take effect.

To access instrument software for a specific instrument, the user has to be logged-on to the workstation to which the instrument is physically connected. Automated tests involving multiple instruments connected to different physical buses cannot be supported. In order to incorporate an instrument on a different bus than the one the test is running on, the cabling would have to be physically changed to the new bus. The address of the instrument would have to be set so that it did not conflict with any other instrument on the new bus location, and the software would have to be recompiled to reflect these changes. Setting up for such changes is time consuming and problematic.

With the expansion of our facility to include new high speed instruments for A/D testing, the current setup is not an efficient use of equipment.

1.2 The Approach

The thesis describes two software packages which together address these issues. The first is a "virtual instrument bus" which makes a large number of physical buses on a computer network look like a single bus. The Virtual Instrument Library is designed to support the computer network communications making the computer networking transparent to program developers.

The second software package is an object-oriented instrument library which is specific to instruments within the communications lab. The Communications Lab Library is designed to support a variety of instruments using a common framework in an easily maintained software package. The communication between the libraries is illustrated in Figure 1.2.

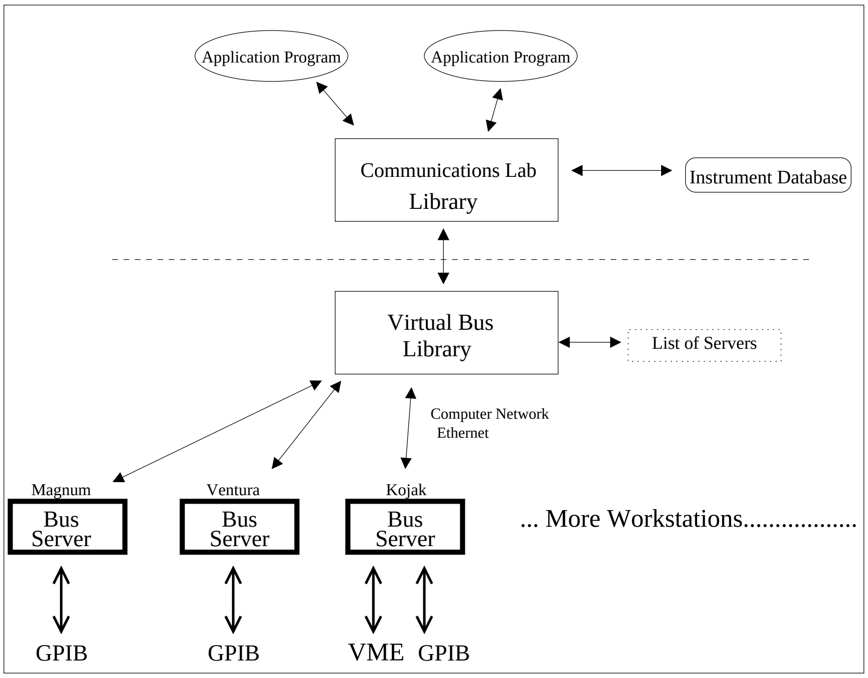

Figure 1.2: Virtual Bus Block Diagram.

1.2.1 Virtual Instrument Library

The virtual instrument library is developed using remote procedure calls (RPC). Chapter 3 gives an overview of the RPC paradigm. RPC is a mechanism for building a distributed system of programs that handle all communications between the physical buses, the workstations, and the network. All workstations supporting an instrument bus run a background program that handles all bus communications (like GPIB) and provides an interface to the computer network. These programs are shown in Figure 1.2 as Bus Servers. The Bus Server can be programmed to communicate with instruments using any kind of bus (not just a GPIB).

Communication to the various Bus Servers is handled by the Virtual Bus Library. This interface makes the physical configuration of the instrument buses transparent to the data acquisition software developer. The library supports a

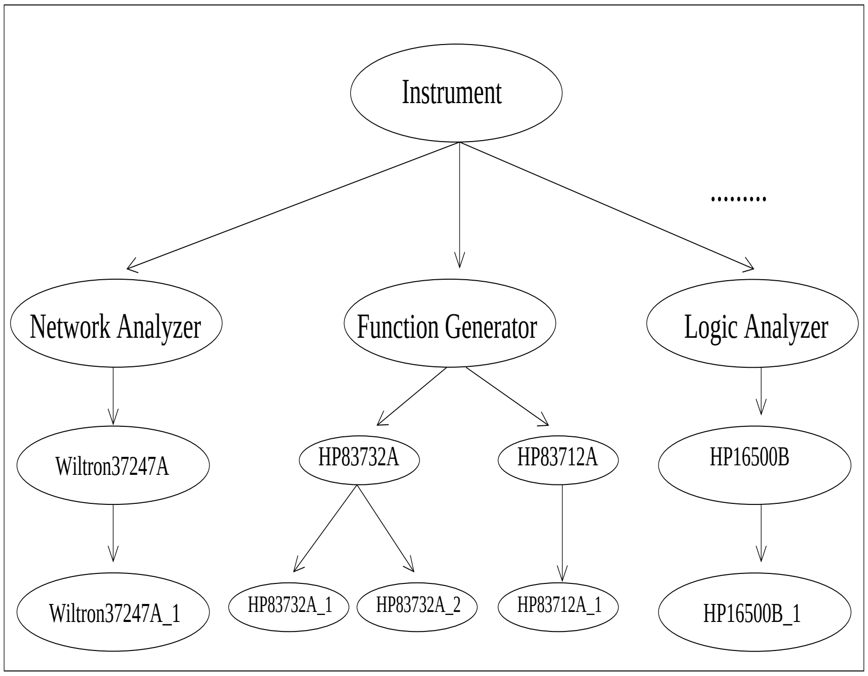

Figure 1.3: Communications Lab Library Software Structure.

small set of routines modeled after the IEEE 488.2 GPIB standard. It also provides searching functions for locating specific instruments on the computer network, and maintains a list of all machines that have instrument buses connected to them. This library handles all communications to the RPC Bus Servers, and is the interface to the computer network for the Communications Lab library.

1.2.2 Communications Lab Library

The Communications Lab library is created using an object-oriented architecture design. It is designed to represent the functionality of the test instruments and provide simplified software reuse and changeability in a modularized fashion. The structure of the library is illustrated in Figure 1.3.

Object-oriented programming is a method of extending abstract data types to allow for relationships among them. In C++ this is accomplished with inheritance. Instead of re-implementing shared characteristics, an object can inherit the functionality of the class it was derived from. The C++ class mechanism allows programmers to define their own data type.

The Communications Lab library uses C++ inheritance extensively. Each level in Figure 1.3 inherits the functionality of the level above it. All test equipment are bus instruments that have common `Instrument` functions. A piece of test equipment, such as a `Function Generator`, has its own common functions to complement the common `Instrument` functions. For example, every `Function Generator` supports a common software interface for controlling the frequency or amplitude of the generator. A specific generator is a Hewlett Packard 83732a which has a variety of functions specific to that model.

The Communications Lab library maintains the bus configuration database shown in Figure 1.2 as the Instrument Database. The database is automatically updated if a change to an instrument's address or location is detected. When one of these instruments, like an `HP83732a`, is used in a program, the library first checks the current location and address in the instrument database. This is done to make sure the program is talking to the correct instrument. If not, search functions of the Virtual Bus library are run to locate the test instrument and update the instrument database of the new test instrument location and address.

The virtual bus software provides easy code reuse for quick program generation used for automated testing, at the same time making all instruments appear to be located on one single bus. This software will greatly facilitate the future development of complex experiments requiring multiple bus instrument coordination.

1.3 Thesis Organization

This thesis paper is organized to provide the reader the proper background information before introducing more advanced topics. The following itemized list outlines the chapters.

- Chapter 2 provides an overview of the entire software project and is meant as a preview for up coming more detailed chapters. The client-server architecture implementation is also reviewed.
- Chapter 3 introduces the concepts of Remote Procedure Calls (RPC) used for distributed computing.
- Chapter 4 explains in detail the implementation of the Virtual Bus Software, including the implementation of Bus Server daemons and Virtual Bus Library function specifications.
- Chapter 5 details the Communication Lab Library Software architecture and design. The object-oriented library contains objects that represent the instruments existing in our lab.
- Chapter 6 explains the structure and purpose of the Instrument Database. This database returns the last known location on any given instrument on the network.
- Chapter 7 discusses software maintenance and proper procedures for supporting this project.
- Finally, chapter 8 reviews the overall results and possibilities of future work.

CHAPTER 2

Virtual Bus Software Architecture

This chapter gives an overview of the software architecture including the names and purposes of the major executables and routines. Figure 2.1 shows the client-server architecture used for the Virtual Bus Software.

2.1 Client Side

The Application Programs are the client side of the architecture. All Application programs use the two software libraries, the Communications Lab Library and the Virtual Bus Library, to create client executables. The Communications Lab Library is an object-oriented library that models types of instruments, and communicates with the Instrument Database Server for current information on instrument locations. The Virtual Bus Library is the interface to network communications. This interface is used by the Communications Lab Library to provide reusable objects for Application Programs.

2.1.1 Virtual Bus Interface

The interface for the virtual bus abstracts away the ideas of network programming from the Communications Lab Library and Application Programs. All interface functions establish connections with specified servers and handle network communications. When completed, each function disconnects from the server. Each function provides an interface that makes it appear to be running locally, when in fact, it may be executing on a different workstation. The following is a brief review of each interface function used on the client side. These functions are defined in detail in Chapter 4.

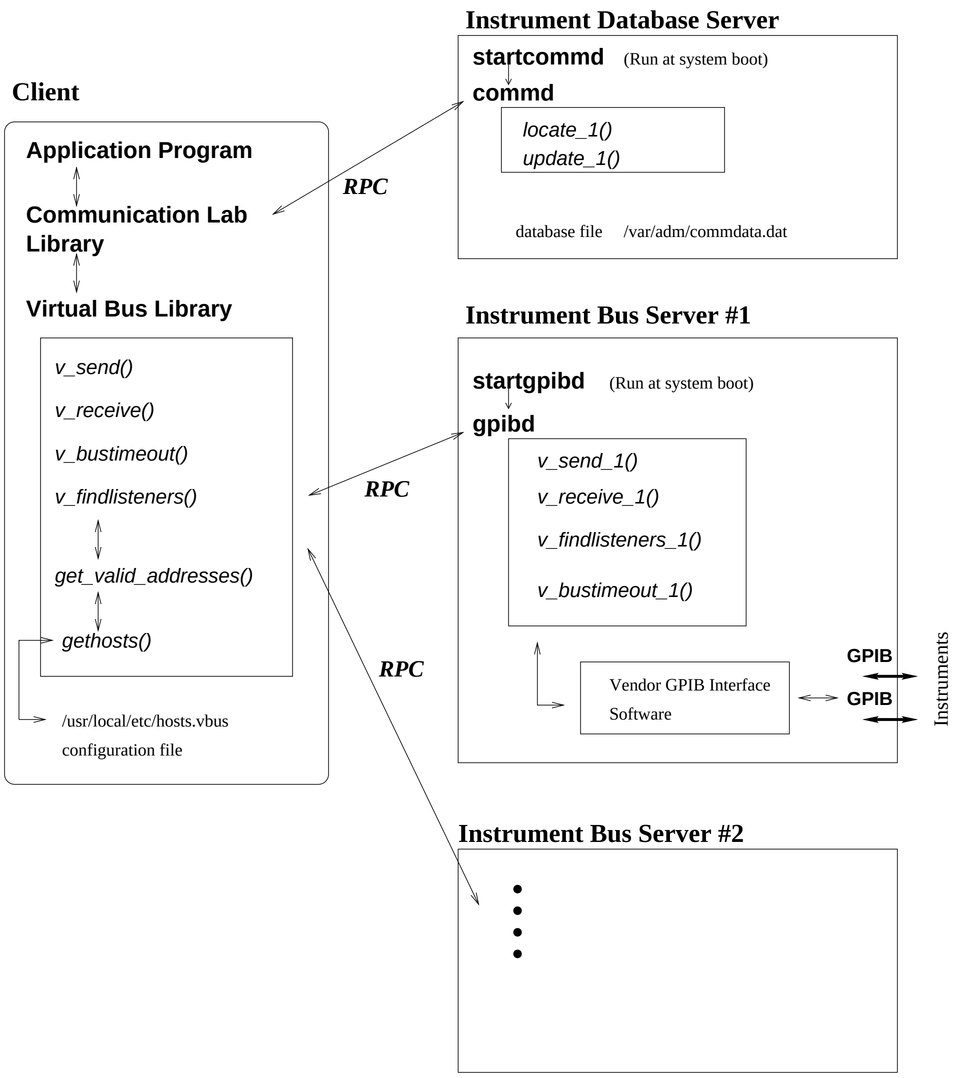

Figure 2.1: Software Architecture

1. **v_send()**: Send commands or data to a specified instrument.

