A PC-BASED DATA ACQUISITION SYSTEM FOR SUB-ATOMIC PHYSICS MEASUREMENTS

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan

By
Daron Chabot

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5A9

ABSTRACT

Modern particle physics measurements are heavily dependent upon automated data acquistion systems (DAQ) to collect and process experiment-generated information. One research group from the University of Saskatchewan utilizes a DAQ known as the *Lucid* data acquisition and analysis system. This thesis examines the project undertaken to upgrade the hardware and software components of Lucid. To establish the effectiveness of the system upgrades, several performance metrics were obtained including the system's dead time and input/output bandwidth.

Hardware upgrades to Lucid consisted of replacing its aging digitization equipment with modern, faster-converting Versa-Module Eurobus (VME) technology and replacing the instrumentation processing platform with common, PC hardware. The new processor platform is coupled to the instrumentation modules via a fiber-optic bridging-device, the sis1100/3100 from Struck Innovative Systems.

The software systems of Lucid were also modified to follow suit with the new hardware. Originally constructed to utilize a proprietary real-time operating system, the data acquisition application was ported to run under the freely available Real-Time Executive for Multiprocessor Systems (RTEMS). The device driver software provided with sis1100/3100 interface also had to be ported for use under the RTEMS-based system.

Performance measurements of the upgraded DAQ indicate that the dead time has been reduced from being on the order of *milliseconds* to being on the order of several tens of *microseconds*. This increased capability means that Lucid's users may acquire significantly more data in a shorter period of time, thereby decreasing both the statistical uncertainties and data collection duration associated with a given experiment.

ACKNOWLEDGMENTS

Thank you to my thesis supervisors: Dr. W. Eric Norum for helping me get my foot in the door, Professor David Dodds for "going to bat" for an orphaned grad student, and Dr. Ru Igarashi for his tireless efforts. None of this would've happened without you, Ru. Thanks for everything! A special thank you to Dr. Rob Pywell for his guidance and support.

Thank you to the other grad students that I've had the privilege of working with: Tom Regier, Joss Ives, Octavian Mavrichi, Ward Wurtz, and Brian Bewer.

To my parents and family: thank you for your endless encouragement and support.

And last, but not least, thanks to my friends: you've helped more than you'll ever know.

For Jen.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

ADC	Analog-to-Digital Converter
AM	Address Modifier
API	Application Programmer Interface
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input/Output Service
BLT	Block Transfer
BNF	Backus-Naur Form
BSD	Berkeley Software Distribution
BSP	Board Support Package
CAMAC	Computer Automated Measurement and Control
CBLT	Chained Block Transfer
CFD	Constant Fraction Discriminator
DAQ	Data Acquisition System
DFELL	Duke Free Electron Laser Laboratory
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
EOI	Event of Interest
FSM	Finite State Machine
GUI	Graphical User Interface
HIGS	High Intensity Gamma Source
HV	High Voltage
IACK	Interrupt Acknowledge
ICMP	Internet Control Message Protocol
IMFS	In-Memory File System
INH	Inhibit
IOM	Input/Output Manager
IRQ	Interrupt Request
ISR	Interrupt Service Routine
IST	Interrupt Service Thread
LAM	Look-At-Me
MCST	Multi-cast
NIM	Nuclear Instrumentation Module
NTP	Network Time Protocol
PCI	Peripheral Component Interconnect
PIC	Programmable Interrupt Controller
PMT	Photo-Multiplier Tube
POSIX	Portable Operating System Interface
QDC	Charge-to-Digital Converter
QoS	Quality of Service
ROAK	Release on Acknowledgement
RORA	Release on Register Access

RTEID	Real-Time Executive Interface Definition
RTEMS	Real-Time Executive for Multiprocessor System
RTOS	Real-Time Operating System
SAL	Saskatchewan Accelerator Laboratory
SSRL	Stanford Synchrotron Radiation Laboratory
TDC	Time-to-Digital Converter
TFTP	Trivial File Transfer Protocol
TOF	Time of Flight
UML	Unified Modeling Language
VME	Versa Module Eurobus

CHAPTER 1
INTRODUCTION

Photo-nuclear physics experiments utilize high-energy light, γ-rays, to study nature at the subatomic level. Typically, the γ-rays are directed at some target material in order to induce nuclear reactions. Particulate products originating from this beam-target collision may then interact with a physical transducer, or detector, thus producing a measurable electrical signal. This signal is then quantized and analyzed to obtain physical information, such as the identity of the reaction products and their trajectory and energy.

Modern particle physics experiments rely on systems of computer-controlled hardware to collect data from discrete collision events. These data acquisition systems form the interface between experimenters and detectors, and realize vital services for the user, such as analog signal digitization and data transport, processing, and storage. This thesis describes the project undertaken to upgrade several components of a data acquisition system developed and used by researchers from the University of Saskatchewan (U of S). In addition to a detailed design analysis of the system and its modifications, results obtained from the measurement of several key performance metrics are also presented, including dead time, software process latencies, and data bandwidth requirements.

1.1 Data Acquisition Systems

A data acquisition system, or DAQ, may be defined as those components participating in the process of transforming physical phenomena into electrical signals which are then measured and converted into digital format for collection, processing, and storage by a computer [1]. Often, analysis of the collected data is used to influence the data collection policy itself. Therefore, the above definition of a data acquisition system should be appended to include control activities as well.

Data acquisition systems may be as simple as a personal computer (PC) recording audio information from a microphone at a rate of tens of kilobytes per second, or as complex as the computing challenges presented by the ATLAS detector at the Large Hadron Collider facility, where data must be gathered from tens of millions of detector channels

producing terabytes of information per second [2]. The DAQ discussed in this work was designed for small to medium sized data acquisition requirements, typically monitoring several hundred channels of digital information at rates in the tens of kilohertz and volumes on the order of a megabyte per second. This collaboration of hardware devices and the software controlling them, as used by one research group from the U of S, is known as the Lucid data acquisition and analysis system.

While the above definition includes the functionality of a sensor, or detector, which provides the workload to drive the downstream DAQ components, the system detailed in this work is presented from a detector-independent perspective wherever possible, in order to highlight the system's range of utility. While many contemporary experiments rely on standardized systems of electronic hardware to interface with their detectors and other equipment, software interfaces for data acquisition vary from site to site and even from one detector to another. One of Lucid's strengths is its flexibility with respect to the degree of coupling between it and the detector it is interfaced with.

1.2 Background

The topology of the Lucid DAQ and the origin of information flow, from detector to experiment workstation, is illustrated in Figure 1.1. The computational engine of the Lucid data acquisition system consists of several software processes in close cooperation with electronic hardware that is, in turn, interfaced directly with a detector. Particulate products, resulting from physical interactions between incident photons and the target material, fuels the detector and downstream systems with events for measurement and collection.

Prior to outlining the high-level architecture of Lucid and the motivation and scope of the upgrades to it, it will be helpful to provide some background information on those systems external, but essential, to Lucid's operation: the mechanisms by which input events are fed to the DAQ. This entails discussion of the gamma-ray production facility and the sensory device which produces electrical signals characterizing the photo-nuclear reaction products.

1.2.1 HIγS Facility

With the decomissioning of the Saskatchewan Accelerator Laboratory (SAL) in 1999, researchers from the University of Saskatchewan physics department sought new facilities where they may continue to perform nuclear physics experiments. A collaboration with researchers from the University of Virginia and Duke University has met this requirement

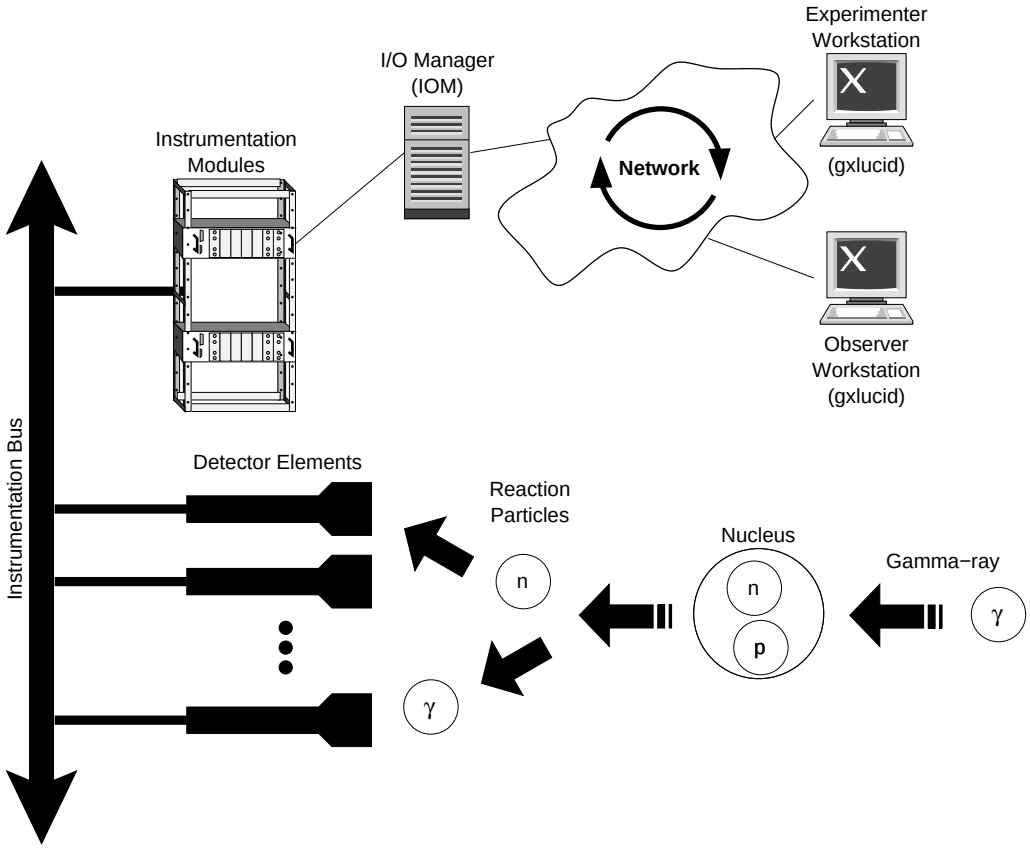

Figure 1.1: Conceptual illustration of equipment and data flow associated with the Lucid DAQ.

in the form of the High Intensity Gamma-Source (HIγS), located at the Duke Free Electron Laser laboratory (DFELL), in Durham, North Carolina. Experimenters utilize its intense γ-ray beams to study nuclear phenomena in a controlled setting. Research areas include studies of the strong force, nucleon polarizabilities and the internal dynamics implied by it, as well as stellar astrophysical reaction channels.

Figure 1.2 depicts the layout of the DFELL storage ring and the generation of γ-rays. As electron bunches circulating in the storage ring pass through the wigglers, an intense beam of photons is emitted in the forward direction along the trajectory of the particle beam. Those photons are reflected from the mirror at the end of the hall, returning to elastically scatter from electron bunches still circulating in the storage ring. It is this interaction that produces the γ-rays required to probe matter at the subatomic level.

In the configuration of Figure 1.2, HIγS produces 100% linearly polarized γ-rays with an average flux of $10^5 - 10^7 \gamma \cdot s^{-1}$, and energies ranging from 2-50 MeV (mega-electron-volts) with an energy resolution ($\frac{\Delta E}{E}$) of better than 1% [3]. For comparison, medical x-rays are in the energy range of 16 - 150 keV (kilo-electon-volts) [4].