2. **v_receive()**: Receive data from a specified instrument.

3. **v_bustimeout()**: Set the timeout value for the physical bus. The timeout value is the approximate minimum length of time that I/O functions can take before a timeout occurs.

4. **v_findlisteners()**: Poll the bus to find the number of listeners.

There are two helper functions that are used by `v_findlisteners()`:

1. **get_valid_addresses()**: Build a list of addresses for the `v_findlisteners()` function.

2. **gethosts()**: Get a list of host workstations and possible bus addresses from `/usr/local/etc/hosts.vbus` configuration file.

2.2 Server Side

Two different types of servers are used for the virtual bus: the Instrument Server and the Instrument Database Server.

2.2.1 Instrument Server

The Instrument Server, also called the Bus Server, is run as a background process which is configured by the `startgpibd` executable. When this process is started during workstation boot-up, it is replaced with the `gpibd` executable. `gpibd` is the server that handles all client requests to communicate with the instrument bus. When a connection is made, a specific service is performed by calling one of the following functions:

1. **v_send_1()**: Send commands or data to a specified instrument physically connected to the workstation executing the procedure.

2. **v_receive_1()**: Receive data from a specified instrument physically connected to the workstation executing the procedure.

3. **v_bustimeout_1()**: Sets the timeout value for the local bus.

4. **v_findlisteners_1()**: Poll the local bus to find the number of listeners.

Each one of these routines calls vender specific GPIB interface software in order to communicate on the bus.

2.2.2 Instrument Database Server

The Instrument Database Server is run as a background process which is configured by the `startcommd` executable. When this process is started during workstation boot-up, it is replaced with the `commd` executable. The `commd` server handles all client requests for information about location of specific instruments. This server provides two database services:

1. **locate_1()**: Given an instrument identifier, return the last known location of that instrument.

2. **update_1()**: Update the instrument location in the database to the current location.

These database routines store the data in the `/var/adm/commdata.dat` database file.

There may be as many Instrument Servers as there are workstations that have external buses, but only one Instrument Database Server is needed to maintain instrument locations.

The next chapter is an introduction to the concepts of Remote Procedure calls (RPC) and the concepts of distributed computing.

CHAPTER 3

Remote Procedure Calls

Data-communication computer networks provide an alternative to centralized computer facilities. Networking computers allows for exchange of data and sharing of resources between computers on the network. Resources available for one computer can now be made available to other computers via networking. Applications can be structured to distribute their results to other resources on the network.

A computer user should not have to specifically request a resource by explicitly requesting network communications. Software should make the resource location seem transparent to the user by locating the resource without letting the user know the steps the software takes to find it. It should appear to the user that the resource is available on the local system. A system that makes the interface to resources on a network seem transparent is referred to as a distributed system.

Software developers now have a set of routines for specific functions that can be used in generating distributed systems. The Open Network Computing (ONC) platform developed by Sun Microsystems consists of Remote Procedure Call (RPC) and eXternal Data Representation (XDR) routines. This chapter provides an overview of the use of RPCs, and provides an introduction to the implementation of distributed systems.

3.1 Distributed Applications

RPC is based on a procedure call model found in conventional programming languages. Procedures allow programmers to separate issues into smaller more manageable pieces, and provide a model of program execution for the user. The idea is to create procedures that identify and implement different parts of a

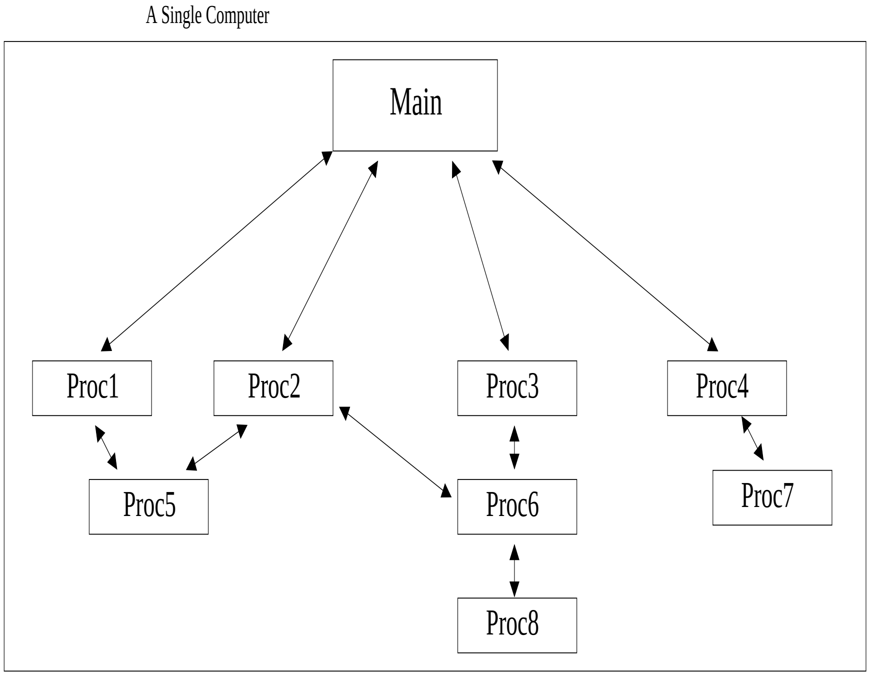

Figure 3.1: The Normal Procedure Concept

program. Figure 3.1 shows the standard procedure concept [1].

Remote procedure calls allow the programmer to extend the procedure model to include more than one computer. Using RPC, programs that are divided into procedures can be executed on different computers to take advantage of resources. Figure 3.2 illustrates the remote procedure model [1].

3.2 Advantages of RPC

The RPC model effectively hides the details for network communications from the programmer. It allows the programmer to concentrate on the problem that the program is trying to solve, and not on details of network programming.

Once a defined set of useful procedures is created, the RPC library can be used to construct building blocks for many distributed applications. These procedures that are available to the network can be viewed as network services. The

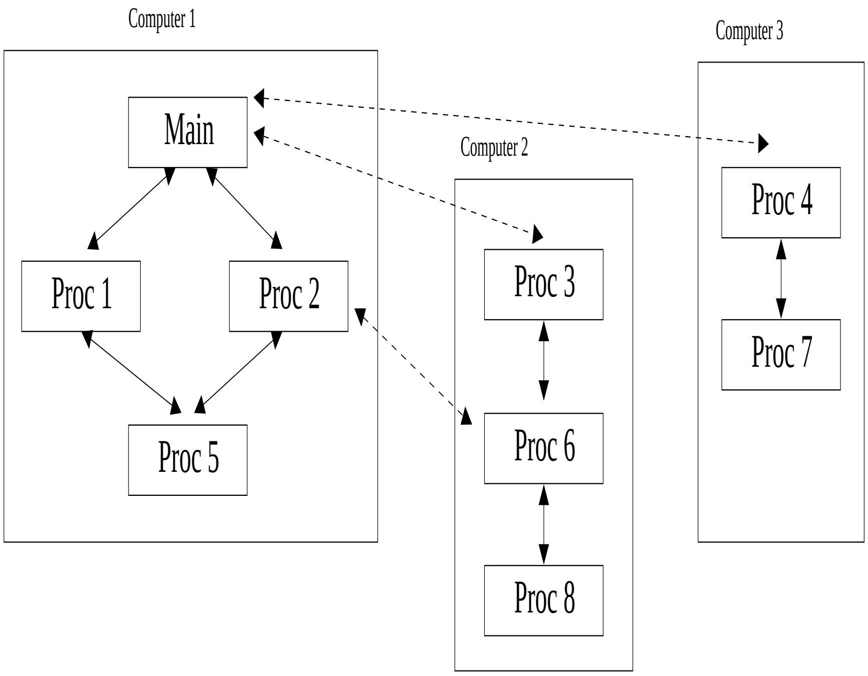

Figure 3.2: Remote Procedure Concept

modular nature of RPC works well with separating the tasks in the program, works well with reuse concepts, and makes software more manageable and maintainable.

3.3 RPC Concerns

The Client-Server model is defined as having a server process that waits for the client process to request a service. RPC uses the same concept, a server program must exist before a client can request service.

Several constraints prevent local and remote procedure calls from behaving identically. A remote procedure call has to wait for the network in order for the program to execute, this can mean several orders of magnitude slower execution than local procedure calls.

Another major concern is that for remote procedure calls, the calling procedure does not share the same address space as the called procedure. The method in which the procedures pass arguments and large amounts of data is much more difficult. Since the program and the procedure live on separate hosts, the procedure has no access to the program's host operating system. Error messages and other information can not be directly displayed to the user from the remote procedure. The programmer must take all of this into account when designing a distributed system.

3.4 eXternal Data Representation (XDR)

Computers do not all store data with the same format. The byte ordering of data can be handled with different representations. Sun Microsystems, Incorporated devised an external data representation that specifies how to represent common forms of data when transferring Data across a network. By way of popularity, XDR is a de facto standard for the description of data in a machine-independent format. XDR addresses two issues in network programming. The first is the issue of data representation. The second is how to describe data structures of arbitrary complexity in a standard way.

The need for uniform data representation can be explained by the following example. Consider a 16-bit integer that is made up of 2 bytes. This value can be stored in two ways: with the low-order byte at the starting address, called little endian, or with the high-order byte at the starting address, called big endian. Figure 3.3 shows the byte representation. Little endian has increasing memory addresses going from right to left in the diagram. The reasoning behind this byte ordering is that a lower address implies a lower order byte. Big endian format shows increasing memory addresses going from left to right. The byte ordering used by some computer systems is shown in Table 3.1 [5].

The incompatibility between storage formats for different hosts makes it difficult to transfer data across a network. The solution to this problem is for

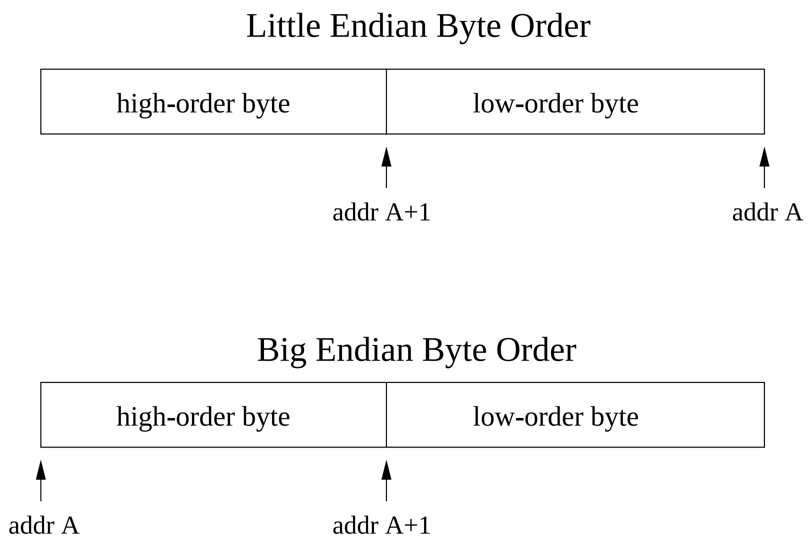

Figure 3.3: Endian Byte Ordering for a 16-bit Quantity

big endian	IBM 370, Motorola 68000, Pyramid
little endian	Intel 80x86 (IBM PC), DEC VAX, DEC PDP-11

Table 3.1: Byte ordering used by different systems.

a network protocol to specify its network byte order. XDR represents data in a manner that is independent of computer architectures, programming languages, and compiler conventions. This makes the data portable. It would be very difficult for a process to have to figure out the data representation for each process that it wanted to communicate with. Each process knows how to convert its local byte format to XDR, and therefore does not have to consider the data format of a process it is sending to. Each process also knows how to convert XDR to its local format.