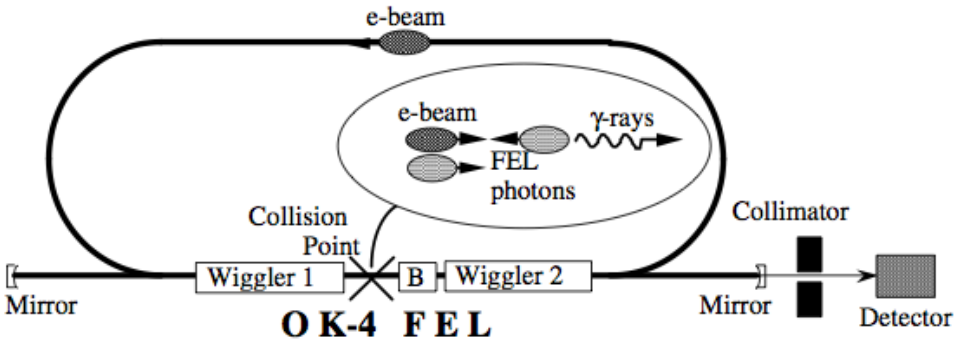

Figure 1.2: Illustration of γ-ray production at Duke University's HIGS facility [3].

Recent upgrades to the HIγS facility include the addition of a 1.2 GeV booster-injector and a helical undulator. These new components will see the laboratory capable of producing completely circularly polarized γ-rays with energies of up to 225 MeV at an average flux of $10^8 - 10^9 \gamma \cdot s^{-1}$.

While the laboratory provides the tools to produce and direct γ-rays at a target, it is the domain of the beam-user to provide the necessary measurement apparatus, or detector. For the neutron-based physics studied by U of S researchers and their collaborators, the detector used is known as the Blowfish array.

1.2.2 The Blowfish Neutron Detector

So named for its resemblance to the fish of the same name, the Blowfish neutron detector was constructed in collaboration with researchers from the University of Virginia (see Figure 1.3). Blowfish is a spherical arrangement of 88 liquid, organic scintillator cells covering a solid angle of approximately π steradians, and is designed to accurately determine the angular distribution of reaction probabilities.

Each scintillator cell is optically coupled to a photomultiplier tube, or PMT. When ionizing radiation interacts with the scintillator, ultra-violet (UV) radiation is emitted. These UV-photons are then directed by an optical waveguide into the photomultiplier tube. In turn, a PMT exploits the photoelectric effect to produce a current pulse. This pulse is on the order of tens of nanoseconds in length and is in direct proportion to the quantity of photons captured from the organic scintillator. Thus, the electronic signal produced is a measure of the energy deposited in the scintillator by an ionizing particle.

The analog signals generated by Blowfish constitute the primary source of input events driving the data acquisition process. Each PMT "hit" is digitized over a period of tens of microseconds along three independent channels: two charge-to-digital (QDC) channels of

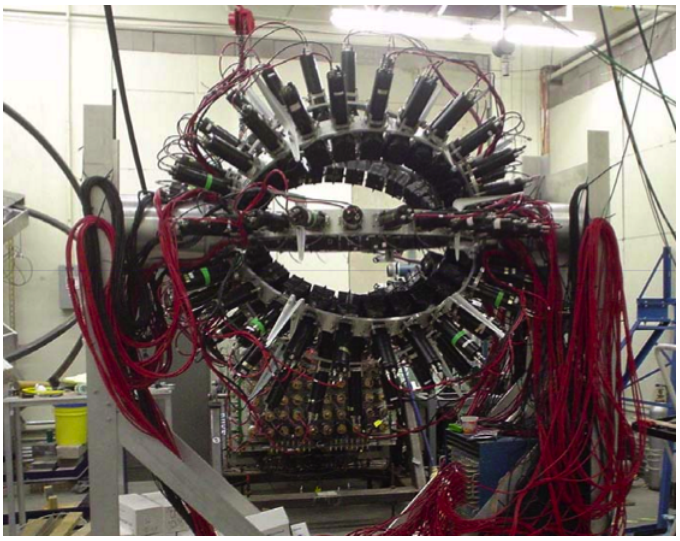

Figure 1.3: The Blowfish neutron detector array.

differing integration times, and one time-to-digital (TDC) conversion channel, which measures the time-of-flight (TOF) of reaction products, from the arrival of γ-rays at the target, to the arrival of reaction-ejecta at the PMT. These three channels of digitized information are then utilized to discern the species and energy of the reaction products. The number of "hits" experienced by a particular PMT of the Blowfish array reveals information regarding a reaction's angular distribution probabilities.

The following section presents a technique to estimate the expected rate at which these events will occur. This information plays an important role in determining schedulability and DAQ requirements for a given experiment.

1.2.3 Input Event Rates

The measurements performed by U of S researchers at the HIγS facility are of a type known collectively as *fixed-target scattering experiments*. In this type of measurement, gamma-rays produced by HIγS are directed at target positioned at the center of a detection apparatus. For the purpose of example, this apparatus may be the Blowfish neutron detector array. The incident gamma-rays interact with the target material, thereby scattering reaction products into the detector, which produces an electrical signal characteristic of the physical parameter(s) under study.

Not all incident gamma-rays are scattered, or react with the target. The number of interactions is proportional to the incident photon flux (in γ/s), the *target constant* (F), and the reaction *cross-section* (σ) [5]. The interaction cross-section, σ , is a parameter derived from quantum mechanical theory, and is expressed in units known as *barns* (1 b

$= 10^{-24} cm^2$). As σ has the dimension of an area, it may be thought of as representing an effective area of interaction between incident and target particles. The target constant, F, is defined as:

$$F = \frac{N_A \rho L}{A} \tag{1.1}$$

where A is the atomic mass in g/mol, N_A is Avagadro's number ($6.02 \times 10^{23}\, mol^{-1}$), ρ is the density in g/cm^3, and L is the target thickness parallel to the γ-ray trajectory. With this information, the rate at which events of interest (EOI) are generated may be estimated.

Given a liquid hydrogen target of length, $L = 10$ cm, and density, $\rho = 0.071\, g \cdot cm^{-3}$, the target constant is:

$$F = \frac{(6.02 \times 10^{23} mol^{-1})(0.071\, g \cdot cm^{-3})(10\, cm)}{1\, g \cdot mol^{-1}} = (2.3\, barn)^{-1} \tag{1.2}$$

Assuming incident gamma-ray flux and reaction cross-section values on the order of $10^6 \gamma \cdot s^{-1}$ and $10^{-3}\, b$, respectively, the rate of occurrence of events of interest (λ_{EOI}) may be now be estimated as:

$$\lambda_{EOI} = \sigma \cdot F \cdot 10^6 \gamma \cdot s^{-1} \tag{1.3}$$

$$\lambda_{EOI} \sim 435\, s^{-1} \tag{1.4}$$

It must be kept in mind that this figure is a "back-of-the-envelope" estimation, and that both the cross-section and the incident flux may vary around the values used here by several orders of magnitude, depending upon experimental conditions. However, the method used to find λ_{EOI} would remain the same.

It must also be noted that detectors themselves are typically incapable of discerning which events are interesting to an experimenter and which events are not. Hence, the total rate of events observed by the detector will inevitably include "uninteresting" events, hereafter referred to as *background* events. Thus, the total event rate, λ_T, seen by the detector will be the sum of the rates of background events and events of interest:

$$\lambda_T = \lambda_{EOI} + \lambda_B \tag{1.5}$$

The HIGS facility in particular has a high rate of background events, on the order of 10 kHz, as experienced by the 88 detector elements of the Blowfish array. Thus, the total event rate seen by the Blowfish detector is often dominated by background events.

1.3 Lucid

Lucid was developed at the SAL in 1988, in an effort to reduce the complexity of developing data acquisition and analysis software for the nuclear physics experiments conducted at the laboratory [6]. The design philosophy of Lucid was that it should, first and foremost, be easy to use, presenting the user with an intuitive interface and a simple framework for application software creation.

Utilizing a top-down design approach, physics experiment software may be decomposed into these basic use cases [7]:

1. *Reading* data, whether the source be detector-generated, or file storage.
2. Data *analysis*, including visual inspection of graphical data representations and computational application.
3. *Storage* of data, to disk, tape drive, or removable optical media.

These ideas are depicted in the *Use Case* diagram of Figure 1.4. This diagram is the first of several in this thesis to utilize the notational features of the Unified Modeling Language (UML). The "stick figures" are known as *actors*: agents that are external, but operationally essential to the system under study. The solid connecting lines are *associations*, symbolizing an information conduit between connected entities, while the dashed lines indicate a *dependency* relationship. The guillemots (<< >>) denote UML *stereotypes*, metadata conveying additional information to diagram elements.

In Lucid's original design, the *Acquire*, *Analyze*, and *Store Data* use cases were reified as three software entities, the *Reader*, *Looker*, and *Writer*, respectively. However, in the present software configuration, the *Writer* process is now defunct, its functionality being subsumed by the *Looker* process and configured via the graphical user interface (GUI), known as *gxlucid*. The *Reader* may obtain data from a variety of sources, including instrumentation modules, disk files, or even simulation program output. The *Looker* process may perform calculations on the data and format it for visual depiction in *gxlucid* in the format of histograms, prior to its being written to permanent storage.

The system's primary modes of operation, *Online* and *Offline*, are delineated primarily by the source of the data stream feeding the system. Online-mode entails reading a data stream provided by an external agent, the I/O manager (IOM), as it collects digitized information from experiment events (see Figure 1.1). Offline-mode is defined as such when the data is sourced from file, either originating from a previously recorded experiment, or supplied by computer-simulation output.

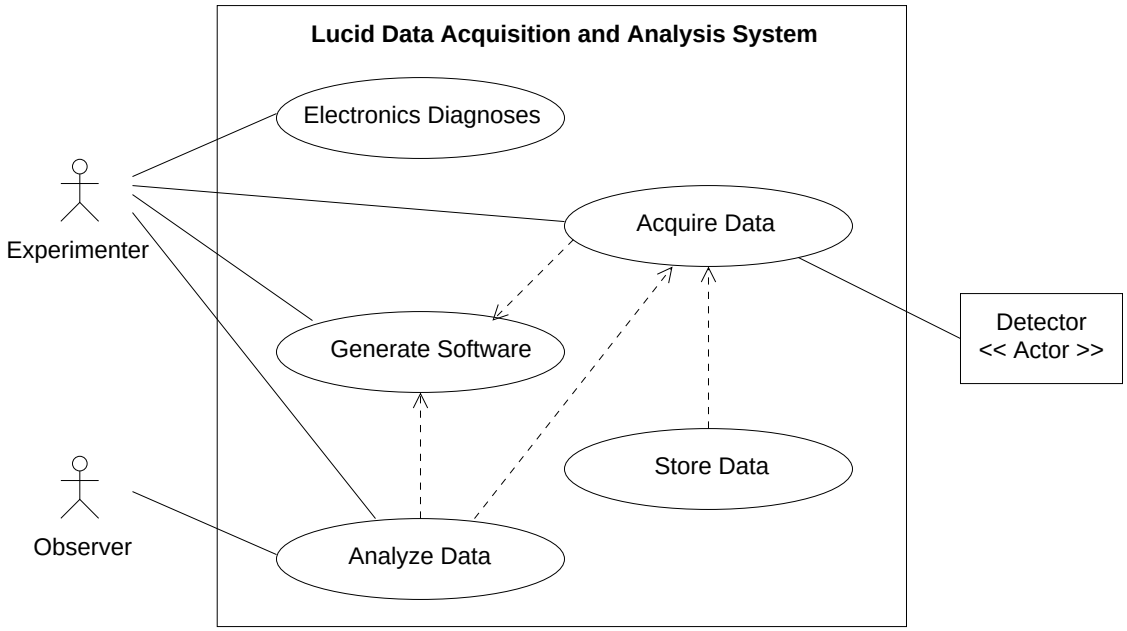

Figure 1.4: Use case diagram of Lucid.