3.5 RPCGEN - Distributed Programming Tool

Most of the software code needed to handle network communications is repetitive and will not change. To help the programmer, Sun RPC includes a tool that generates much of the code needed to implement a distributed program automatically. This tool is called RPCGEN. RPCGEN reads a specification file created by the programmer and generates files of C source code as its output. The specification file is called a ".X file" and it contains declarations for global data types, constants, global data, and remote procedures. The specification also contains remote procedure arguments and the return value. The resulting files RPCGEN produces are source code for the client and server programs that provide specified remote procedure calls.

The form of a distributed program created using RPCGEN is shown in Figure 3.4. RPCGEN generates basic communication stubs automatically, and the programmer supplies two interface procedures. Called procedures (like PROCB) do not change, only an interface to them is created. A procedure like PROCA is then created to call this interface which uses the communication stubs to execute PROCB.

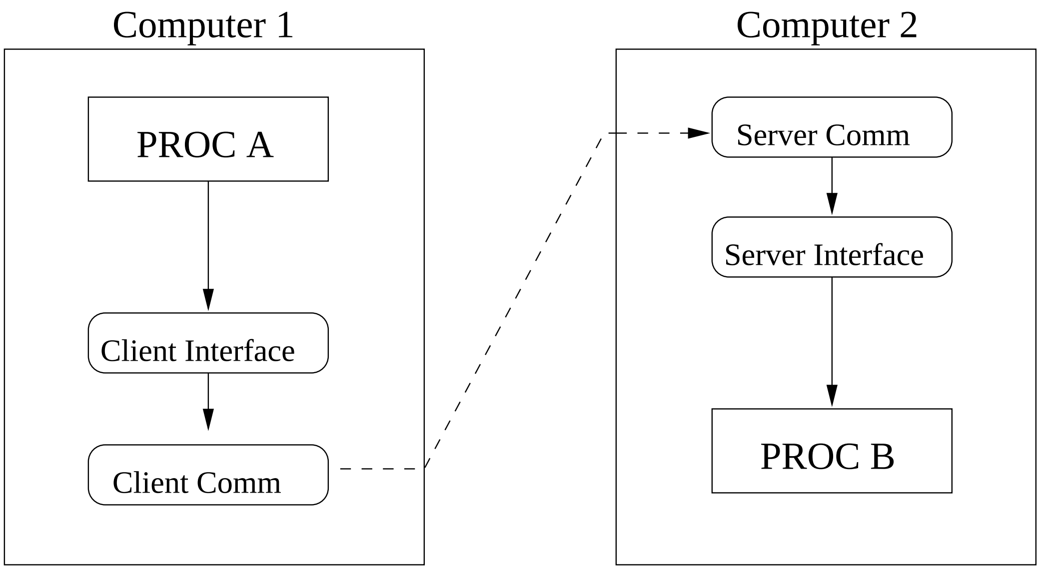

Figure 3.4: RPCGEN distributed program generation.

3.5.1 RPCGEN Example

An example of how RPCGEN is used will help explain these concepts and how they are used. In this example, a procedure running on a different workstation simply outputs a message to the server screen. The message is sent from the local client program and the output procedure is executed by the server program. The remote procedure is defined first and is as follows:

The procedure takes the input string and simply writes the message to the standard output. Then it declares some memory and creates a string to send back to the calling program. Return variables must be declared as `static`. If the variables weren't static or external, but were automatic variables instead, their values would be undefined after the `return` statement passes control back to the server stub that calls the remote procedure.

The memory to store the return string value is allocated dynamically to demonstrate issues involved with dynamic memory allocation using remote procedures. In procedures used for the Bus Server, an unknown size of data storage may need to be allocated (see Chapter 4). The method used in this example is the same as that used for Bus Server procedures. The issue is that dynamic memory allocated within the procedure is normally deallocated at the end of a procedure. When dynamically allocating memory in RPC procedures, if you use a memory block as a return value, deallocating it would mean losing it. E.g., in order to avoid always allocating and never deallocating memory, the first time a pointer to memory is declared `words` it is assigned a `NULL` value. Once the procedure has been run, memory is allocated. When the procedure is repeated, the same memory space is used. If memory space requirements change, it makes sense to deallocate the space used on the previous run before allocating a new block.

With the remote procedure in place, the next step is to create a RPC specification file. The file is called `example.x` as follows.

This file assigns a program number to `exampleprog` of 0x31234569. This number falls within a user defined range of RPC programs, while using a number outside the user defined range could create a conflict with another RPC program such as the Network File System (NFS). Version and procedure numbers are also defined. The version number allows the program developer to improve the code with new versions while maintaining keep compatibility with previous versions.

The rpcgen compiler generates actual remote procedure names by converting the `OUTPUT_MESSAGE` procedure name to `output_message_1`. To do this it converts to lower case and appends an underscore with the version number.

The rpcgen compiler takes the `example.x` file and generates two C files, the client stub code and the server stub code, along with a header file that is

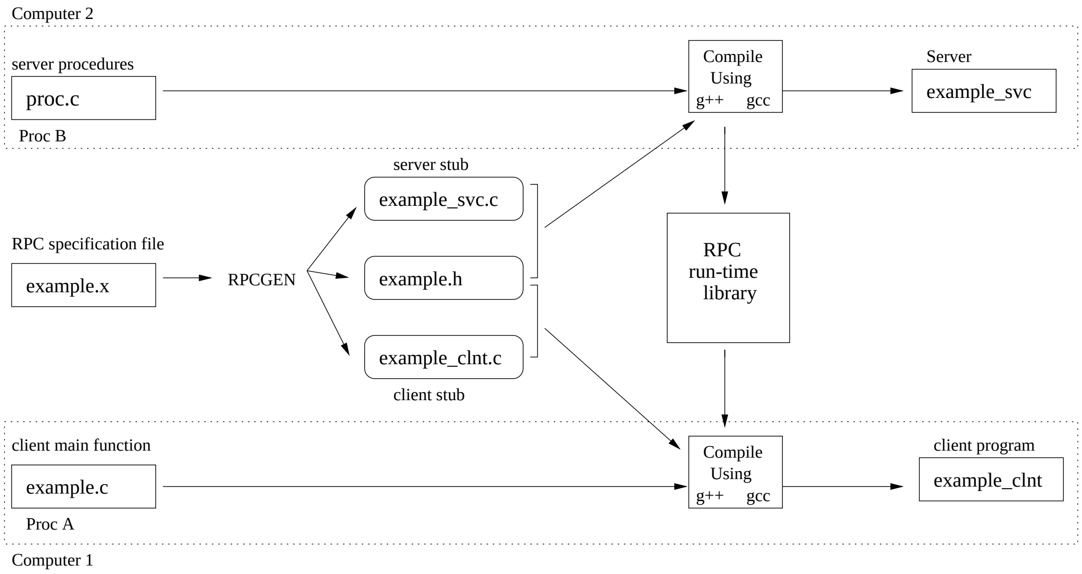

Figure 3.5: RPCGEN distributed program generation example.

included by both stub files. Figure 3.5 illustrates the overall process. The client stub `example_clnt.c` is compiled with the client `main` function to generate the executable client program called `example_clnt`. Much in the same manner, the server program is generated by compiling the server stub `example_svc.c` (which contains the `main` function for the server) along with the server procedure found in `proc.c`.

The `main` function of the client program is where the client makes a connection to the server and calls a remote procedure. This file is created by the developer. Following is the file called `example.c` used in this example.

The program first makes a connection to the server program with the call:

```
if((cl = clnt_create(server,exampleprog, EXAMPLE_VER, "tcp")) == NULL).
```

`cl` is now a handle to the server program, the client program uses this handle to make the remote call to output the message:

```
if ( (sresult = output_message_1(&words,cl)) == NULL)
```

To run this example, we first start our server as a background process.

```
example_svc &
```

Then we invoke a call for service using our client program.

```
example_clnt kojak
```

Where kojak is the workstation name where the server is running. This generates the following output:

From the server

```
allocating memory
this is a test....
```

From the client

```
hi there
```

The next chapter shows how this RPC mechanism was used specifically for for the Virtual Bus Software. The topic of daemon processes and how they work in relation to this project is explained. Finally, the interface functions to the Virtual Bus Library are defined and used in a short example.

CHAPTER 4

Virtual Bus Software

The Virtual Bus Software defines the interface for all network communications. This interface includes a mechanism for calling procedures that makes them appear to be running locally with the calling software, when actually they may be executed on a different workstation. The concerns of network programming are abstracted away from the other software components of the Virtual Instrument Bus. This chapter will discuss two major parts of the Virtual Bus Software: the Bus Server and Virtual Bus Library software. For source code listing see the Virtual Instrument Bus Source Code text [3].

4.1 Bus Server Implementation

Bus Servers run on individual workstations which are physically connected to an instrument bus. They are the server side of the RPC framework. (see Figure 2.1). These programs are run as background processes that only wait for a request from a client to provide a service. Bus Servers implement actual routines that perform all accesses to the GPIB bus. The Bus Servers could be easily configured to use another kind of bus by simply changing the bus routines.

There are four basic GPIB bus procedures that generalize the types of communication that are needed. They are `send`, `receive`, `findlisteners`, and `bustimeout`. The `send` command is used to transmit commands or data to the bus. `Receive` is used to collect data from the bus. The `findlisteners` command reports the address of all devices available to talk on the bus. The `bustimeout` command is used to adjust how long the bus will permit a bus action to occur before it flags an error. This can be used to speed up or slow down how the bus responds to software commands. These procedures are defined in the RPC specification file called vbusrpc.x. The following fragment of that file defines program , version, and procedures names of the RPC software:

All RPC programs are assigned a number that must be unique to the computer network. Since there are other RPC programs on your network, the number needs to be chosen so as not to cause a conflict. The above RPC program name is assigned a number 0x31234567. This number was chosen because it is in the specified range that is reserved for user generated programs [5].

The idea of a version number is to allow the software to be upgraded while maintaining compatibility with previous versions. That means more than one version of your procedures can exist. A version number of 1 is assigned to this program.

All procedures are also assigned a numeric value called procedure numbers. Procedure numbers are used by the RPCGEN client code for identifying a specific procedure.

Notice that all procedures declared in the vbusrpc.x have only a single input argument. RPC only allows a single argument and a single return value. Multiple arguments or multiple return values must be packaged in a C structure. For example, the V_SEND procedure uses a snd structure as an argument and returns a sndreturn. These structures are also defined in the .x specification file vbusrpc.x. The following fragment of that file defines those structures:

When the vbusrpc.x specification file is interpreted by the RPCGEN program, several source files are produced that the programmer incorporates and compiles into a client and a server process. Figure 4.1 shows the files used and created

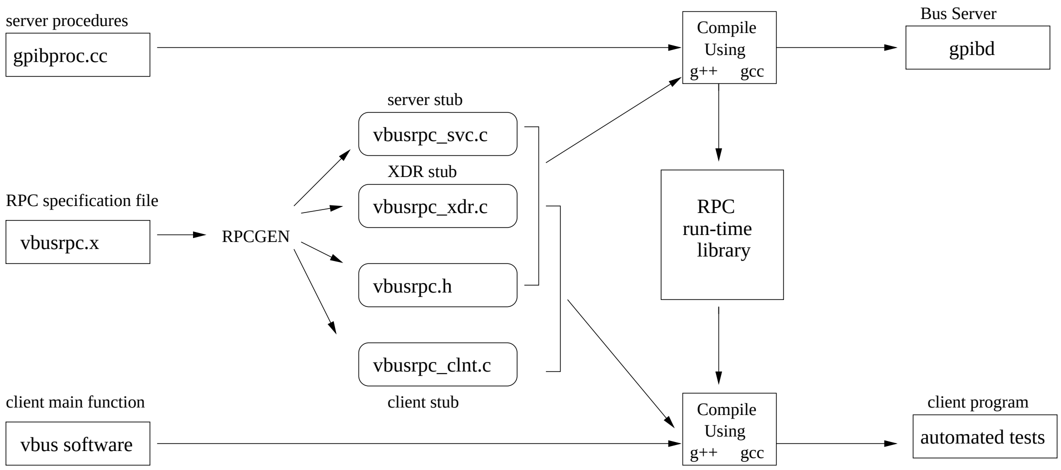

Figure 4.1: RPCGEN generation bus server files and how they are used.

by RPCGEN. These files are then used to create the Bus Server executable, and the interface files used by the Virtual Bus Library. How the Virtual Bus Library uses the generated files is explained more in Section 4.3.