An important element of Lucid's design is network transparency: a user may start the *gxlucid* GUI on one workstation, and connect to an experiment residing on a second machine on another network, which performs the actual data acquisition, and merely forwards the display data to the user's screen. The I/O manager too, forwards the data it collects over a network link. Several observers may monitor an experiment's progress in a network-distributed fashion, but only one user, the "experimenter", may create and modify an experiment.

1.3.1 Physical Subsystems

Structurally, Lucid is composed of three major subsystems, illustrated in Figure 1.5, and ordered here according to their proximity to the detector:

1. *Instrumentation Modules* - these electronic modules may be classified according to standards-family and function:

   (a) *NIM* - Nuclear Instrumentation Modules (DOE/ER-0457T). Provides Boolean operations, signal level-shifting, level-crossing detection, and rate monitoring. Collectively realizes the trigger logic subsystem (discussed in Sections 2.1.1 and 6.1).

   (b) *CAMAC* - Computer Automated Measurement and Control (IEEE-583). Provides analog- and time-to-digital conversion (ADC and TDC), scalers, digital

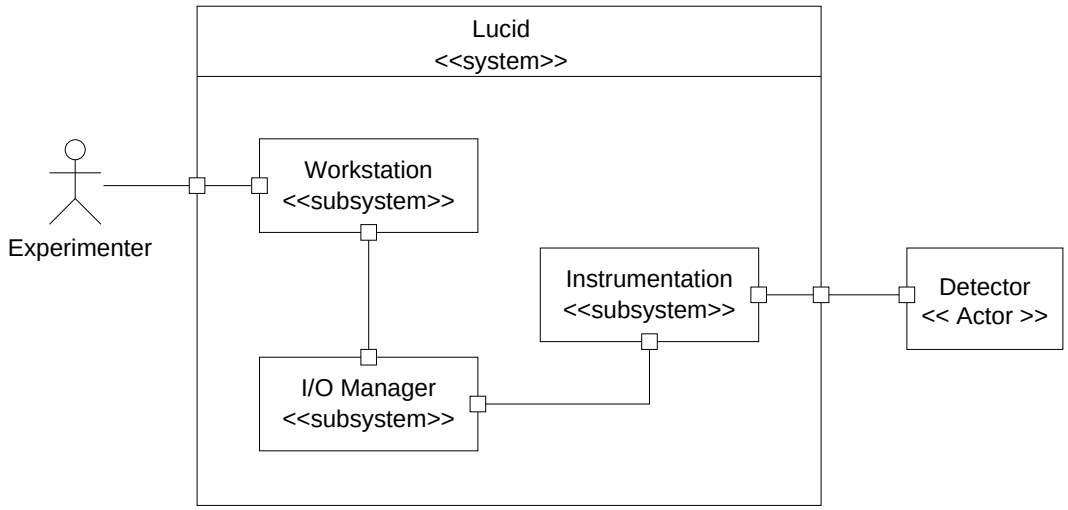

Figure 1.5: Lucid's major subsystems.

I/O, and high-voltage (HV) subsystem control interface.

(c) *VME* - VersaModule Eurobus (IEEE-1014). Provides ADC and TDC services, as well as digital I/O. Serves as a bridge between the CAMAC bus and the I/O manager.

2. *I/O manager (IOM)* - Responsible for direct communication and control of the instrumentation modules. Hosting a real-time operating system (RTOS), the IOM executes software generated by Lucid's code-generation subsystem. The I/O manager's software components are discussed in Chapter 3, while the code generation subsystem is presented in Appendix D.

3. *Workstation* - The software subsystem active on a user's desktop Linux console. Provides the GUI, experiment management services (locating, connecting, user permissions, etc.), software generation infrastructure, as well as communication with the IOM. Also provides essential network services for the IOM, such as DHCP, TFTP, and NTP. These services will be described in Chapter 3.

Furnished with a picture of the DAQ components, their roles, and location in the path of information flow from detector to user, the next section will illustrate the system's employment from a user's point of view.

1.3.2 The User's Perspective: a usage example

Lucid's acquisition and analysis processes, the *Reader* and *Looker*, execute instructions specified by the experimenter in the form of a high-level programming language. This

language shall be denoted as the Experiment Description Language (EDL) for the purposes of this thesis. Typically, the reading and analysis operations required to achieve the experiment's goals are specified in two files:

1. *Reader-description file* - this contains definitions of hardware modules and variables, as well as a list of triggers conditions and the actions that are to be executed upon trigger invocation.

2. *Looker-description file* - this is optional, but may contain definitions of histograms for *gxlucid* to display, computations to perform on the data stream, etc.

These description files are simple ASCII-text files that may be created and modified using almost any text-editing application.

Three programs are used to parse EDL files and generate C language code from their contents: *vmebuildreader, buildreader*, and *buildlooker*. Generation of C source code permits compilation to executable format using the standard, open-source compiler suite *gcc*. While *buildlooker* and *buildreader* produce executable entities that run on a Linux workstation, *vmebuildreader* produces code which is loaded onto and executed by the I/O manager.

An *Online*-session of data acquisition begins by with the creation of an *Experiment* directory, followed by its population with Reader and Looker-description files, expressed in the EDL. In this example, *myExperiment*, Lucid will generate code to read all channels of a VME ADC module every tenth of a second. The Reader-description file, *myExperiment.r*, for this experiment is given below, while the generated "C" code (executed by the IOM) is provided in Appendix B.

```
# Module and variable definitions
define myADC "caen792" S(5)

# Trigger definitions
trigger ReadADC every 0.1 seconds

# Event definitions
event ReadADC:
        blockread and save myADC
```

The Looker-description file, *myExperiment.l*, may be as simple as:

```
# Variable definitions
define myADCHistogram[34] hist from 0 to 4095 (4096 bins)
# Event responses
beginrun:
        myADCHistogram = 0
event ReadADC:
        incr myADCHistogram using myADC
```

While a Reader-description file is comprised of module, variable, trigger, and event definitions, a Looker-description file contains only variable definitions and computations to be executed when events defined in the Reader are encountered in the data stream produced by the I/O manager. For example, *myExperiment.l*, specifies an array of 34 histograms, *myADCHistogram*, each of which corresponding to one channel of the analog-to-digital converter, *myADC*. When the *Looker* encounters a *ReadADC* event in the data stream, it updates the histogram via the *incr*, or increment statement. The *gxlucid* program will provide the graphical depiction of these histograms for the user's inspection.

When *gxlucid* is initially invoked, the user is prompted to create the experiment, *myExperiment*, if it does not already exist. After confirmation, *gxlucid*'s main and message logging windows will appear, as in Figure 1.6. This simple interface was designed to mimic the appearance of a multimedia recording and playback application, in keeping with the theme of a data acquisition device. The *build window*, available via the *Build* drop-down menu of the main window and also shown in Figure 1.6, permits the experimenter to invoke Lucid's code generation mechanism. Upon the successful completion of this stage, the generated executable will be downloaded to and initiated on the IOM. Simply clicking the "Play", or "Record" buttons will initiate the data acquisition code. The data available in *myADCHistogram* will also be available for inspection. This window is also shown in Figure 1.6.

1.4 Lucid Upgrade Project

With the modifications to HIγS, covered in Section 1.2.1, beam-users can expect to see an order of magnitude increase (at least) in generated EOI from a given experiment cross-section. Users might reasonably expect a corresponding decrease in experiment duration given the increased gamma-flux. However, this assumption only holds if their data acquisition system and detectors are able to process data at an equally increased rate.

With the exception of the workstations used during an online data-session, many of Lucid's systems were based on technologies that are now outdated by a decade or more.

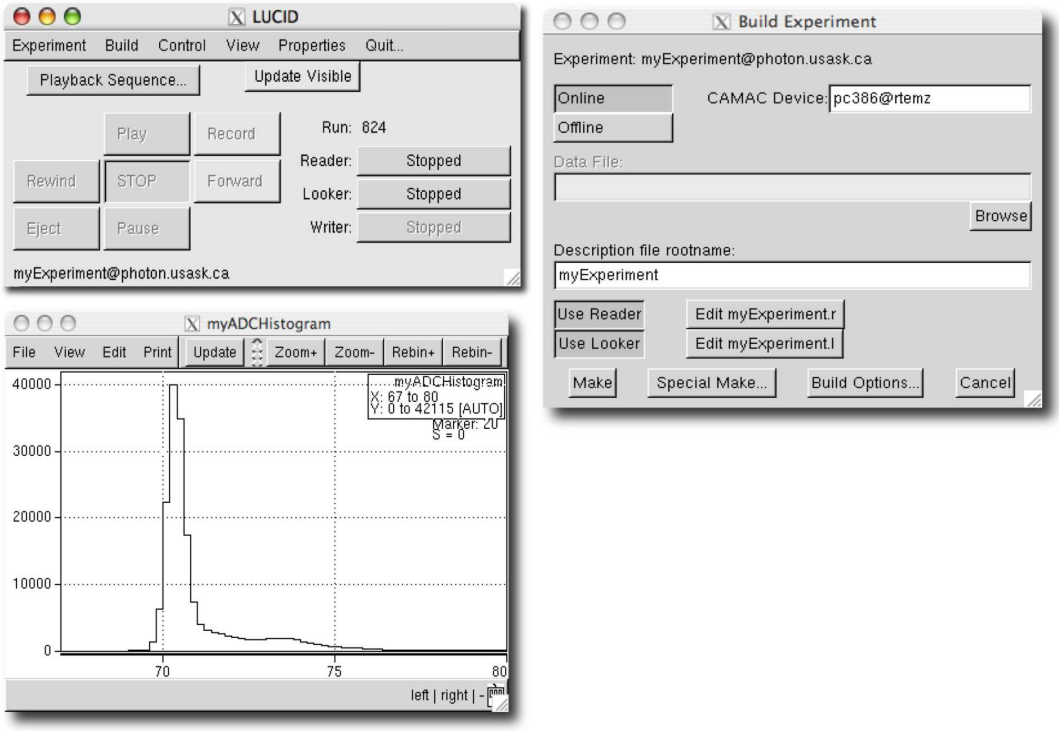

Figure 1.6: Clockwise from top left: the main, build, and histogram windows of *gxlucid*.

This is particularly true of those components in the most time-critical segment of the data processing path: the CAMAC ADC modules and I/O manager, responsible for digitization and extraction of detector information, respectively. Thus, the focus of the upgrades to Lucid must target the IOM and instrumentation (digitization) subsystems.

Given the rapid pace of hardware development, considerations must also be made for the future compatibility of any components that may be candidates for replacement. In addition, it is recommended that hardware/software systems be composed of well-supported, standardized components. These features ease system-integration of advances made in the base technology, and permit access to established technology and documentation.

1.4.1 DAQ Component Upgrades

Figure 1.7 is a "before-and-after" illustration showing the hardware components of the pre- and post-upgrade Lucid data acquisition system. The migration path of each component is detailed in the following:

1. *Digitization Hardware* - migrating the bulk of the digitization workload from CAMAC to VME modules will yield an order of magnitude decrease in the duration

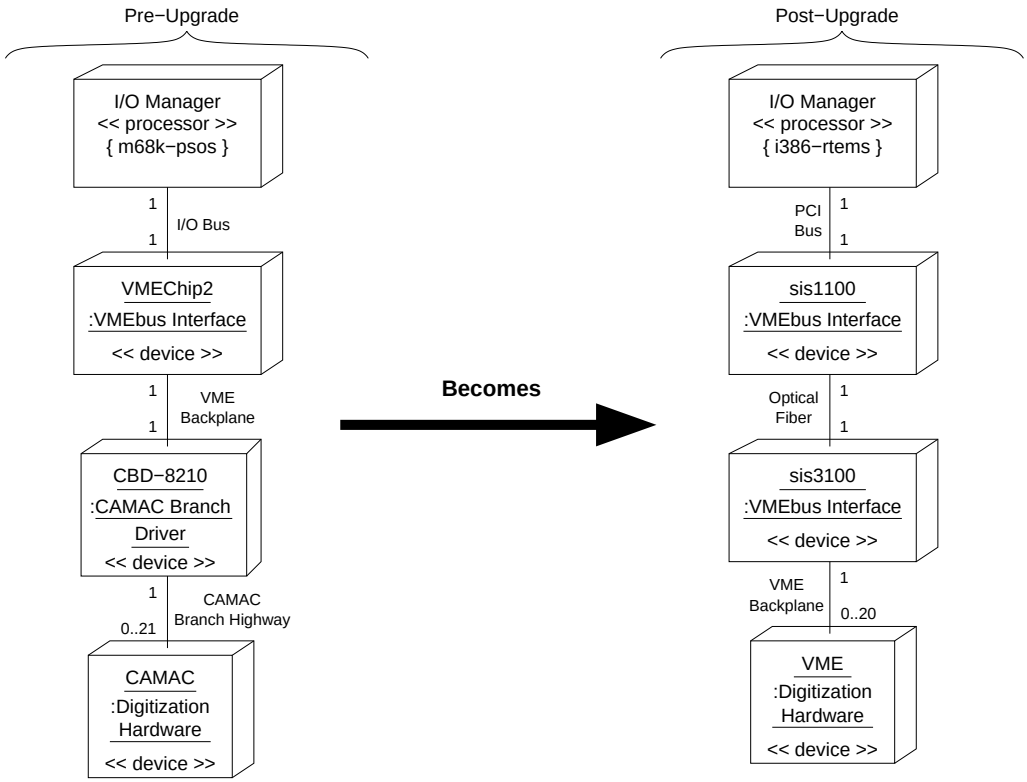

Figure 1.7: Mapping of DAQ component upgrades.

required for that activity (60 μs versus 7 μs). The bandwidth across the VME backplane is some 40 to 80-times greater than that available across the CAMAC dataway. In addition, the particular models of VME modules used possess sophisticated features, like zero and overflow channel suppression, which is advantageous when using detectors such as Blowfish that produce "sparse" data ($\sim 1\%$ channel occupancy). These, and other VME features will be further detailed in Chapter 2.

2. *VMEbus Interface* - the migration of this interface from an application-specific integrated circuit (ASIC), the VMEChip2, to the PCI-based sis1100/3100 device endows the system designer with the freedom to choose any available host platform with PCI support. Also, the VMEChip2 device does not support VME slave-to-master DMA read operations, thus negating one of the advantageous features of the new VME digitization modules.

3. *IOM Host Platform* - The pre-upgrade MVME167 single board computer (SBC) hosts a 40 MHz 68040 processor. This upgrade channel would see the IOM host platform shifted to the Intel i386 architecture; the ubiquitous desktop PC. Not only is this hardware plentiful and inexpensive, but the processing speeds are 10 to 100-

times what the 68040 can achieve. This makes available the possibility of having the IOM perform filtering computations on data in real-time, instead of relegating that task to the domain of software on the Lucid workstation.

4. *IOM Operating System (OS)* - although in this context data acquisition is not a hard real-time application (discussed in Chapter 3), the choice of a real-time operating system for the I/O manager is based on several factors:

   (a) *Quality of Service (QoS)* - given that a general-purpose OS, such as Linux, is designed to schedule processes in a fair manner, while an RTOS employs unfair scheduling by design, a real-time time OS is thus able to guarantee a certain level of performance that may be unachievable using another type of operating system [8]. Plus, a RTOS is typically much less complex than a general-purpose OS, and simplicity is nearly always an advantageous design and development feature.

   (b) *Licensing and Support Issues* - pSOS+ is a proprietary RTOS: fees are due for both usage and support. The Real-time Executive for Multiprocessor Systems, or RTEMS, is freely available under a derivative of the *Gnu Public License* (GPL), and has a responsive support community in the form an online user's mailing list. Additionally, there was some in-house experience with RTEMS prior to the planned upgrades.

   (c) *A Common API* - both pSOS+ and RTEMS feature application programmer interfaces (API) based on the same standard, the Real-time Executive Interface Definition, or RTEID. This greatly eases porting applications between the two operating systems.

   (d) *Increased Feature Set* - obtainable via extensions, RTEMS is configurable with features not found in pSOS+, such as dynamic loading and linking of code libraries into its run-time environment.

Although not indicated in Figure 1.7 above, additional software had to be written in order to take advantage of the increased functionality available with the new VME hardware. This entailed additions and modifications to Lucid's code-generation infrastructure. Also, the sis1100/3100 VME-PCI interface shipped with support for the Linux and NetBSD operating systems. Thus, the device driver software had to be ported for operation under RTEMS.

Based on these details, the I/O Manager is the primary focus of this thesis.

1.4.2 DAQ Performance

With the upgrades of the previous section in place, some questions naturally arise: "How well does the new system perform?" and "How does it compare to the system it replaced?" It is performance-related questions such as these which this thesis answers.

One of the most important performance characteristics of a data acquisition system is its *dead time*. This is the period of time during which the system is busy processing, and cannot respond to new event arrivals. Dead time results in the loss of input events. This prolongs an experiment's duration and therefore plays an important role in determining scheduling requirements.

Another important performance metric for a DAQ is its bandwidth requirements. Considering only the I/O manager, its input bandwidth is determined by the rate of data transfer available from VME to PC, and on the output by the rate of data transfer across the network connecting it and the experimenter's workstation.

The performance of the upgraded I/O manager was measured in several areas, including dead time, software and hardware processing latencies, and the range of possible data transfer rates. While a detailed comparative study between the old and new data acquisition systems was not performed, some performance figures are available for the old system [9].

1.5 Thesis Overview

This thesis focuses on the I/O manager subsystem of Lucid, and is organized around two main topics: the design of and upgrades performed to that subsystem, followed by a description of DAQ performance metrics and the results of their measurement for the IOM.

The next chapter will examine the instrumentation subsystem, including VME, CAMAC, and PCI bus systems and their features. Following the hardware treatment, the I/O manager is hierarchically decomposed, exposing the details of its components, services, and features. This includes details on the porting of the sis1100 driver and RTEMS. An examination of Lucid's code generating component and the modifications performed to it are discussed in Appendix D.

The second half of the thesis begins with a discussion of the analytical framework on which the DAQ performance measurements are based. This includes theoretical discussions on dead time and its relation to the field of queueing theory. Following this are details of the performance measurement techniques. The thesis closes with a presentation of the measurement outcomes, conclusions drawn, and suggestions for future investigation.

CHAPTER 2
DAQ HARDWARE SYSTEMS

Lucid is an event-driven, or *reactive* system: actions are executed in response to events generated by device or software signals. Digital information sourced by the instrumentation modules is the catalyst which drives data acquisition activities. Although the system's software and hardware play equally important roles, the design of the software is influenced to a large extent by characteristics of the physical devices under its control. In light of their systemic importance, this chapter examines the structure and capabilities of Lucid's hardware infrastructure.

The hardware devices of Lucid may be broadly catagorized as belonging to two, major groups:

1. *Instrumentation Devices* - provide detector-centric services, such as trigger and digitization logic, as well as high-voltage distribution and monitoring. These are *slave* devices, performing their tasks only at the direction of a managerial processor.

2. *Personal Computers (PC)* - both the I/O manager (IOM) and Linux workstations are implemented on the PC architecture. These devices perform *user services*, such as instrumentation control, data analysis and storage, and inter-PC communication.

The instrumentation electronics are largely based on the modular, standardized technologies of NIM, CAMAC, and VME specifications. These modules are ultimately connected to the PC architecture of the IOM via a specialized bridge device, the sis1100/3100 VME-PCI interface. In turn, the IOM communicates over standard Ethernet cable networks with software processes on other PC workstations. These network communications are also enabled via PCI hardware.

Lucid's instrumentation devices are discussed here, ordered by their historical appearance: NIM, CAMAC, and VME. This approach reveals the evolution of modular electronic equipment commonly found in particle physics measurements. The influence of earlier systems on newer generations of devices is readily apparent when presented in this manner. Coincidentally, this approach also closely mirrors the flow of data in Lucid, from source to sink.

Following the path of data and control signals, from their origin in the instrumentation subsystem to their destination in PC memory, leads naturally to a discussion of the PC architecture and its PCI bus system. Continuing along a similar vein, the chapter concludes by examining the issues that arise when interfacing the VME and PCI bus systems. This includes an overview of the bridging technology used within Lucid, the sis1100/3100 VME-PCI interface, and its features for coping with the bridging issues.

2.1 Instrumentation Subsystems

Figure 2.1 reveals those subsystems of which the instrumentation subsystem is itself comprised. The high-voltage, trigger, and digitization logic subsystems are directly interfaced with the detector, providing key services for its signal production. An overview of each of those services is provided in the following discussion.

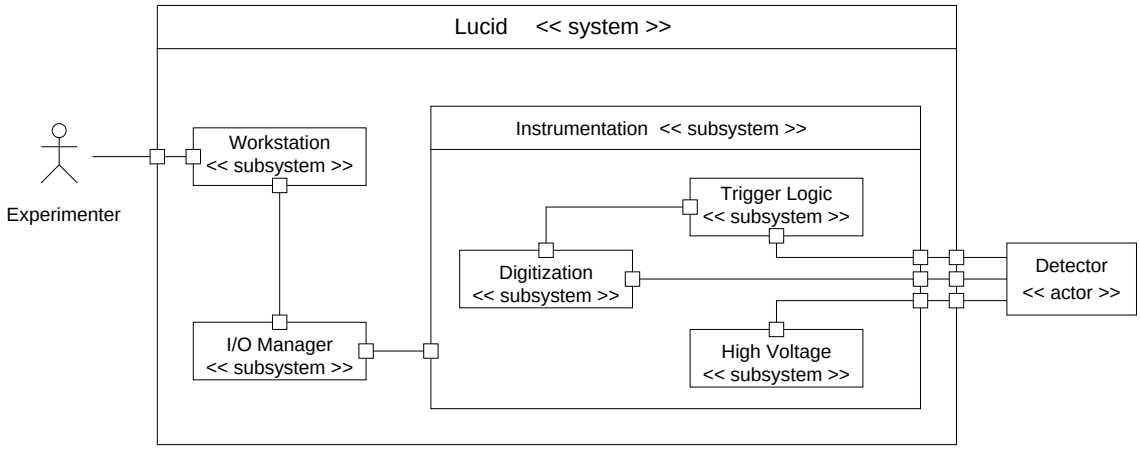

Figure 2.1: System view, exposing the inner-details of the Instrumentation subsystem.