The `gpibproc.cc` file in Figure 4.1 is first compiled as a C++ file and then linked with the compiled `vbusrpc_svc.c` and `vbusrpc_xdr.c` files to form the Bus Server executable. The `vbusrpc_svc.c` contains the interface routines to the `gpibproc.cc` procedures and contains the main function of the Bus Server executable. The `vbusrpc_xdr.c` contains the XDR routines for translating XDR data types into locally C-compatible data structures.

The `vbusrpc_clnt.c` file contains interface code for the client software that calls remote procedures. The Virtual Bus Library uses this interface code for calling remote procedures.

4.2 Bus Server Background Daemon Process

All Virtual Bus commands are handled on UNIX workstations by Bus Servers. Bus Servers therefore need to be available to handle services for their clients at all times the workstations are running. Should a workstation fail and have to be shutdown, the Bus Servers must restart along with all other processes on the workstation.

A daemon process is a process that executes "in the background" (i.e., without an associated terminal or login shell). The daemon is either waiting for some event to occur, or waiting to perform some specified service or task on a periodic basis. Bus Servers need to run in this exact manner so that a Bus Server, running as a daemon process, waits to perform a bus call for the calling program (client).

4.2.1 Daemon Requirements

Because the details of initiating and executing daemon processes can be confusing, this section details a set of standard rules for coding a daemon. The following rules are intended to provide a robust daemon that can be invoked in many different ways [5].

1. **Close All Open File Descriptors**: All unnecessary file descriptors should be closed since they were most likely inherited from the process that started the daemon. This is especially important for standard input, standard output, and standard error file descriptors. File descriptors are closed with the `close()` system command.

2. **Change Current Working Directory**: The kernel holds open the current working directory for a process for the life of the process. The result is that the mounted file system, to which this directory belongs, cannot be unmounted while an active process has its current working directory in that file system. Therefore, a system administrator would not be able to unmount a file system. To avoid this situation a daemon should execute `chdir("/")`

to change its current working directory to the root directory.

3. **Reset the File Access Creation Mask**: Any files created by a daemon process could have their access bits modified by the process that created them. This is because a process inherits its file access creation mask from its parent. To avoid this situation a daemon should execute the `unmask(0)` system call to reset this mask.

4. **Run Process in Background**: The daemon process must run in the background. If a daemon is started from a login session and is not placed in the background when it is invoked (this can be done with the shell's & operator), the daemon will tie up the terminal while it is executing. If the daemon is started from a shell script without being placed in the background, the daemon will cause the processing of the shell script to stop until the daemon finishes. This is common when the daemon is started automatically when the workstation boots (in our case the shell script `/sbin/init.d/local`). Not putting the daemon in the background would cause the workstation to not complete its booting sequence.

   To avoid this, the daemon should do a `fork` and have the parent `exit`, allowing the actual daemon to continue in the child process. The `fork` system call creates a copy of the process that was executing. The process that executed the `fork` is called the parent process and the new process is called the child process. The `exit` system call terminates a process.

5. **Disassociate from the Process Group**: The daemon must also disassociate from the process group. By belonging to some process group, the daemon is susceptible to signals sent to the entire process group. During system boot-up, process groups are being configured and being associated to a group could mean receiving a kill signal that could destroy the daemon

process. Disassociating from the process group and forming its own process group is accomplished with the `setsid` system call.

6. **Ignore Terminal I/O Signals**: Some systems allow you to control the ability of a background process to write output on the control terminal with an `stty` option. (`stty` is a UNIX program that lets the user enable or disable some terminal characteristics.) If a background process attempts to write to a control terminal when writes are not allowed, the process is sent the `SIGTTOU` signal. The result of this signal being sent is that the background process is stopped.

   If the daemon process does want to write error messages on the control terminal when started, it can execute the `signal(SIGTTOU, SIG_IGN)` system call. This means ignore this signal and continue uneffected.

4.2.2 Daemon Processes and RPC

Bus Servers are implemented with RPC. The only software that is provided for the Bus Servers is the procedures they perform. The networking code and the main function of the Bus Server program are provided by RPCGEN.

In order to start the Bus Servers as a daemon process, the rules of section 4.2.1 have to be followed. But most of those steps to start a daemon properly need to be performed within the main function of a program. Since the main function of the Bus Servers is provided by RPCGEN, to do so would mean having to modify RPCGEN created source code files to incorporate these rules. Since RPC source code files are generated automatically, editing RPCGEN created files is not a good solution. If the specification ".x file" were to change, RPCGEN would have to be rerun to handle the new changes. RPCGEN would replace any edits made to the RPC file and daemon would not work without the source code having to be modified whenever RPCGEN was rerun.

A solution to this problem is to have another program that has already performed the steps needed to become a background process, invoke the RPC Bus Server executable. The only way in which a program is executed by UNIX is for an existing process to issue the `exec` system call [5]. The process that issues the `exec` system call is referred to as the calling process, and the program that is `exec`'ed is referred to as the new program. The `exec` call replaces the current process with a new program. This is not a new process, it is really a new program executing in the context of the calling process. A new process is not created, it is really replaced by a different one.

In our implementation when the workstations boot, the shell script `/sbin /init.d/local` is executed. This script invokes the `startgpibd` process which performs the necessary daemon initialization requirements, and then `exec`'s the `gpibd` Bus Server process which remains in the background waiting for a request to provide a service. As a result the RPC specification that creates the `gpibd` Bus Server can change, and not effect how `gpibd` starts as a daemon process.

4.3 Virtual Bus Library Software

The Virtual Bus Library software provides an interface to the virtual bus. The routines in the library abstract away the details of communicating across the network, so that the software that uses this interface does not need to be concerned with the connecting to the server, disconnecting from the server, and XDR data types.

The Communications Lab Library uses these routines as a foundation for providing communication with instruments. Communication between the Virtual Bus and Communications Lab libraries is achieved by a common data object used to share data. The `Address` class data object has all the information needed to locate an instrument. Data members of the `Address` class are as follows.

`server_name`	(char *)The name of the workstation that the instrument is physically attached to.
`device_address`	(int) A number that is the bus address of the instrument.
`bus_id`	(int) The bus identification number. The reason for the bus id is that there may be more than one bus on a given workstation.

The `Address` data structure is required as an input argument for most Virtual Bus Library functions. The Communications Lab Library therefore must maintain the information needed to fill that data structure.

4.4 Virtual Bus Function Specifications

The most important functions of the Virtual Bus Library are discussed in detail in the following sections.

4.4.1 Virtual Send Function

The `v_send()` function sends the desired command (an arbitrary string of bytes) to a desired instrument on the network. It first makes a connection to the Bus Server running on the workstation given in the `Address` data structure. Once connected, the RPC procedure that sends the `command` string out on the bus to the specified instrument address is executed. When completed, the function disconnects from the Bus Server and returns a status value. The return value of this function is -1 unless there has been an error. In that case a GPIB bus error value is returned. Following is the function prototype for the `v_send` function.

```
int v_send(Address addr,
           char *command,
           long count,
           int eotmode);
```

The input parameters have the following meanings.

`addr`	The Bus Server location, the instrument address on the bus, and the bus number.
`command`	A command used to control the addressed specific instrument.
`count`	The size of the command string.
`eotmode`	A flag used to describe the method of signaling the end of data to the listener. Valid values are NULLend = 0x00 which does nothing at the end of a transfer. NLend = 0x01 sends NL with EOI after a transfer. DABend = 0x02 sends EOI with the last data byte.

4.4.2 Virtual Receive Function

The `v_receive()` function acquires data from an instrument on the virtual bus. It first makes a connection to the Bus Server running on the workstation given in the `Address` data structure, and then the RPC procedure that receives information from the bus is called. The data is transmitted over the network and stored in memory where the `data` parameter points to. When the data transfer is completed, the `v_receive()` function disconnects from the Bus Server and returns a status value. The return value is -1 unless there has been an error. In that case a GPIB error value is returned. The prototype for the `v_receive` function is given by the following listing.

```
int v_receive(Address addr,
              char *data,
              unsigned long count,
              int termination);
```

The input parameters have the following meaning.

`addr`	The Bus Server location, the instrument address on the bus, and the bus number.
`data`	A pointer to the location where the data will be stored after completion.
`count`	The maximum size of the data to be received.
`termination`	Specifies the termination char sent to signal the end of data. This value is set to the GPIB 488.2 default STOPend = 0x0100.

4.4.3 Bus Timeout Function

The `v_bustimeout()` function allows the programmer to set the length of time to complete a bus operation before having the controller time-out and end the operation. First, a connection is made to the Bus Server running on the workstation given in the `Address` data structure, and then the RPC procedure that sets the bus time-out is called. When completed, the function disconnects from the Bus Server and returns a status value. The return value of this function is -1 unless there has been an error. In that case a GPIB bus error value is returned. The prototype for the `v_bustimeout()` function is given as follows.

```
int v_bustimeout(Address addr,
                 double  timeoutval);
```

The input parameters have the following meanings:

`addr`	The Bus Server location, the instrument address on the bus, and the bus number.
`timeoutval`	The value that translates to the time-out value in seconds. If the bus time-out value cannot be set to the exact time, then the next highest valid time is chosen.

4.4.4 Find Listeners Function

The `v_findlisteners()` function is a searching function used for locating all listeners on a given instrument bus. First, a connection is made to the

Bus Server running on the workstation the software is querying. Then the RPC procedure that polls the bus for the addresses of all listeners is executed. When completed the function disconnects from the Bus Server and returns a status value. The return value of this function is -1 unless there has been an error. In that case a GPIB bus error value is returned. The prototype for the `v_findlisteners()` function is as follows.

```
int  v_findlisteners(Address* addr,
                     int sizeofaddrlist,
                     Address *resultlist,
                     int& numreturned);
```

The input parameters have the following meanings:

`addr`	An array of possible address values that should be checked for valid listeners.
`sizeofaddrlist`	The size of the address list that contains possible address values that should be checked for valid listeners.

The last two input parameters are used to return data.

`resultlist`	The resulting list of `Address` data structures that have listeners on the bus. The storage for this data should be as large as the `sizeofaddrlist` input parameter because there are that many possible addresses that may have listeners available. The storage for this list is also maintained by the user.
`numreturned`	The number of listeners in the `resultlist`.

4.4.5 Get Valid Addresses Function

The `get_valid_addresses()` function gets the list of all addresses that the virtual bus is connected to. This list is then used to check for listeners at those addresses using the `v_findlisteners` function. The input value will point to the list of possible addresses when the call is complete. The return value is the number of addresses contained in the returned list. The prototype for the get_valid_addresses() function is as follows.

```
int get_valid_addresses(Address ** validaddrs);
```

The input parameter has the following meaning.

validaddrs	A pointer to a list of valid addresses that can be checked for valid listeners. On return, this pointer will be set to the location of the address list. The function allocates memory for the list. It is the caller's responsibility to provide space for this storage.

4.4.6 Get Hosts Function

The gethosts() function gets a list of valid hosts and addresses from a text file /usr/local/etc/hosts.vbus. It's used by the get_valid_addresses() function to help generate a list of addresses to be checked. The input value points to the list of hosts when the call is complete. The return value is the number of hosts contained in the returned list. The prototype for the gethosts() function is as follows.

```
int gethosts(Hosts** h);
```

The input parameter has the following meaning.

h	A pointer to a list of Hosts data structures that contain a server name, bus number, starting and ending addresses read from the hosts.vbus file. On return, this pointer is set to the host list location The function allocates memory for the list. It is the caller's responsibility to this storage.