2.1.1 Trigger Logic

In the context of particle physics experiments, a *trigger* is a collection of one or more signals indicating that some event of interest (EOI) has occurred, based on some pre-defined condition for its existence [5]. These existence conditions are known as an *event signature*. For example, in the case of the experiments performed at HIγS with the Blowfish detector, a "neutron event" may be triggered by a coincidence of photo-multiplier tube (PMT) and γ-pulses within a time-window of pre-defined tolerance. Thus, a trigger may be formed by the logical AND of signals from the detector and from the beamline state.

Other trigger schemes may be more complicated and include multiple "levels" of trigger decisions, where event signature discrimination is made increasingly stringent with each subsequent level of event analysis. While the initial stage of a multi-level trigger system is almost certain to be implemented entirely at the hardware level, the latter stages may be realized by software processes using information available from the preceding trigger stages.

2.1.2 Digitization Logic

In order to store and process analog signals, they must first be digitized. The digitization systems employed by Lucid consist primarily of two types of logic: charge-integrating and time-to-digital converters, or QDC's and TDC's, respectively.

Recall from Section 1.2.2, the output signal of a photomultiplier tube (PMT) is an electrical current pulse. QDC modules integrate that input current over the duration of a GATE signal (typically on the order of 100 ns), enabling experimenters to determine the amount of charge, and hence the amount of electrons produced in the particle-scintillator interaction. This value is directly related to the amount of energy deposited in the scintillator by the incident particle. Furthermore, the particle species may also be determined from the QDC values by a technique known as pulse-shape discrimination (PSD). This technique exploits the fact that the time-constant of the scintillator's light-decay characteristics differs between neutrons and gamma-rays. However, a detailed discussion of PSD is beyond the scope of this thesis, and will not be covered further.

TDC devices are functionally equivalent to the familiar "stop-watch": separate *start* and *stop* signals control a counter of sub-nanosecond resolution, allowing precise determination of event durations. For example, time-of-flight (TOF) measurements of sub-relativistic particles are obtained from the interval defined by a start-pulse, triggered on the arrival of γ-rays at the target, and a stop-signal emitted by a detector PMT in response to a scintillator event. Given this duration, the distribution of times-of-flight may be used to discriminate between gamma-rays and neutrons. In the case of the slower-travelling neutrons, the TOF distribution is also representative of their energy distribution.

2.1.3 Detector High Voltage (HV) System

The eighty-eight photomultiplier tubes of Blowfish require high-voltage (HV), DC supplies for operation. PMT's require DC bias-voltages on the order of a kilovolt to generate the electric fields necessary for correct operation. Other detector systems commonly used in particle physics, such a multi-wire proportional chambers (MWPC), drift chambers, and

time-projection chambers (TPC), also have high voltage requirements for similar reasons.

The high-voltage system used with Blowfish is a LeCroy 1440 multi-channel HV system, consisting of a central control-node and multiple 16-channel distribution modules delivering up to 2.5 kV at 2.5 mA per channel. Each PMT requires one delivery channel. Per channel control is achieved through a CAMAC-to-serial interface, a LeCroy 2132 module. This module communicates with the HV-1440 control-node over a 2400-baud, universal asynchronous receiver/transmitter (UART) channel, and with the IOM via its CAMAC interface. Under software control, the experimenter may activate/de-activate the control-node, get and set voltage points and current limits, and also get and set voltage ramp-rates.

2.2 Instrumentation Components

The services described in the previous section are realized by modular electronic devices, or modules for short. These modules may be characterized not only by function, but also by the standards to which they adhere. Standards provide the mechanical, electrical, and communication specifications that define families of devices. The use of standards-based technologies provides a stable, flexible platform, in the sense that compliant devices may be shared, exchanged, and inter-operate within a well-tested and well-documented framework.

Another benefit of modular instrumentation systems is their configuration flexibility. It is simple to restructure a system of modules to meet the requirements of a new experiment, or to add new modules to augment an existing configuration in response to increased experiment demands, or technological advances. Also, modules may be independently tested, prior to their integration with a more complex system.

NIM, CAMAC, and VME are the standardized instrumentation modules used within the Lucid data acquisition system. These modules share the common feature of being housed in a crate, or bin structure. All three module-families provide and distribute power from a crate-integrated supply, but CAMAC and VME crates also feature a digital data communication path, known as a CAMAC *dataway* or VME *backplane*. This communication path, or *bus*, shuttles control and data signals between crate-resident modules and an instrumentation manager, typically a microprocessor. Each family of instrumentation modules is further detailed in the following discussion.

2.2.1 Electrical Bus Systems

Although it is always possible to achieve inter-device communication by wiring them in a point-to-point configuration, this scheme has obvious drawbacks that could be avoided by requiring the devices to share a common communication path. Power distribution requirements may be met in the same fashion.

In the context of computer architecture, a bus may be defined as a subsystem that enables the transport of data and/or power between components of a larger system [10]. These data transfers are initiated upon request by a single bus *master* communicating with one or more *slave* devices; slaves may not publish data autonomously.

A master wishing to initiate communication with a single slave device will broadcast the address of the slave on the bus. All slaves on the bus will compare this address with their own, and connect to the bus if their address corresponds to that of the broadcast address. Note, this implies slave addresses must be unique. Assuming the master-slave connection has been established, only then may data and/or control signals be transferred over the bus. The procedure governing the sequence of information exchange between master and slave is known as a *bus protocol*. The most basic exchange of a single slave address and data piece is known as a *bus cycle*. Typical data transactions may require several bus cycles for completion.

All bus transactions require a synchronization mechanism to maintain timing integrity across bus cycles. To accomplish this, bus cycles may be executed using either *synchronous* or *asynchronous* methods. Asynchronous architectures depend on a *handshake* mechanism between master and slave. This serves as an acknowledgment from the slave that the previous instruction was received and the next stage of the transaction may now proceed. Synchronous designs rely on a clock to orchestrate transactions between master and slave. Acknowledgments need not be sent by the slave, thus eliminating the time consumed by a handshake apparatus.

Byte-ordering, or endian-ness, is an important aspect to consider when interfacing multiple bus systems. The term endian refers to the order in which bytes are stored in memory. A bus is classified as being either *big-endian*, or *little-endian*, based upon the position of the most significant byte (MSB) of a data word relative to the address it occupies in memory. For example, consider the 32-bit number, $1234ABCD_h$: within a little-endian architecture, successive bytes of this number would be stored in memory exactly as it is written, with the MSB occupying the highest address (see Figure 2.2). However, a big-endian system would store the data word as $CDAB3412_h$, with the MSB at the numerically smallest address. Conversion from one representation to another is a task best realized

using hardware resources, if at all possible, thus removing the burden of byte-swapping from a CPU.

Figure 2.2: Illustration of byte-ordering for little-endian (left) and big-endian (right) systems. Memory addresses are indicated above the storage cells.

Bus systems share several features in common, differing only in the details of their implementation. The following general features constitute a framework within which different buses may be compared and contrasted [5]:

1. *Arbitration* - as a bus constitutes a shared resource, some provision must be made to permit the arbitration of resource contention among multiple bus masters.

2. *Addressing* - three techniques are typically used to identify communication endpoints: i) geographical, or positional addressing, ii) logical addressing, and iii) broadcast, or multicast addressing.

3. *Data Transfer* - this includes both the volume per transaction, in terms of bytes, and the delivery method, which may be either serial or parallel.

4. *Control/Status Signals* - control signals permit variation of general read/write transfers into more specialized operations, such read-modify-write, bitwise test-and-set, or block transfer operations. Also, some method must be available for devices to indicate transaction success or the presence of abnormal conditions, perhaps compelling a controlling device to take appropriate action.

5. *Interrupts* - these signals permit a device on the bus to indicate that it requires the attention of another device, typically a central processing unit.

With the exception of interrupts, which are always asynchronous signals, and may therefore occur at any time, the above features are listed roughly in order of their occurrence within a typical bus cycle.

In the following discussions, CAMAC, VME, and PCI buses will all be examined in the context provided by this framework.

2.2.2 NIM

Although devices from the Nuclear Instrumentation Methods (NIM) family do not share a communication pathway, and therefore do not constitute a bus system by the definition given above, they are directly interfaced with both the Blowfish detector and the CAMAC and VME bus systems. Hence, their role in the data acquisition system is critical, and it is natural to discuss the NIM standards prior to the buses with which they are interconnected

Nuclear Instrumentation Methods was established as a standard in 1964, and still enjoy wide-spread use in contemporary particle physics measurements. The standard specifies physical dimensions, power requirements, and pin configuration for modules housed in a NIM bin, or crate, which provides power for the modules at ± 6, ± 12, ± 24 volts.

The NIM standard specifies three sets of logic levels:

1. Fast-negative, or NIM-logic levels provide rise-times on the order of 1 ns at 50 Ω input\output impedance. This logic level is defined by current ranges, corresponding to voltages of 0 V and -0.8 V for logic 0 and 1, respectively.

2. Slow-positive logic is rarely used due to the unsuitability of its very slow rise-times for use in fast-pulse electronics.

3. Emitter-coupled logic (ECL) voltage levels and interconnects have also been added to the NIM standard.

Of these logic types, the DAQ makes extensive use of the fast-negative and ECL versions.

Within the DAQ, NIM modules are used to realize Boolean-logic functions, rate-meters, signal level-shifting, and threshold-crossing detectors, known as *constant fraction discriminators* (CFD). The analog output signal from each of Blowfish's photomultiplier tubes is fed directly into a CFD channel, which produces a fast logic pulse at a constant fraction of the input-pulse height only if the analog pulse exceeds a programmable threshold. Taken in coincidence with a facility-provided signal indicating the passage of an electron bunch in the HIγS storage ring, the CFD output pulses denote the *signature* of an event of interest, and hence set in motion the entire chain of processes to acquire data pertinent to that event.

2.2.3 CAMAC

CAMAC, or IEEE 583, was originally developed in 1969 by the European Standards on Nuclear Electronics (ESONE) committee. The NIM committee in the United States further

refined the standard, where it soon experienced widespread adoption in nuclear, astronomical, and medical research, as well as in industrial control system applications [11].

A CAMAC system is composed of crates, populated with instrumentation modules, like ADC's, TDC's, and scaler devices. Not only does the 25-slot crate house the modules and provide an integrated power supply, but it also contains the module interconnection fabric, or motherboard, also known as the *dataway* in CAMAC jargon. The dataway is the communication medium over which data and control signals flow between modules and the crate controller (CC), occupying the two, rightmost slots of the CAMAC crate.

The role of the crate controller is two-fold: first, it serves as crate master, delivering commands to slave modules, and second, the CC serves to link the instrumentation devices of a crate with managerial logic, typically a microprocessor.

Multi-crate systems may connected in either a serial (loop), or a parallel (branch) arrangement. For the hardware configuration presently used at HIγS with the Blowfish array, the latter, branch-topology is employed.

In the branch configuration, the CC is interfaced with a device known as a *branch driver*. Lucid utilizes the CBD 8210 parallel branch driver, a VME module, to bridge the CAMAC dataway with the VME bus. The branch driver connects to the crate controller by way of a 66-wire cable known as the *branch highway*.

Command and control signals are encoded in a 24-bit word, passed from controlling entity (the IOM in this case), to branch driver, to crate controller, and finally across the dataway to a target module. Data and status signals follow the reverse path, back to the IOM. The code-word is associated with the mnemonic *BCNAF* (**B**ranch, **C**rate, Statio**N**, Sub-**A**ddress, **F**unction), and has the structure illustrated in Figure 2.3.