4.5 Virtual Bus Example Program

The following example shows how the virtual bus functions of `v_send` and `v_receive` to ask a GPIB instrument to identify itself.

First an `Address` object is defined to hold the location of the instrument. Then the end of transmission mode for the GPIB bus is specified. End of transmission mode is a flag used by the GPIB bus to describe the method of signaling the end of data to the instrument receiving the data. Next the `v_send` function then sends the identification query command to the instrument defined by the `Addr` object.

Once the message is sent, the `v_receive` command is used to retrieve the instrument's response. Data storage is created for the result, and the termination character for receiving on the GPIB bus is set. The termination value is used to signal the end of data.

The above example showed how to call virtual bus functions directly to communicate with an instrument. The next example shows how virtual bus functions are used to create a routine used to locate the address of an instrument before trying to send data to it.

First a pointer to an `Address` object is created called `valid_addrs`. The function `get_valid_addresses` is then called using this pointer and returns the number of addresses that the `valid_addrs` pointer now points to.

Now a new list of `Address` objects is created to store the a list of addresses that will be returned when the `v_findlisteners` is called. This list is called search_list. The valid_addrs list is also passed to the v_findlisteners function with the number of addresses in this list num_addresses_valid. The newly allocated search_list contains the valid listeners when the call is completed. The numreturned argument contains the number of addresses pointed to by the search_list.

At this point all the valid listeners on the network have been found, and this list could be used to check for a specific instrument. After using the list, its memory storage must be deleted to avoid memory losses, and memory created by the get_valid_addresses function must also be deleted.

CHAPTER 5

Communications Lab Software

The Communications Lab Library software is written using the C++ programming language, and it is developed to take advantage of object-oriented structures. This chapter explains how the library is created using C++ features and then defines object interfaces that have been created. For source code listing see the Virtual Instrument Bus Source Code text [3].

5.1 Communications Lab Library Architecture

The Communications Lab Library software is designed using an object-oriented approach. Each object within this library represents an instrument at some level. In C++ the class mechanism allows users to define their own data types to represent objects. The C++ class has many attributes, the two most important attributes discussed here are data members and member functions. Data members represent the data an object needs. There may be zero or more data members of any type in a class. Member functions are referred to as the class interface. Member functions are the set of operations that may be applied to objects of that class. There may be zero or more member functions for a class.

Object-oriented programming extends abstract data types to allow for type-/subtype relationships [2]. This is accomplished through a mechanism referred to as inheritance. Rather then re-implementing shared characteristics, a class can inherit selected data members and member functions of other classes [2]. Objects can inherit functionality only from objects they are inherited from.

The concept of instruments as objects maps nicely into an inheritance hierarchy. Figure 5.1 illustrates a sub-portion of the `Instrument` family derivation. This inheritance hierarchy is referred to as single inheritance. With this type of

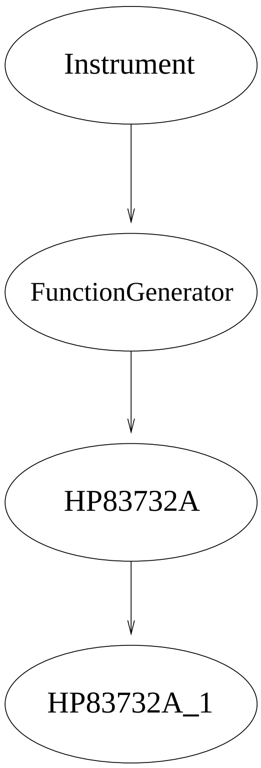

Figure 5.1: Single Inheritance

hierarchy, each derived class has only one immediate base class.

The design starts with the `Instrument` base class. An `Instrument` object has a simple interface that all instruments on the virtual bus will need. It has a bus address data member to keep track of the `Instrument`'s location. An `Instrument` object also needs to send and receive data and so those member functions are provided. With this as a base class, all objects that inherit from the `Instrument` object will already have that functionality built in.

The `FunctionGenerator` object is then defined to inherit from an `Instrument` object. This is desirable because a `FunctionGenerator` is a type of `Instrument` and therefore will need to do things like send and receive on the virtual bus. The software mechanism for sending and receiving has already been implemented by the `Instrument` class.

A `FunctionGenerator` object defines the standard interface for all function generators. All function generators must conform to this interface. This means that any object that inherits from a `FunctionGenerator` object must provide some standard member functions as an interface. For example, all function generators must implement a standard `set_freq()` function which is used to control the output signal frequency. By using inheritance, different types of function generators may have different implementations of the `set_freq()` function, depending upon the specific GPIB commands required for that instrument. However, inheritance ensures a standard interface for all the different kinds of function generators. Similar interfaces are provided for setting the amplitude, or reading the frequency and amplitude from the generator.

The next object defined in Figure 5.1 is a specific brand of function generator called `HP83732A`. The `HP83732A` is a vendor specific type of function generator that defines its own implementation for setting frequency, amplitude , and other operations, such as modulation modes . Once these functions are provided, a specific function generator can be defined to inherit these operations. The `HP83732A_1` object represents the physical object connected to the bus. If there are several `HP83732A` instruments on the bus, other specific function generators would be defined to represent them using the naming convention `HP83732A_2`, `HP83732A_3`, etc.

5.2 Advantages of Inheritance in The Library

One of the most important advantages to inheritance architecture is code reuse. Adding a new type of function generator to the bus only means implementing required functions to inherit from a `FunctionGenerator` object. Sending and receiving on the bus are already implemented and can just be reused.

Another important advantage is the ability to abstract away details of lower level objects in other routines. Application programs can be written without regard for the particular instrument type. For example, calling the standard `set_freq()` member function can be done without knowing which instrument is actually going to perform it. If an application program is defined to accept a `FunctionGenerator` object as an input, the routine knows that the function generator that was passed into the routine can do some standard things (set a frequency and amplitude). It does not matter which specific type of function generator was given, the functionality of a function generator will not change.

5.3 Instrument Classes Defined

The following sections contain detailed information about the functionality of the objects contained in the Communication Lab Library, the interface to each object, and how it is used.

5.3.1 Instrument Class

Description:

Defines the common communication routines for all instrument types that will inherit from it. This is the base class for all Communication Lab instruments.

Inheritance:

None

Constructors:

```
Instrument(void);
```

Set data members to default values.

```
Instrument(char *server, int device_addr, int busid, int
term=0x0100, int eotmode=0x01);
```

Sets the Address data member to argument values.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation a bus identification argument is needed.
`term`	See the `set_termination` access function p. 42.
`eotmode`	See the `set_eotmode` access function p. 42.

Member Functions:

```
void send(char *command, long count);
```

Send the command to the virtual bus. The Address data member value must already be specified.

`command`	The specific command.
`count`	The number of characters in the command.

`void receive(char *data, unsigned long count);`

Receive data from the virtual bus. The Address data member value must already be specified.

`data`	The storage space allocated for the data.
`count`	The size of the allocated data.

`void timeout(double timeoutval);`

Set the timeout value for the bus.

`timeoutval`	Value listed in seconds.

`virtual int check(void);`

Defines a check routine that checks if a listener is located at the Address data member address. A positive return value means the instrument is present.

`virtual int reset(void);`

Sends the GPIB 488.2 reset command "*RST" to the Address data member address. The return value of this function is -1 unless there has been an error. In that case a GPIB bus error value is returned.

Access Functions:

`int set_address(const Address& a);`

Sets the Address data member to the given value.

`a`	Address class has a server name, device address, and bus identification number.

```
Address get_address(void);
```

Returns the current Address data member value.

```
int set_termination(int termchar);
```

Sets the termination character for the Receive member function. The termination is a value used to describe the method of signaling the end of the data.

`termchar`	If it is a value between 0 and hex 00FF, the ASCII character with the corresponding hex value is considered the termination character, and the read is stopped when the character is detected. If the termination character is the constant 0x0100, the read is stopped when END is detected.

```
int get_termination(void);
```

Returns the current termchar data member value.

```
int set_eotmode(int eotmodeval);
```

Set end of transmission value for the Send member function.

`eotmodeval`	This is a flag used to describe the method of signaling the end of the data to the Listener. These constants are described in the `v_send` section on p. 29.

```
int get_eotmode(void);
```

Returns the current eotmode (end of transmission value).

5.3.2 Function Generator Class

Description:

Defines the interface that all types of function generators must satisfy.

Inheritance:

Instrument p.40

Constructors:

```
FunctionGenerator(double freq=0, double amp=0) :
Frequency(freq), Amplitude(amp), Instrument();
```

Set data members to default values and call the Instrument class constructor.

`freq`	Value in Hz.
`amp`	Value in Volts peak.

```
FunctionGenerator(char *server,int device_addr, int
busid, double freq=0, double amp=0) :  Frequency(freq),
Amplitude(amp), Instrument(server,device_addr,busid);
```

Sets data members to default values and creates an Instrument object with specified Address values.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.
`freq`	Value in Hz.
`amp`	Value in Volts peak.

Member Functions:

```
virtual void set_freq(double freq) = 0;
```

This is a pure virtual member function that must be defined by all objects that inherit from the Function Generator class. This member function will set the output frequency value of the instrument.

`freq` Value in Hz.

`virtual double get_freq(void);`

Returns the value of the Frequency data member. This function should be redefined by derived classes to query the instrument for its current frequency value. By default, this function returns the value of the data member `Frequency`.

`virtual void set_ampl(double amp) = 0;`

This is a pure virtual member function that must be defined by all objects that inherit from the Function Generator class. This member function will set the output amplitude value of the instrument.

`amp` Value in Volts peak.

`virtual double get_ampl(void);`

Returns the value of the Amplitude data member. This function should be redefined by derived classes to query the instrument for its current amplitude value. By default, this function returns the value of the data member `Amplitude`.

5.3.3 HP83712A Class

Description:

This class implements the Function Generator class interface for the Hewlett Packard 83712A function generator.

Inheritance:

Instrument p.40, Function Generator p.42

Constructors:

`HP83712A() FunctionGenerator();`

Call the `FunctionGenerator` constructor.

```
HP83712A(char *server, int device_addr, int busid)
FunctionGenerator(server,device_addr,busid);
```

Creates a `FunctionGenerator` object with specified `Address` value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation a bus identification argument is needed.

Member Functions

`virtual int check(void)`

Check function must be implemented for inheriting classes to provide a mechanism of communication to the correct instrument. A positive return value means the instrument is present.

`void set_freq(double freq);`

Set the output frequency value of the instrument.

`freq`	Frequency in Hz. The value is limited to 0.01 GHz - 20 GHz.

`void set_ampl(double amp);`

Set the output amplitude value for the instrument.

`amp`	Value in Volts peak.

`double get_freq(void);`

Query the device for the output frequency value. Returns the current frequency in Hz.

`double get_ampl(void);`

Query the device fot the output amplitude value. Returns the current amplitude in Volts peak.

5.3.4 HP83712A_1 Class

Description: This class is a representation of a specific Hewlett Packard 83712A instrument.

Inheritance:

Instrument p.40, Function Generator p.42, HP83712A p.44

Constructors:

`HP83712A_1();`

This default constructor queries the database for the last known location of the HP83712A_1 function generator. If this address is incorrect searching can occur.

`HP83712A_1(char *name, int dev, int id);`

This constructor attempts to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

`int check(void);`

Check to see if the current address specified in the Instrument class is for this function generator by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.5 HP83732A Class

Description:

This class implements the Function Generator class interface for the Hewlett Packard 83732A function generator.

Inheritance:

Instrument p.40, Function Generator p.42

Constructors:

`HP83732A(): FunctionGenerator();`

Call the `FunctionGenerator` constructor.

```
HP83732A(char *server, int device_addr, int busid):
FunctionGenerator(server,device_addr,busid);
```

Creates a `FunctionGenerator` object with specified `Address` value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions

`virtual int check(void)`

Check function must be implemented for inheriting classes to provide a mechanism of communication to the correct instrument. A positive return value means an instrument has been found.

`void set_freq(double freq);`

Set the output frequency value of the instrument.