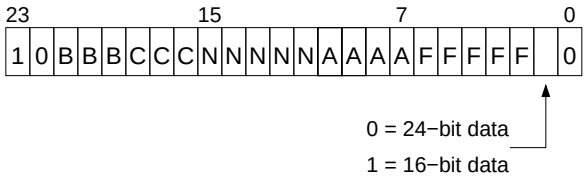

Figure 2.3: The 24 bit CAMAC code-word structure.

2.2.3.1 Arbitration

In the context of Lucid, all CAMAC access is serialized by the I/O manager. All CAMAC modules are slave devices and respond only to the direction of the Crate Controller, the lone bus master of each crate. Therefore, because there is no possibility of competition for the bus, the CAMAC standard does not define an arbiter.

2.2.3.2 Addressing

The “N” line is a point-to-point connection allowing the CC to select one or more modules as transaction targets (see Figure 2.4). This is a form of geographical addressing, as each “N” line corresponds directly to the module’s slot, or position, within a crate.

The sub-address, or “A”, component of the BCNAF code-word permits addressing up to 16 different components internal to a module. These may be registers in a device, such as the per-channel conversion buffers in an ADC module.

Broadcast addressing is used for the delivery of the self-explanatory dataway signals *Busy, Initialize, Inhibit*, and *Clear* signals (B, Z, I, and C).

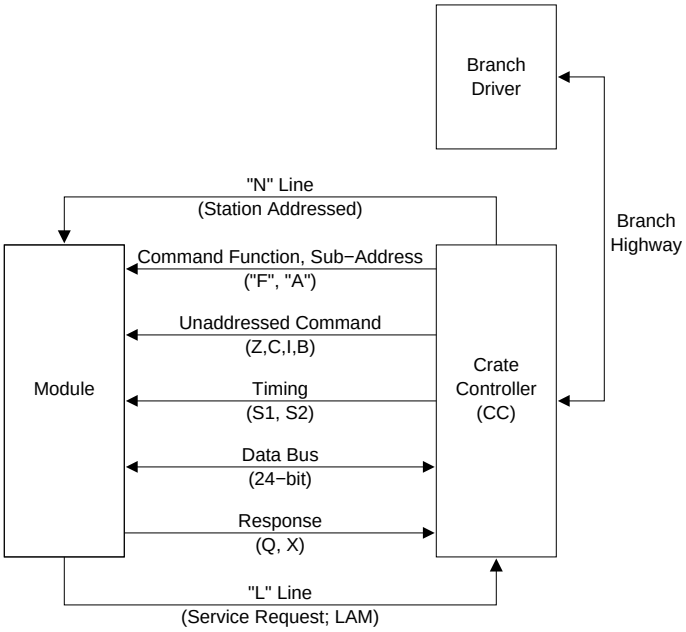

Figure 2.4: Block diagram of CAMAC system.

2.2.3.3 Data Transfer

To initiate a data transfer, the I/O manager sends an approriate BCNAF code to the CC, which in turn drives the “N” line to select a target module, encodes the desired internal register of the target in the “A” component of the code-word, and encodes the transaction type in the “F” component. Module data may be up to 24-bits per word, and is transferred in parallel, across both the dataway and the branch highway.

The CAMAC standard dictates a minimum time of 1.0 μs per control or data transaction. When the data’s path across the branch highway is accounted for, a single 24-bit transaction requires on the order of 1.5 μs. Therefore, the bandwidth available over the

CAMAC branch highway is limited to approximately 2 MB/s.

This relatively small bandwidth is one of the chief motivational factors behind the pursuit to replace CAMAC technology as the primary digitization instrumentation within the Lucid data acquisition system.

2.2.3.4 Control/Status Signals

The "F" component of the CAMAC code-word allows for the definition of 32 function codes, approximately half of which are defined by the CAMAC standard. For example, F(0) is defined as, "read data from group register one", and F(16) is, "write data to group register one". Other defined function codes may instruct the module to clear a register, or silence a module demand signal, the "L" line of Figure 2.4. The remaining undefined "F" codes are left to instrumentation designers to implement in a module-dependent fashion.

The "X" signal is issued by a module when it has successfully recognized and accepted the NAF portion of the command code as being an action it can perform.

The interpretation of the "Q" signal was left to the control of module designers, and therefore its meaning varies from module to module. For example, it may indicate an operation has successfully been completed.

2.2.3.5 Interrupts

Each module has a point-to-point link to the crate controller via its "L" line, also known as the "Look-At-Me" line, or LAM. Modules use this connection to indicate that they require service. This type of asynchronous request notification is more generally known as an *interrupt*. From the I/O manager's perspective, all CAMAC module interrupts are routed through the CBD8210 branch-driver on the module's behalf.

Communication of LAM signals over the branch highway is by way of the BD, or *Branch Demand* wire. The 24-bit bidirectional read/write lines of the branch highway carry a demand-status word known as the GL, or *Graded-LAM* word. This is a logical OR of all interrupting modules within a parallel branch.

The *Branch Graded L-request*, or BG, commands each crate in the parallel branch to place its contribution to the GL-word on the read/write bus. In this fashion, a branch driver is able to determine which crates in the branch contain interrupting modules.

2.2.3.6 Concluding Remarks

As mentioned, the CBD 8210 branch driver is the key piece of technology interfacing CAMAC instrumentation modules with the VME backplane. This permits a single point

of control for both systems, thus simplifying communication and control requirements.

However, CAMAC has at least two features which designate it as a technology whose role within Lucid should be minimized:

1. the relatively slow rate of data transfer (2 MB/s) over the dataway and branch highway, and

2. the lengthy digitization period of CAMAC ADC and TDC modules.

Given that detector event digitization lies on a time-critical path within Lucid's event-driven data collection process, faster data digitization and extraction from instrumentation can only be beneficial to system performance.

The next section will examine the VME bus and those features which denote it as a viable candidate to supplant CAMAC in its role as the primary digitization technology within the Lucid data acquisition system.

2.2.4 The VME Bus

The development efforts of several companies resulted in the 1980 release of a microprocessor-independent instrumentation system known today as the VME bus. The electrical specifications were originally based upon Motorola's VERSAbus standard, which supported their 68000 series of processors, while the mechanical specifications (crates and modules) are rooted in the Eurocard form factor. The form of VME hardware found today is a direct result of the hybridization of these two technologies [10].

Over the years since its inception, the VME bus specifications have experienced several refinements, revisions, and additions. For the purposes of this thesis, the most important additions to the standard are known as VME64, and its super-set with extensions, VME64x. Devices compliant with the VME64x specifications include features such as geographic addressing, configuration ROM and control and status regions (CR/CSR), and bandwidth increases up to 160 MB/s with two-edge, source-synchronous transmission (2eSST). Key VME technologies used within Lucid will be covered in the discussion of this section.

The VME bus is a multi-master, multiprocessor bus design that is not tied to any processor family. The maximum capacity of a single crate is twenty-one slots, with the first slot (leftmost) reserved for the occupation of the System Controller (SC).

A system controller implements several vital functions, such as the bus arbiter: a single VME crate may contain up to twenty-one masters, each with the ability to request bus

ownership and data transmission activities. The SC arbitrates via three policies: priority-based, round-robin sharing, or single-level based. An arbitration mode is selected as system initialization and generally remains static over the life-cycle of a system.

Internally, the VME bus is comprised of four sub-buses, as illustrated in Figure 2.5:

1. *Data Transfer Bus* - composed of address, data, and control signal lines, it is used by system masters to move data to/from slaves. It is also used to transfer interrupt status/ID information (i.e. an interrupt request (IRQ) vector) from interrupting modules during an interrupt acknowledge (IACK) sequence.

2. *Data Transfer Arbitration Bus* - the arbiter, functionally housed in the system controller, determines which master is granted ownership of the Data Transfer Bus.

3. *Priority Interrupt Bus* - up to seven, prioritized levels of interrupt may used. During an interrupt acknowledge sequence the IACK and IACKIN/IACKOUT daisy-chain drivers, initiated by the system controller, deliver information between *interrupters* and *interrupt handlers*.

4. *Utility Bus* - a collection of miscellaneous signals used for reset, timing, diagnostics, power failure, and geographic addressing information transport.

Next, characteristics of the VME bus will be examined in the context of the general bus framework of Section 2.2.1.

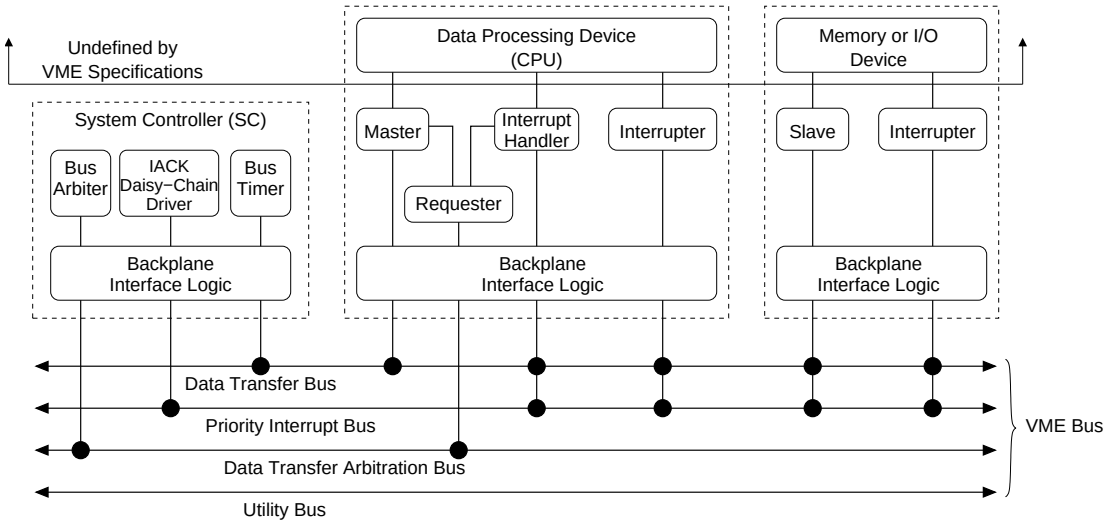

Figure 2.5: Block diagram of a VME system. Figure adapted from [10].

2.2.4.1 Arbitration

All VME bus masters must negotiate for bus ownership with the Arbiter in slot one, prior to any master-slave bus transactions. Typically, an Arbiter chooses a bus owner amongst contending masters based on one of three policies, chosen at system power-up:

1. *Priority Policy* - masters may assert one of four lines *bus-request* lines, to indicate their desire for bus ownership. In the event that multiple masters simultaneously assert the same bus-request line, the master physically closest to the Arbiter (System Controller) will win bus ownership due to the way in which "ownership" is propagated between adjacent slots along the Data Transfer Arbitration Bus.

2. *Round-Robin Policy* - all masters will be sequentially granted bus ownership. When the last master has surrendered the bus, the first master will again be granted the bus, and the cycle will begin anew.

3. *Single-Level Policy* - this is a simple FIFO-based scheduling policy (First-In, First-Out). Multiple, simultaneous requests for bus ownership are handled the in the same fashion as in 1), above.

Caution must be exercised when configuring a multi-master VME system: it is possible for a single master to monopolize the bus, thus starving all other masters of bus access. VME masters will relinquish bus ownership according to their subscription to one of two policies:

1. *Release-When-Done* (RWD) - the master will only yield the bus when it has determined that it no longer needs the bus.

2. *Release-on-Request* (ROR)- the master will surrender the bus to any competitor's request for bus ownership.

The present configuration of Lucid utilizes a single VME bus master subscribing to the *Release-When-Done* policy and the *Single-Level* scheduling policy.