`freq`	Frequency in Hz. The value is limited to 0.01 GHz - 20 GHz.

`void set_ampl(double amp);`

Set the output amplitude value for the instrument.

`amp`	Value in Volts peak.

`double get_freq(void);`

Query the device for the output frequency value. Returns current frequency in Hz.

`double get_ampl(void);`

Query the device fot the output amplitude value. Returns current amplitude in Volts peak.

5.3.6 HP83732A_1 Class

Description: This class is a representation of a specific Hewlett Packard 83732A instrument.

Inheritance:

Instrument p.40, Function Generator p.42, HP83732A p.47

Constructors:

`HP83732A_1();`

This default constructor queries the database for the last known location of the HP83732A_1 function generator. If this address is incorrect searching can occur.

`HP83732A_1(char *name, int dev, int id);`

This constructor attempts to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

`int check(void);`

Check to see if the current address specified in the Instrument class is for this function generator by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.7 HP8645A Class

Description:

This class implements the Function Generator class interface for the Hewlett Packard 8645A function generator.

Inheritance:

Instrument p.40, Function Generator p.42

Constructors:

`HP8645A(): FunctionGenerator();`

Call the `FunctionGenerator` constructor.

```
HP8645A(char *server, int device_addr, int busid):
FunctionGenerator(server,device_addr,busid);
```

Creates a `FunctionGenerator` object with specified `Address` value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions

`virtual int check(void)`

Check function must be implemented in inheriting classes to provide a mechanism of communication to the correct instrument. A positive return value means an instrument has been found.

`void set_freq(double freq);`

Set the output frequency value of the instrument.

`freq` Frequency in Hz. The value is limited to 251 kHz - 1030 MHz.

`void set_ampl(double amp);`

Set the output amplitude value for the instrument. Amplitude value is limited to +16 to -137 dBm.

`amp` Value in Volts peak.

`double get_freq(void);`

Query the device for the output frequency value. Returns current frequency in Hz.

`double get_ampl(void);`

Query the device for the output amplitude value. Returns current amplitude in Volts peak.

5.3.8 HP8645A_1 Class

Description: This class is a representation of a specific Hewlett Packard 8645A instrument.

Inheritance:

Instrument p.40, Function Generator p.42, HP8645A p.50

Constructors:

`HP8645A_1();`

This default constructor queries the database for the last known location of the HP8645A_1 function generator. If this address is incorrect searching can occur.

`HP8645A_1(char *name, int dev, int id);`

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int check(void);
```

Check to see if the current address specified in the Instrument class is for this function generator by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.9 HP8642A Class

Description:

This class implements the Function Generator class interface for the Hewlett Packard 8642A function generator.

Inheritance:

Instrument p.40, Function Generator p.42

Constructors:

```
HP8642A(): FunctionGenerator();
```

Call the `FunctionGenerator` constructor.

```
HP8642A(char *server, int device_addr, int busid):
FunctionGenerator(server,device_addr,busid);
```

Creates a `FunctionGenerator` object with specified `Address` value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions

`virtual int check(void)`

Check function must be implemented in inheriting classes to provide a mechanism of communication to the correct instrument. A positive return value means an instrument has been found.

`void set_freq(double freq);`

Set the output frequency value of the instrument.

`freq`	Frequency in Hz. The value is limited to 100 kHz - 1057.5 MHz.

`void set_ampl(double amp);`

Set the output amplitude value for the instrument. Amplitude value is limited to 0.023 - 2.24 V-RMS.

`amp`	Value in Volts peak.

`double get_freq(void);`

Query the device for the output frequency value. Returns current frequency in Hz.

`double get_ampl(void);`

Query the device fot the output amplitude value. Returns current amplitude in Volts peak.

5.3.10 HP8642A_1 Class

Description: This class is a representation of a specific Hewlett Packard 8642A instrument.

Inheritance:

Instrument p.40, Function Generator p.42, HP8642A p.53

Constructors:

`HP8642A_1();`

This default constructor queries the database for the last known location of the HP8642A_1 function generator. If this address is incorrect searching can occur.

`HP8642A_1(char *name, int dev, int id);`

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int check(void);
```

Check to see if the current address specified in the `Instrument` class is for this function generator by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.11 Logic Analyzer Class

Description: Defines the interface that all types of logic analyzers must satisfy.

Inheritance:

Instrument p.40

Constructors:

```
LogicAnalyzer(void): Byte1(-1), Byte2(-1), Byte3(-1),
Byte4(-1), Logic_family(1),Clock_number(0),Clock_edge(1),
Instrument();
```

Provide default values for data members and create an `Instrument` object.

```
LogicAnalyzer(char *server,int device_addr, int
busid) : Byte1(-1), Byte2(-1), Byte3(-1), Byte4(-1),
Logic_family(-1),Clock_number(0),Clock_edge(-1),
Instrument(server,device_addr,busid);
```

Provide default values for data members and create an Instrument object with the following parameters.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

`virtual int trigger(void) = 0;`

This is a pure virtual member function that must be defined by all objects that want to inherit from a Logic Analyzer. This member function will trigger the device for data acquisition. The return value is meant to return an error value.

`virtual int podselect(int podbyte4, int podbyte3, int podbyte2, int podbyte1) = 0;`

This is a pure virtual member function that must be defined by all objects that want to inherit from a Logic Analyzer. This member function selects up to four pods to be used for data acquisition. The order in which the pods are selected determines the byte order of the integer data that is returned.

`podbyte4`	The pod number that is selected here will have its data placed in the fourth byte in the returned integer.
`podbyte3`	The pod number that is selected here will have its data placed in the third byte in the returned integer.
`podbyte2`	The pod number that is selected here will have its data placed in the second byte in the returned integer.
`podbyte1`	The pod number that is selected here will have its data placed in the first byte in the returned integer.

`virtual int getdata(int *data, int npts) = 0;`

This is a pure virtual member function that must be defined by all objects that want to inherit from a Logic Analyzer. This member function acquires data from the Logic Analyzer.

`data`	The data acquired from the Logic Analyzer is returned in this array. Storage for this array must already exist.
`npts`	The size of the data to be acquired.

Access Functions

`void set_logic_family(int type);`

Set the logic family for data acquisition.

`type`	Two types of logic families are 0=ECL, 1=TTL.

`void set_clock_number(int num);`

Set the clock number.

`num`	An integer determining the clock to be used. The interpretation of num depends on the type of logic analyzer.

`void set_clock_edge(int edge);`

Set the clock edge type for data acquisition.

`edge`	The clock edge can have two values, 0=NEG negative edge triggered, 1=POS positive edge triggered.

5.3.12 HP16500B Class

Description: This class implements the Logic Analyzer class interface for the Hewlett Packard 16500B Logic Analyzer.

Inheritance:

Instrument p.40, LogicAnalyzer p. 56

Constructors:

`HP16500B(void):LogicAnalyzer();`

Calls constructor for a LogicAnalyzer object.

```
HP16500B(char *server, int device_addr, int busid) :
LogicAnalyzer(server, device_addr, busid);
```

Calls the constructor for a LogicAnalyzer with a specified Address value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there maybe one or more buses connected to a workstation a bus identification argument is needed.

0	0	Pod2	Pod1
Byte4	Byte3	Byte2	Byte1

```
HP16500B_1 logic;
int errorval = logic.podselect(-1,-1,2,1);
```

Figure 5.2: Podselect example

Member Functions

`int podselect(int podbyte4, int podbyte3, int podbyte2, int podbyte1);`

Pod selection defines the order of bytes of return data when calling getdata. Since getdata returns integer values, the order of the four bytes in an integer is determined by this function. Higher order bytes that will not represent valid data must have a "-1" value used as an argument. These byte values will be returned as 0. Figure 5.2 shows an example. Pod selection must be used before the getdata member function.

`podbyte4`	The pod number that is selected here will have its data placed in the fourth byte in the returned integer.
`podbyte3`	The pod number that is selected here will have its data placed in the third byte in the returned integer.
`podbyte2`	The pod number that is selected here will have its data placed in the second byte in the returned integer.
`podbyte1`	The pod number that is selected here will have its data placed in the first byte in the returned integer.

`int trigger(void);`

This trigger function first selects slot 4 of the HP16500B as the master for data acquisition. All pods 0-5 are selected for use and labeled. The logic family is set. (Use Logic Analyzer member function p.56) The HP16500B is then ready to get data. The trigger function must be used before the getdata member function.

`int getdata(int *data, int npts);`

Getdata acquires data from the HP16500B, and stores the result in the data argument according to the Podselect member function. The storage for the data argument must already be created. The npts argument defines the amount of data that is acquired.

`data`	The data acquired from the Logic Analyzer is returned in this array. Storage for this array must already exist.
`npts`	The size of the data to be acquired.

5.3.13 HP16500B_1 Class

Description: This class is a representation of a specific Hewlett Packard 16500B instrument.

Inheritance:

Instrument p.40, Logic Analyzer p.56, HP16500B p.58

Constructors:

`HP16500B_1();`

This default constructor queries the database for the last known location of the HP16500B_1 function generator. If this address is incorrect searching can occur.

`HP16500B_1(char *name, int dev, int id);`

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int check(void);
```

Check to see if the current address specified in the Instrument class is for this logic analyzer by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.14 Tektronix9504 Class

Description: This class implements the Logic Analyzer class interface for the Tektronix 9504 Data Cache.

Inheritance:

Instrument p.40, LogicAnalyzer p. 56

Constructors:

```
TEKTRONIX9504() : Record_length(8192), LogicAnalyzer();
```

Calls constructor for a LogicAnalyzer object and sets the Record_Length data member to a default value.

```
TEKTRONIX9504(char *server, int device_addr, int busid)
: Record_length(8192), LogicAnalyzer(server, device_addr,
busid);
```

Calls the constructor for a LogicAnalyzer with a specified Address value, and set the Record_Length data member to a default value.

Podbyte Value	TEKTRONIX9504 Bus Selected
0	Channel 1, LSB
1	Channel 1. MSB
2	Channel 2, LSB
3	Channel 2, MSB

Table 5.1: TEKTRONIX9504 Podselect Byte Ordering

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions

```
int podselect(int podbyte4, int podbyte3, int podbyte2, int
podbyte1);
```

Pod selection defines the order of bytes of the return data when calling getdata. Since getdata returns integer values, the order of the four bytes in an integer is determined by this function. Pod selection must be used before the getdata member function. The Tektronix 9504 has two channels, Table 5.1 shows which pods go with which channel. The Tektronix 9504 can trigger one channel at a time for a maximum 16 bits of data.

`podbyte4`	The pod number that is selected here will have its data placed in the fourth byte in the returned integer.
`podbyte3`	The pod number that is selected here will have its data placed in the third byte in the returned integer.
`podbyte2`	The pod number that is selected here will have its data placed in the second byte in the returned integer.
`podbyte1`	The pod number that is selected here will have its data placed in the first byte in the returned integer.

`int trigger(void);`

This trigger function determines the channel being triggered by the pod values that are selected by the podselect member function for data acquisition. The logic family is then set. (Use Logic Analyzer member function to change the logic family value). p.56) The Tektronix 9504 is then ready to get data. The trigger function must be used before the getdata member function.

`int getdata(int *data, int npts);`

Getdata acquires data from the Tektronix 9504, and stores the result in the data argument according to the Podselect member function. The storage for the data argument must already be created. The npts argument defines the amount of data that is acquired.

`data`	The data acquired from the Logic Analyzer is returned in this array. Storage for this array must already exist.
`npts`	The size of the data to be acquired. The limit of the number of points that can be acquired is determined by the amount of data the cache is set to store. The `set_record_length` access function controls this.

Access Functions:

```
void set_record_length(int len);
```

Set the record length of the cache. By default this object only creates 1 record consisting of 8192 segments that fills the entire 2 MWord memory board. Each segment contains 256 words.

`len` Any value between 1 and 8192 that will satisfy the amount of data desired to be acquired.

```
int get_record_length(void);
```

Return the current record length data member value.

5.3.15 Tektronix9504_1 Class

Description: This class is a representation of a specific Tektronix 9504 instrument.

Inheritance:

Instrument p.40, Logic Analyzer p.56, Tektronix9504 p.63

Constructors:

```
Tektronix9504_1();
```

This default constructor queries the database for the last known location of the Tektronix9504_1 function generator. If this address is incorrect searching can occur.

```
Tektronix9504_1(char *name, int dev, int id);
```

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int check(void);
```

Check to see if the current address specified in the Instrument class is for this logic analyzer by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.16 Network Analyzer Class

Description: Defines the interface that all types of network analyzers must satisfy.

Inheritance:

Instrument p.40

Constructors:

```
NetworkAnalyzer() : Instrument();
```

Create an Instrument object.

```
NetworkAnalyzer(char *server, int device_addr, int busid) :
Instrument(server, device_addr, busid);
```

Create an Instrument object with the following parameters.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
virtual int configure(double startfreq, double stopfreq, int
numpts) = 0;
```

This is a pure virtual member function that must be defined by all objects that want to inherit from a Network Analyzer object. Configure sets the start and stop frequencies of the analyzer and the number of data values it stores. The return value is used to flag an error, a return value of 0 means normal operation.

`startfreq`	The starting frequency for the sweep in Hz.
`stopfreq`	The stoping frequency for the sweep in Hz.
`numpts`	The number of data values stored.

```
virtual int calibrate(void) = 0;
```

Run a stored calibration program. The return value is used to flag an error, a return value of 0 means normal operation.

```
virtual int get_sparms(char* sparms, double *realdata, double
*imagdata);
```

Get the s-parameter data of size numpts. The return value is used to flag an error, a return value of 0 means normal operation.

`sparms`	Character string representing the s-parameter.
`realdata`	The real part of the s-parameter. Storage must be provided and maintained by the user.
`imagdata`	The real part of the s-parameter. Storage must be provided and maintained by the user.

Access Functions:

```
virtual double get_startfreq(void);
```

Returns the value of the Startfreq data member in Hz.

```
virtual double get_stopfreq(void);
```

Returns the value of the Stopfreq data member in Hz.

5.3.17 Wiltron 37247A Class

Description: This class implements the Logic Analyzer class interface for the Wiltron 37247A network analyzer.

Inheritance:

Instrument p.40, Network Analyzer p. 67

Constructors:

```
WILTRON37247A() : NetworkAnalyzer();
```

Calls constructor for a NetworkAnalyzer object.

```
WILTRON37247A(char *server, int device_addr, int busid) :
NetworkAnalyzer(server, device_addr, busid);
```

Calls the constructor for a NetworkAnalyzer with a specified Address value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there maybe one or more buses connected to a workstation a bus identification argument is needed.

Member Functions

```
int configure(double startfreq, double stopfreq, int numpts);
```

Configure sets the start and stop frequencies of the analyzer and the number of data values it stores. The return value is used to flag an error, a return value of 0 means normal operation.

`startfreq`	The starting frequency for the sweep in Hz. Frequency range is 40 MHz - 20 GHz.
`stopfreq`	The stopping frequency for the sweep in Hz. Frequency range is 40 MHz - 20 GHz.
`numpts`	The number of data values stored. Valid values are 51, 101, 201, 401, 801, 1601.

```
int calibrate(void);
```

The purpose of this function is to run a stored calibration program. This function is not yet provided. Instead calling this function will prompt the user to perform manual calibration. The return value is used to flag an error, a return value of 0 means normal operation.

```
int get_sparms(char* sparm, double *realdata, double
*imagdata);
```

Get the s-parameter data of size numpts. The return value is used to flag an error, a return value of 0 means normal operation.

`sparms`	Character string representing the s-parameter. Choices are "s11", "s12", "s21", "s22".
`realdata`	The real part of the s-parameter. Storage must be provided and maintained by the user.
`imagdata`	The imag part of the s-parameter. Storage must be provided and maintained by the user.

5.3.18 Wiltron37247A_1 Class

Description: This class is a representation of a specific Wiltron37247a instrument.

Inheritance:

Instrument p.40, Logic Analyzer p.56, Wiltron37247a p.71

Constructors:

`WILTRON37247A_1();`

This default constructor queries the database for the last known location of the Wiltron37247A_1 network analyzer. If this address is incorrect searching can occur.

`WILTRON37247A_1(char *name, int dev, int id);`

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int check(void);
```

Check to see if the current address specified in the Instrument class is for this network analyzer by comparing serial numbers. A positive return value means that the correct instrument has been found.

5.3.19 Power Supply Class

Description: Defines the interface that all types of power supplies must satisfy.

Inheritance:

Instrument p.40

Constructors:

```
Powersupply(void):  active_output(1), Instrument();
```

Creates an Instrument object and sets the active_output data member to the default value.

```
Powersupply(char *server, int device_addr, int busid) :
active_output(1), Instrument(server,device_addr,busid);
```

Calls the constructor for an Instrument object with a specified Address value, and sets the active_output data member to a default value.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

`virtual int select_output(int) = 0;`

This function must be implemented to inherit from this object. Select the active output. The return value is an error value where 0 represents normal operation and 1 represents an error.

`virtual int voltage(double voltage) = 0;`

This function must be implemented to inherit from this object. Set the output voltage for the currently active output. The return value is an error value where 0 represents normal operation and 1 represents an error.

`voltage` Output voltage.

`virtual int enable(void) = 0;`

This function must be implemented to inherit from this object. Enable the currently active output . The return value is an error value where 0 represents normal operation and 1 represents an error.

`virtual int disable(void) = 0;`

This function must be implemented to inherit from this object. Disable the currently active output. The return value is an error value where 0 represents normal operation and 1 represents an error.

`virtual int set_current_limit(double limit);`

Set the output current limit for the currently active output. The return value is an error value where 0 represents normal operation and 1 represents an error.

`limit` Output current.

Access Functions:

`virtual double get_voltage(void);`

Returns the value of the Voltage data member in Volts.

`virtual double get_current_limit(void);`

Returns the value of the Current_limit data member in Amps.

5.3.20 HP6621A Class

Description: This class implements the power supply class interface for the HP6621A power supply.

Inheritance:

Instrument p.40, Powersupply p.72.

Constructors:

```
HP6621A(void): MaxVoltage(20.2), MaxCurrent(10.3) ,
Powersupply();
```

This constructor creates a `Powersupply` object and sets default member function values.

```
HP6621A(char *server, int device_addr, int
busid) :  MaxVoltage(20.2), MaxCurrent(10.3),
Powersupply(server,device_addr,busid);
```

This constructor sets default member function values and creates a Powersupply object with specific Address values.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

```
int select_output(int output);
```

Select the active output channel. The return value is an error value where 0 represents normal operation and 1 represents an error.

`output`	Output channel values are 1 or 2.

```
int voltage(double voltage);
```

Set the output voltage. The return value is an error value where 0 represents normal operation and 1 represents an error.

`voltage`	The voltage can be set from 0 - 20 Volts.

```
int enable(void);
```

Enable the output. The return value is an error value where 0 represents normal operation and 1 represents an error.

```
int disable(void);
```

Disable the output. The return value is an error value where 0 represents normal operation and 1 represents an error.

`int set_current_limit(double limit);`

Set the output current limit. The return value is an error value where 0 represents normal operation and 1 represents an error.

`limit`	The output current limit can be set from 0 - 10 Amps.

`int check(void);`

Check to see if the current address specified in the Instrument class is for this function generator by comparing serial numbers. A positive return value means that the correct instrument has been found.

Access Functions:

`double get_voltage(void);`

Query the instrument for the current output voltage value in Volts.

`double get_current_limit(void);`

Query the instrument for the output current value in Amps.

5.3.21 HP6621A_1 Class

Description: This class is a representation of a specific Hewlett Packard 6621A power supply instrument.

Inheritance:

Instrument p.40, Powersupply p.72, HP6621A p.74

Constructors:

`HP6621A_1();`

This default constructor queries the database for the last known location of the HP6621A_1 power supply. If this address is incorrect searching can occur.

`HP6621A_1(char *name, int dev, int id);`

This constructor will attempt to locate the instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

5.3.22 Cache Class

Description: Defines the interface that all types of caches must satisfy. Cache objects can be used to create specific setup configurations for data acquisition.

Inheritance:

Instrument p.40

Constructors:

`Cache() :  Instrument();`

This constructor creates an `Instrument` object.

```
Cache(char *server, int device_addr, int busid) :
Instrument(server,device_addr,busid);
```

This constructor creates an `Instrument` object with specific `Address` values.

Member Functions:

`virtual int trigger(void) = 0;`

This function must be implemented to inherit from this object. The Cache instrument has a Logic Analyzer data member that provides the trigger member function. The return value is used to represent an error value. A 0 value represents normal operation.

`virtual int getdata(int *data, int npts) = 0;`

This function must be implemented to inherit from this object. The Cache instrument has a Logic Analyzer data member that provides the getdata member function. The return value is used to represent an error value. A 0 value represents normal operation.

5.3.23 AD20_1 Class

Description: This class is a representation of a specific Analog to Digital converter setup.

Inheritance:

Instrument p.40, Cache p.77

Constructors:

`AD20_1();` This default constructor queries the database for the last known location of the TEKTRONIX9504_1 cache. If this address is incorrect searching can occur.

`AD20_1(char *name, int dev, int id);`

This constructor will attempt to locate the TEKTRONIX9504_1 instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions:

`int trigger(void);`

Call the `TEKTRONIX9504` trigger member function p. 65.

`int getdata(int *data, int npts);`

Call the `TEKTRONIX9504` getdata member function p. 65.

`int check(void);`

Call the `TEKTRONIX9504_1` check member function p. 67.

5.3.24 HDCOMP_1 Class

Description: This class is a representation of a specific Analog to Digital convert setup using a hardware compensator.

Inheritance:

Instrument p.40, Cache p.77

Constructors:

`HDCOMP_1();`

This default constructor queries the database for the last known location of the `HP16500B_1` cache. If this address is incorrect searching can occur.

`HDCOMP_1(char *name, int dev, int id);`

This constructor will attempt to locate the `HP16500B_1` instrument at the location specified by the arguments. If this address is incorrect searching can occur.

`server`	The name of the workstation the instrument is connected to.
`device_add`	The bus address number of the instrument.
`busid`	Since there may be one or more buses connected to a workstation, a bus identification argument is needed.

Member Functions: `int trigger(void);`

Call the HP16500B trigger member function p. 61.

```
int getdata(int *data, int npts);
```

Call the HP16500B getdata member function p. 61.

```
int check(void);
```

Call the HP16500B_1 check member function p. 63.

5.4 Communications Lab Program Examples

The following sections are program examples of how to use some of the above interfaces in the communications lab library.

5.4.1 Function Generator Example

The following program declares a Hewlett Packard 83732A function generator and uses its member functions.

This program first declares a function generator object then sets its frequency to 155 MHz. The instrument is then queried for the output frequency value. Next the output amplitude is set to 0.02031 Volts peak and then the instrument is queried for the output amplitude.

5.4.2 Logic Analyzer Example

The following program declares a Hewlett Packard 16500B logic analyzer and uses some of its member functions.

This program first declares a logic analyzer object then selects the pods needed for data acquisition. Next the TTL logic family is selected and the instrument is triggered. Now the program is ready to receive data, and the `getdata` function is called with the storage for the data already created.

5.4.3 Network Analyzer Example

The following program declares a Wiltron 37247A network analyzer and uses some of its member functions.

This program first declares a network analyzer object and then configures the start and stop frequencies and the number of data points to acquire. Storage is then created for the data and the member function `get_sparms` is called to acquire the data.

The next chapter explains the why an Instrument Database is created and what would happen if it was not available to provide service to the programs. The interface and implementation of the database are also explained.

CHAPTER 6

Instrument Database

This chapter explains the purpose, interface, and implementation of the Instrument Database. The purpose of the database is to provide the software with the last known location of an instrument. Using the database, the programs do not have to search for the location of an instrument as often, decreasing execution time over a program that always searches for an instrument's location. For source code listing see the Virtual Instrument Bus Source Code text [3].