2.2.4.2 Addressing

VME standards permit several address ranges, or *address spaces*, to concurrently exist. These are A16, A24, A32, and A64 in the the VME nomenclature, and define viable address widths of 16, 24, 32, and 64-bits, respectively. A40 is also defined, but is very rarely used.

An extra classifier is used to further characterize a VME address region and data transfer-type. This 6-bit entity is known as an *address-modifier* (AM), and it serves to denote additional attributes of an address space. For example, the 24-bit CR/CSR address space is associated with an address-modifier of 0x2F (hexadecimal notation), while the AM associated with an A24/D32 block transfer is 0x3F.

During a bus cycle, the master tags each address with an AM code. Slave devices monitor these codes, and thus determine which data and address lines to monitor and what type of transaction is expected of them. The AM codes relevant to the Lucid DAQ are listed in Table 2.1.

AM Code	Description
0x3F	A24 BLT
0x3D	A24 Data Access
0x2F	A24 CR/CSR Space
0x2D	A16 Data Access
0x0F	A32 BLT
0x09	A32 Data Access

Table 2.1: Address-modifier codes used within Lucid.

VME address space accesses are uniquely specified by the triplet of data access width, address space width, and address-modifier code, $\{Dxx, Axx, AMxx\}$. It may be helpful to visualize these triplets as parallel planes in a 3-dimensional space, as illustrated in Figure 2.6.

Geographic addressing capability was added to the VME standards with the appearance of the VME64x specifications. This feature allows modules to discover their slot position in the VME crate, and thereby generate a base-address for themselves, determined by that position. This form of auto-configuration is useful as a means to double-check that modules are in the desired position within a crate. The five, most-significant bits of a board's 24-bit base-address are assigned the value of the board's slot position. Thus, in C-language notation, a board's base-address is:

$$baseAddress = (slotNumber << 18) \tag{2.1}$$

2.2.4.3 Data Transfer

Data may be transferred between master and slave in 8, 16, 32, or multiplexed 64-bit volumes. In VME jargon, these data sizes are denoted by the mnemonics D8, D16, D32, and D64, respectively. Two basic types of transaction are possible between masters and

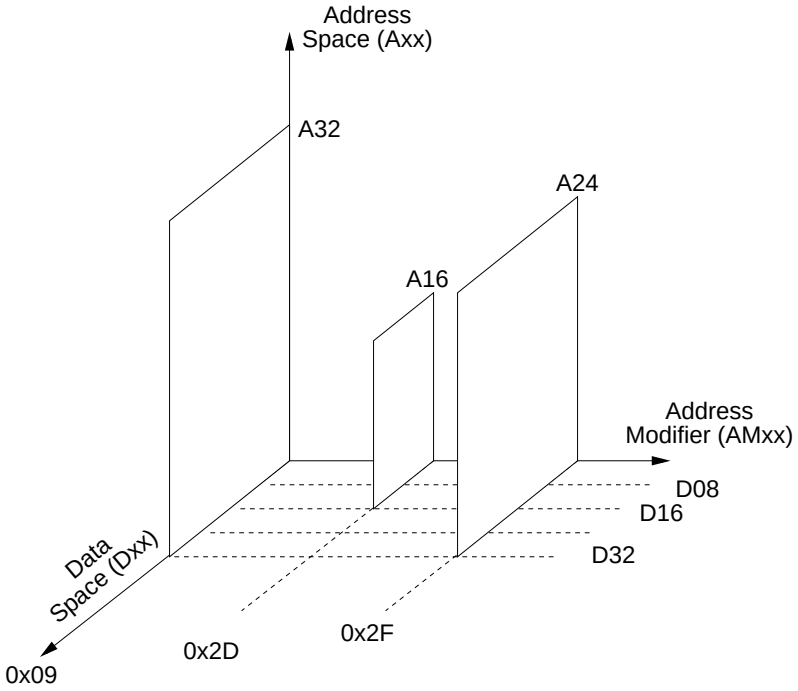

Figure 2.6: Three-dimensional representation of VME address space. Note, the address ranges are not to scale.

slaves: single-cycles and block transfers (BLT). In the remaining discussion, signal lines on a bus are denoted by uppercase lettering, and signals that are active-low have an asterisk (*) appended to their name.

A single-cycle read begins when a master addresses a slave by driving low the address A01-A31, the address-modifier AM0-AM5, IACK*, and LWORD* lines. These are qualified by the falling edge of the address-strobe line (AS*). The master also negates the WRITE* line, and asserts the data strobes DS0* and/or DS1*. The slave then decodes the address, places data onto D0-D31, and asserts the data transfer acknowledge line (DTACK*). After the master has latched the data, it informs the slave of this by negating the data strobe lines. The slave then negates DTACK* to complete the cycle.

Block transfers may be faster than single-cycle accesses because the master presents an address only once, at the onset of the transfer. The slave device will increment the address appropriately as each data word is transmitted, as it knows the data transfer-type is a BLT based on the information it received in the AM of the initial stage of the BLT cycle. In the case of a block transfer to/from a slave with a FIFO memory, such as is present in the CAEN digitization modules used in the Lucid DAQ, the slave need not increment the initial address: all reads from the FIFO start at the first data word contained therein.

Multiplexed block transfers (MBLT) combine the 32-bit address and data lines to form a 64-bit data transfer bus. However, this mode of transfer is not presently used within

Lucid, and will not be discussed further.

2.2.4.4 Control/Status Signals

The VME64x standard specifies two, distinct bus lines for indication of abnormal conditions:

1. BERR*, or *bus error*, is asserted by slaves to indicate the presence of a transaction abnormality, or to signal the termination of a *chained block transfer* (CBLT). Chained block transfers will be discussed in Section 2.2.4.6.

2. BTO*, or *bus time-out*, is issued when the data strobes (DS0-DS1) remain asserted for longer than a predefined period, configured during system initialization. Thus, the BTO* signal realizes the alarm-portion of a "watch-dog" timer service, managed by the VME system controller.

The response of a master to either of these signals is not specified by the VME64x standard, and is therefore implementation-dependent.

2.2.4.5 Interrupts

The VME standards specify a 7-level, prioritized interrupt architecture, driving the bus lines IRQ1*-IRQ7*, with IRQ7* being the highest priority. Those modules generating interrupts on these lines are known as *interrupters*, and those that service their requests are known as *interrupt handlers*. In the case of Lucid, the interrupt handler service is realized by the IOM.

As an example of a typical interrupt sequence, consider an interrupter asserting one of IRQ1*-IRQ7*, indicating a need for service. The interrupt handler monitors one or more of the seven lines, and when it senses one them is active, it requests and acquires the bus. It then places the 3-bit binary number corresponding to the priority of the IRQ level on the address bus, and asserts AS* and IACK*.

The assertion of IACK* serves two purposes: 1) it notifies all modules that an interrupt acknowledgment sequence is in progress, and 2) it initiates the IACK* daisy-chain driver of the system controller, located in slot 01. The daisy-chain signal propagates from IACKIN* to IACKOUT* at each crate slot until the signal reaches the interrupter. At this point, the interrupter places an 8-bit STATUS/ID (also known as an interrupt *vector*) on the data bus, and terminates the cycle with DTACK*.

The next stage of the IACK-cycle depends on which of two types the interrupting module belongs to:

1. *ROAK* - Release-On-AcKnowledgement. This type of interrupter de-asserts its IRQ1*-IRQ7* line when it places its interrupt vector on the bus and drives DTACK*. That is, its interrupt is negated in response to an IACK cycle.

2. *RORA* - Release-On-Register-Access. This type of interrupter only negates its interrupt (i.e. de-asserts) when the handler writes to a certain register on the interrupting module. Thus, the IRQ1*-IRQ7* lines remain asserted even at the conclusion of the IACK cycle.

With the exception of the CBD 8210 CAMAC branch driver, all VME modules used in the Lucid data acquisition system are RORA interrupters.

2.2.4.6 Physics Extensions to the VME64x Standards

Two special protocols were added to the VME64x specifications based on feature requests expressed by the experimental physics community [10]. These are known as the *chained block* transfer (CBLT) and *multicast* transfer (MCST) protocols.

1. *CBLT* - this mechanism allows a master to read a block of data of an indeterminate size from an adjacent set of modules within the same crate, without the need to address each module individually. Thus, one chained block transfer is sufficient to read several modules, each containing data from a single physics event. As the master has no *a priori* knowledge of which modules contain data, the CBLT mechanism is particularly well-suited for scenarios where modules contain little data. That is, when the physics event generates "sparse" data.

2. *MCST* - multicast transactions allow commands (i.e. VME bus *writes*) to be broadcast to a set of slave modules. Thus, a single issuance of a write-cycle reaches all modules of a chosen set.

Both CBLT and MCST transaction take place in the VME A32 address space, with the appropriate D32 address modifier.

Modules supporting these two types of transactions must be configured to respond to a single, unique address in A32 space. This address serves as a *virtual* base-address for the module-members of a VME *Chain-Object*. Additionally, the first and last members of a VME Chain-Object must know their special positions in the chain, in order to be able to initiate and terminate these special transactions.

Both CBLT and MCST transactions utilize the interrupt infrastructure of the VME bus to pass "tokens" between members of a Chain-Object. In a CBLT operation the

IACKIN*/IACKOUT* daisy-chain is used by each Chain-member to indicate to its neighbour that it has finished contributing its block of data, and the neighbour may begin its transference. The last module in the chain asserts BERR* to signal the conclusion of the CBLT. In a similar fashion, the IACK* daisy-chain is used in an MCST transfer to indicate that a module has latched the data successfully, and the neighbouring module may now do the same. The last module in the MCST chain asserts DTACK* to let the master know that the broadcast transaction is complete.

One of the key additions available from the Lucid upgrade project was the CBLT/MCST capabilities of the CAEN QDC and TDC modules. These features greatly aid in reducing both the *number* and the *duration* of accesses to the VME backplane, hence reducing dead time.

2.3 The PC: IOM and Workstation Platform

Figure 2.7 shows the bus topology of a typical PC motherboard. This layout is representative of the systems used within Lucid, as both Linux user workstations and the I/O Manager platform. After the CPU and RAM, arguably the most important components of the PC motherboard are the Northbridge and Southbridge chipsets. Traditionally these form the core logic of PC motherboards.

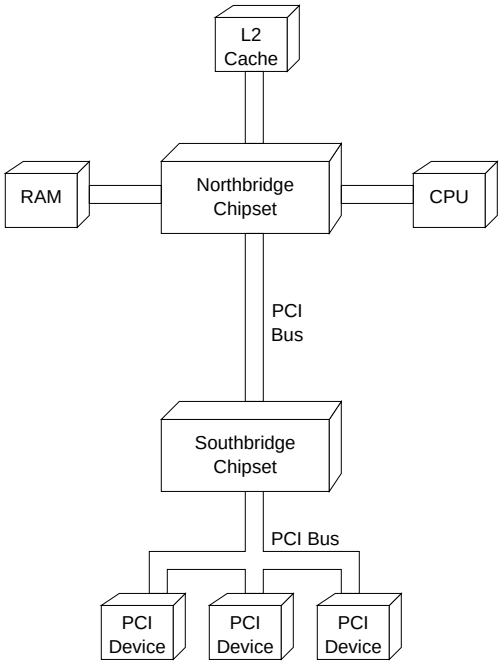

Figure 2.7: PC motherboard layout.

The Northbridge, or memory controller hub (MCH), is responsible for routing communication between the CPU, RAM, L2 cache (if present), and accelerated graphics port (AGP). Additionally, the MCH also houses the host-to-PCI chipset and the PCI arbitration logic.