6.1 Instrument Database Implementation

The RPC mechanism was used to create the Instrument Database. The RPC mechanism was chosen because it is desired to access this database by any machine on the network. RPC allows development of database routines without worrying about how they will be accessed by the network. Since the location data of the instruments will be shared by multiple computers, there is a possibility that one program could be reading data while another is writing data, or one program may change the data that a different program just stored. The RPC mechanism enforces the condition that, at most, one remote procedure can be invoked at a given time [4]. Thus, RPC provides automatic mutual exclusion among procedures within a given remote program. This type of exclusion is important for remote programs that maintain shared data. The system blocks other calls until the current call finishes, thus preventing interference with the same data.

Compiling the `instrumentdb.x` file using RPCGEN creates source files used to form the database program. Figure 6.1 shows the files used and created by RPCGEN.

The `databproc.cc` file is first compiled as a C++ file and then linked

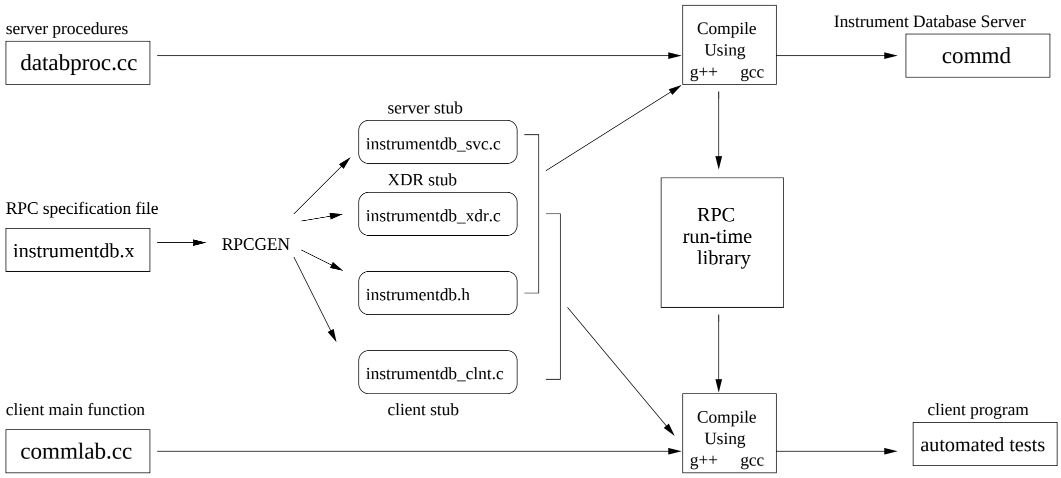

Figure 6.1: RPCGEN generation data base files and how they are used .

with the compiled `instrumentdb_svc.c` and `instruementdb_xdr.c` files to form the Instrument Database server. The `instrumentdb_svc.c` contains the interface routines to the `databproc.cc` procedures and contains the main program of the Instrument Database. The `instruementdb_xdr.c` contains the XDR routines for handling translation between C-structures and XDR data types.

The `instrumentdb_clnt.c` contains the interface routines for the client software that wants to call remote procedures. The Communications Lab Library uses this interface code for calling database remote procedures.

6.2 Database Interface

The database interface provides two basic functions for its operation. These functions are `locate()` and `update()`. `locate()` searches the database for an instrument location and returns its current location if it is in the database. The `update` function changes an instrument's location in the database or adds it to the database after searching has occurred. The Communications Lab Library provides interfaces to these RPC functions to be used in object-oriented programming.

6.2.1 Locate Function

The `locatedb()` function provides an interface to the RPC `locate()` function. First a connection is made to the RPC database server. Once connected, the arguments are translated and the RPC `locate()` function searches the database for the given instrument. If the instrument is new and is not in the database, then it will be added. The return value is a 0 for normal operation and a 1 if an error has occurred. The function prototype is given by

```
int locatedb(Address* a, const char *instrumentid);
```

The input parameters have the following meaning.

`a`	The Address class used to store the returned location of the instrument.
`instrumentid`	Each specific instrument has its own unique identification string for database use.

6.2.2 Update Function

The `updatedb()` function provides an interface to the RPC `update()` function. First a connection is made to the RPC database server. Once connected, the arguments are translated and the RPC `update()` function searches the database for the given instrument and updates its current location. This function is called after searching has occurred because an instrument may have moved or its address may have been changed. The return value is a 0 for normal operation and a 1 if an error has occurred. The function prototype for `updatedb()` is given by the following call statement.

```
int updatedb(Address* a, const char *id)
```

Where input parameters are as follows.

`a`	The Address class used to store the current location of the instrument.
`instrumentid`	Each specific instrument has its own unique identification string for database use. The `instrumentid` in this case represents the instrument that needs to have its location updated.

6.3 Daemon Location and Information

The database runs as a background daemon in the Communication lab. This daemon is located in the `/usr/sbin` directory of one workstation. Both the `startcommd` and `commd` executables should be moved to this location after recompiling. The database commands use an environment variable `COMMDBASE` to located the database daemon. If `COMMDBASE` is not defined the default workstation name is `kojak`.

6.4 Database Examples

The following code fragment is an example of how to use the database functions.

An identification string and `Address` object are created to call the `locatedb` function. The `locatedb` function checks the database for the identification string and if found will return a valid `Address`. If the identification string could not be found, then the program can use other Virtual Bus functions to find a valid address for that instrument. Once a valid address is found the `updatedb` database function can be called to add or update that address for the given identification string.

The next chapter is an overview of the software maintenance process needed to support this software. Source code control and recovering previous versions of the software are also covered.

CHAPTER 7

Software Maintenance

This chapter explains a procedure to maintain source code and libraries , as well as how to upgrade and install all parts of the Virtual Instrument Bus software.

7.1 Source Code Control and Makefile

One of the most important tools for creating large software packages is a source code control mechanism. Source code control is important because it helps the developer(s) maintain different versions of the code. Making changes to working code can be catastrophic if the changes cause the software to fail and the changes cannot be reversed. A source code control system always provides the previous version to enable return to a working version. The source code control system also assigns a specific developer who is able to modify a given source file. Multiple copies can be checked out of the system, but only one developer is able to check a changed version back into the source code control system. The source code control system used by this project is `RCS`.

Compiling source files is accomplished with the `make` utility. It allows one to divide a program's component functions among various files. Whenever file is altered, it notes the fact and recompiles only the altered file. It then links the resulting object module with other required objects to produce an executable program.

The make utility uses a file called `Makefile` to build executables. The `Makefile` for this project builds libraries, bus server daemons, and the database. The source code files for each of these, is listed in the `Makefile`. In order to add files to the libraries, or recompile any changes, the following `make` commands are used.

`make clean`

This command deletes all old executables, object files, and libraries. Doing this ensures that all of these will be recreated and updated. The `clean` function should be executed for all major changes.

`make all`

This command rebuilds all changed source code files and updates executables, object files, and libraries according to dependencies defined in the `Makefile`.

7.2 Creating and Recovering Library Versions

The `make` utility works very well with the RCS tool. It checks out all source code files from a `RCS` directory to compile the necessary program. Program developers should take the following steps to use `RCS` revision names with the `make` utility in order to keep track of major changes to the collection of files.

To set up `Makefile` to use revision names: it is necessary to include the following line in the

`Makefile`: `COFLAGS = $(COFLAGS) $(REVISION?-r ''$(REVISION)":)`.

This allows the specification of REVISION=”name” to create a program using a revision associated with ”name”.

To save a new (named) version of the program/library: First check in all required sources, including the `Makefile`. Next give the working version a symbolic name using `rcs -nname: RCS/*`. Here, `name` can be any string of characters. For example, using “rcs -nrelease_5: RCS/*” associates the name “release_5” with all of the files in the RCS directory. An error occurs if you try to use a revision name which has already been used. To delete a symbolic name from the `RCS` record, omit the “:” using something like “rcs -nbad_name RCS/*”

To make any named version of the software: First determine possible revision names by looking at the log for the `Makefile` using the command `rlog Makefile`. This lists revisions and symbolic names that are associated with each revision. Next check out the appropriate makefile for the desired revision using the correct revision name `co -r"release_1" makefile`. Then make the desired revision e.g., `make all VERSION="release_1"`.

7.3 Creating Programs That Use Virtual Bus Software

Compiling programs that use the Virtual Bus Software must include the Virtual Bus Library and the Communications Lab Library. The following make file called hptest.mk is an example of how to compile a C++ main program called `hptest.cc`.

The first line of this makefile defines a variable called `LIB` that defines the location of the two libraries. (The location will be different for other users). The `INCLUDES` line specifies the location of the include .h files that are needed for compiling. The next line defines the executable name to be `hptest`. `hptest` depends on the `hptest.o` object file and the two library files `libcommlab.a` and `libvbus.a`. If either of those files have been modified since the last time `hptest` was created, the program will be re-linked.

The last line defines how to create the `hptest.o` object file. This file is simply created by compiling the `hptest.cc` source code file. Should the `hptest.cc` source code file have been modified since the last time the `hptest.o` object file was created, it will be re-compiled.

To compile this program use the command `make -f hptest.mk`

7.4 Daemon Location and Information

The Bus Servers that run as background daemons in the Communication lab are located in the `/usr/sbin` directory of all workstations that connect to a bus. Both the `startgpibd` and `gpibd` executables should be moved to this location after recompiling the library. See p. 27 for more information on these executables and how they are started at workstation boot time.

7.5 Virtual Bus Configuration Files

The Virtual Bus software uses a configuration file to know which workstation on the network are host machines for a bus. The `gethosts` function uses the text file `/usr/local/etc/hosts.vbus` to generate a list of possible bus addresses and which host on which these addresses may be valid. (See p 33 for more on `gethosts`). The procedures that use this file first look for an environment variable `VBUSHOSTS` to define the location of `host.vbus`. If this variable is not defined, the default location is `/usr/local/etc`. The following file shows an example configuration file.

The first entry on each line is the workstation name, the second entry is the bus number. In this example there are two buses connected to the kojak workstation. The last two entries are the address range of valid addresses for the bus.

This file must be accessible for every workstation which is accessing the virtual bus. In order to add a new host workstation to the virtual software it must be entered in the `hosts.vbus` file.

CHAPTER 8

Conclusions

The Virtual Instrument Bus software has proven to be an excellent software package for data acquisition across a local network. The convenience of running and creating data acquisition software from any workstation on the network makes development easy for the user. The ease of moving instrument locations and changing instrument addresses for specific test setups without recompiling software allows for easy configuration of automated tests. Once an instrument has had its location or address changed the software will update the database so that no searching will take place the next time the software is run. The Virtual Instrument Bus software is a powerful tool for providing development of complex experiments requiring multiple bus instrument coordination.

8.1 Future Work

The Communications Lab Library could be expanded to include all instruments within the Communications Lab. Any new equipment that will be used for automated tests will have to be included in that library and properly documented in accordance with this thesis.

A graphical user interface could be created to point and click at the instruments included in an automated test.

References

[1] John R. Corbin. *The Art of Distributed Applications.* Springer-Verlag, New York, New York, 1991.

[2] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, Reading, Massachusetts, 1995.

[3] Daryl J. Rawnsley. *Virtual Instrument Bus Source Code.* Communication Lab, University of Maine, Orono, Maine, 1997.

[4] Douglas E. Comer David L. Stevens. *Internetworking with TCP/IP.* Prentice Hall, Englewood Cliff, New Jersey, 1992.

[5] W. Richard Stevens. *UNIX Network Programming.* Pentice Hall, Englewood Cliff, New Jersey, 1990.

Biography of the Author

Daryl Rawnsley was born in Portland, Maine on September 30, 1971. He received his high school education from Lincoln Academy in Newcastle, Maine in 1990.

He entered the University of Maine in 1990 and obtained his Bachelor of Science degree in Computer Engineering in 1995.

In May 1995, he was enrolled for graduate study in Computer Engineering at the University of Maine and served as a Research Assistant . His current research interests include communications and computer networks. He is a member of IEEE, Eta Kappa Nu, Phi Kappa Phi, , and his interests include firefighting and sports. He is the first candidate for the Master of Science degree in Computer Engineering from the University of Maine in May 1997.