The Southbridge, or I/O controller hub (IOCH), is itself a PCI device, interfaced with "slow" I/O peripherals such as other PCI devices, the real-time clock (RTC), system firmware (i.e. BIOS on the PC platform), and the hard-drive. Additionally, interrupt and DMA (direct memory access) controllers are also incorporated into the IOCH chipset.

Traditionally, the bus linking the MCH and the IOCH has been the PCI bus, but newer motherboards often use a proprietary solution for this interface. However, this fact is generally transparent to most software. Given the importance of the PCI bus in the context of the Lucid DAQ, this system is explored next.

2.4 The PCI Bus

Originally developed by the Intel corporation in the early 1990's as a means of interfacing external devices with the motherboards of the 8086 processor line, the Peripheral Component Interconnect (PCI) bus was later adopted as a standard administered by the PCI Special Interest Group (PCI SIG). This group later extended the bus' definition to include a standard expansion-connector for add-on boards. These expansion connectors are the familiar PCI slots found most on modern personal computers today. While designed to be processor-independent, the PCI bus is most prevalent on the PC architecture.

The PCI bus is a synchronous, little-endian design. Thus, data and control signals are transferred between a master and slave, or an *initiator* and *target* in PCI jargon, on the edges of an orchestrating clock signal. While the original specification stipulated a 33 MHz clock, 66 MHz systems are occasionally found. But, the 33 MHz implementation still remains the most commonly found, and it is the version found in the PC hardware used within Lucid.

2.4.1 Arbitration

Unique to each PCI slot are a set of bus request and grant signal lines, REQ* and GNT*. An initiator wishing to use the bus asserts its REQ* line, which is received by a central arbiter located in the Northbridge chipset. If conditions are favorable, the arbiter drives the initiator's GNT* line, indicating that it now owns the bus. This arbiter is required to implement a "fair" algorithm, such as round-robin scheduling, in order to avoid bottleneck

or deadlock conditions.

While the arbitration process is asynchronous, and therefore consumes no bus cycles, the arbiter's policy as to which initiator is granted the bus is influenced by the duration that an initiator has held the bus. Each PCI device has an associated maximum time-limit that the device may own the bus. This limit is set by BIOS firmware and written to the device's configuration space registers at system boot-up.

2.4.2 Addressing

The PCI bus supports three address regions: 1) configuration space, 2) memory space, and 3) I/O space. Memory and I/O space transactions are distinguishable as such by the type of CPU instructions used to access regions in each. That is, instructions used to access I/O address regions differ from those used to access memory regions. Some processor families do not implement separate instructions for access to memory or I/O space, but the PC family does. However, devices' use of range-limited, I/O-instruction addressable memory is a discouraged practice.

All PCI devices are required to implement a 256-byte memory region known as the device's *Configuration Space*. PCI configuration space supports device *auto-configuration*, where each device may learn the location of its own resources, independent of software intervention. The layout of this region is shown in Figure 2.8.

The Vendor and Device identification regions permit software to identify specific devices and multiple instances of those devices. Of the fields set by system firmware, the most important are the *interrupt line*, and the *base address registers*, or BAR. The interrupt line value communicates interrupt routing information to device driver software. The value in this field is the number of the interrupt pin on the system's interrupt controller (implemented in the IOCH chipset) to which the PCI device's interrupt line is routed.

When a PC boots up, system firmware (BIOS) builds an address map of the system by determining the amount of physical memory (RAM) present, and the address ranges required for each I/O device, including those on the PCI bus. That is, a PCI device indicates, by values in its BAR fields, the size and type (memory or I/O space) of address space it requires. System firmware assigns a region with the required characteristics to the device, reporting the assigned address of the region(s) in the BAR. In this fashion, a PCI device is able to map its various registers into the address space of the host PC. With the assigned region's base-address indicated by a device's PCI BAR, driver software may access a device's registers with basic *load* and *store* primitives.

PCI BIOS specifications define a set of software functions that must be implemented

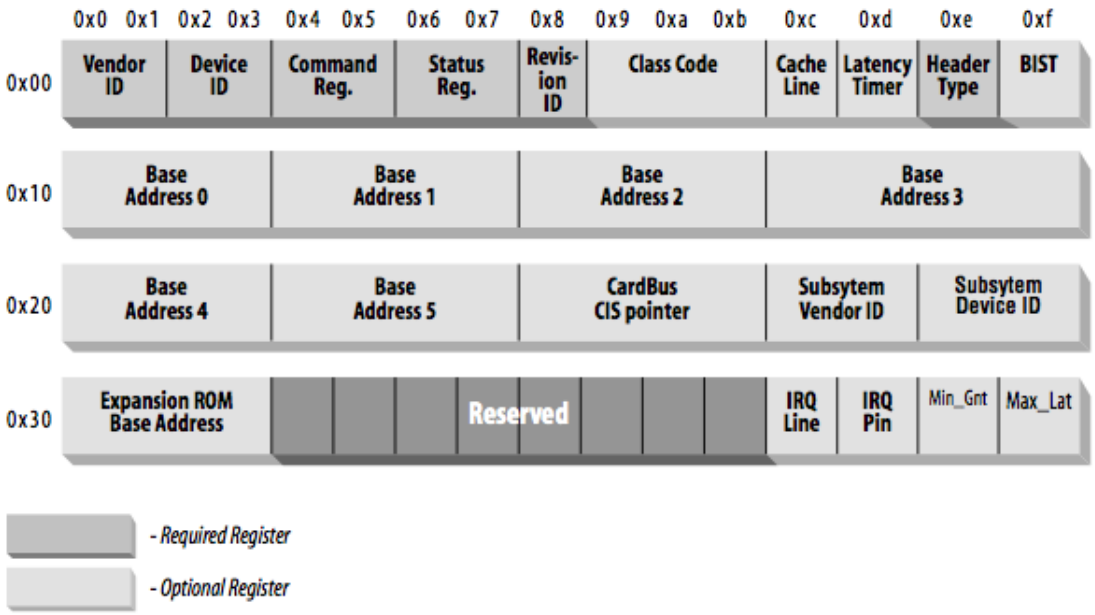

Figure 2.8: PCI configuration space memory region. The required regions must be provided by the PCI device, whereas the optional regions may assigned by the host system's firmware. Figure from [12].

by system firmware, and made available to operating system software. These functions provide OS drivers with access to PCI configuration space, including the ability to read/write configuration space registers, locate devices by Vendor and Device ID, and broadcast commands to all devices on a PCI bus via a "special-cycle" transaction.

2.4.3 Data Transfer

PCI utilizes a multiplexed address and data bus, AD[31:0]. That is, both data and target addresses are sent over the same set of physical lines. Bus cycles are initiated by simultaneously driving FRAME* and an address onto the AD[31:0] lines at one clock edge. This is known as the *address phase* of an initiator-target transaction. The next clock edge initiates one or more *data phases*, where data words are transferred over the AD[31:0] lines. Taken together, the address and data phases constitute the basic PCI bus cycle.

An initiator indicates the type of data transfer it wishes to execute by driving the Command/Byte Enable lines, C/BE[3:0], on the same clock edge as the FRAME* signal: that is, during the address phase. The 4-bit, C/BE[3:0] lines encode the transaction *type*, such as memory read/write, I/O read/write, or interrupt acknowledge. During the data phase, the C/BE[3:0] lines indicate which byte-lanes of [3:0] are valid, thus indicating whether the data payload is 1, 2, or 4 bytes wide.

2.4.4 Control/Status Signals

The nature of a PCI transaction is dictated by the 4-bit, C/BE[31:0] lines asserted by the initiator during the address phase of the transaction. In this way, memory, I/O, and configuration space accesses are distinguishable from one another.

The PCI specifications implement notification of transaction errors via two bus lines, PAR* and SERR*. The PAR* line signals a data phase parity error, and is therefore repairable by error-correcting codes or data re-transmission, while SERR* signals an address phase, configuration space access, or special-cycle error condition. An SERR* condition may be correctable at the device or device driver level, or may be passed on to the operating system for further handling. Hence, this type of error is generally not transparent to software operation.

2.4.5 Interrupts

Four bus lines, INTA*-INTD*, are routed from the PCI bus to the motherboard's external interrupt controller in the IOCH chipset. According to PCI specifications, single-function devices may use only INTA* to request processor servicing [13]. The classification of a device as being "single-function" encompasses most common PCI devices, including the sis1100 and Ethernet controller devices used within Lucid.

Thus, single-function device interrupt requests are multiplexed over the INTA* line, and therefore must be *level-triggered* to avoid interrupt losses. The multiplexing requirement also implies that PCI interrupts must be *sharable*. This imposes conditions on the design of driver software for these devices in order to facilitate interrupt sharing.

2.5 VME-PCI Bridging Issues

VME and PCI buses share the common attribute of being processor-independent. That is, neither bus is obligated to use a particular type of processor. However, to accomplish any useful work, clearly some form of processor must be used. One popular solution providing processor-based management of a VME system, is to link it and the processor via the far more common PCI bus, which is local to the processor. The technology used to accomplish this inter-bus coupling is known as a VME-PCI *bridge* device.

Any device interfacing the asynchronous VME bus with the synchronous PCI bus must contend with several difficulties. These issues are born of fundamental differences between the two systems, including data transfer methods, byte ordering, address mapping, and interrupt protocols.

The general design philosophy that should be employed when bridging the two technologies is that of *decoupling* via FIFO memory regions [14]. For example, using this technique, data may be written into a FIFO from the VME-side at that bus' maximum rate, and read from the PCI-side of the FIFO at *that* bus' maximum rate. Thus, transactions on each bus are segregated: each may proceed independently in the most efficient manner available for each bus.

Each of the four key differences outlined above will now be examined in detail.

2.5.1 Address Mapping

The PCI bus defines three memory address spaces, I/O space, memory space, and configuration space, while the VME bus defines A16, A24, A32, A40, A64, and CR/CSR address spaces. Therefore, a bridging technology must provide mechanisms for mapping VME-accessible addresses and address-modifiers to PCI-accessible addresses, and vice versa.

2.5.2 Byte-Ordering

Due to differences in development platform histories, PCI and VME buses differ in their byte-ordering aspect: the PCI bus is little-endian, reflecting its historical ties to the Intel 8086 architecture, while the VME bus is big-endian, reflecting its historical support for the Motorola 68000 processor family. Thus, a bridging technology must provide a byte-lane swapping mechanism.

In practice, swapping could be achieved in software, but this imposes an unacceptable overhead for all transactions that require byte-swapping. Instead of a software solution, VME-PCI bridging hardware may provide this service using one of two techniques:

1. *Data-Invariant Lane Swapping* - In this method, the *value* of the data is preserved by the byte-lane swapping hardware. This technique requires the bridge to know the number of bytes involved in the transfer. However, since only software may truly know the size of a data transfer, it must assume responsibility for properly configuring a bridge on *each transaction*, thus adding a possibly unacceptable overhead penalty to each transfer.

2. *Address-Invariant Lane Swapping* - In this method, the *address* of each byte is preserved by the lane-swapping hardware of a VME-PCI bridge. Thus, each byte-lane is "cross-routed" to the other bus architecture, such that each byte is stored at the appropriate address on the other bus.

Clearly, the latter of these mechanisms is the better alternative, as the required behavior may be implemented entirely in hardware.

2.5.3 Data Transfer

Both the VME and PCI buses permit the transfer of blocks of data in a single transaction. However, PCI devices are only guaranteed to hold bus mastership for a pre-determined amount of time. If a block transfer cannot complete within this time window, the PCI bus may be allocated to another arbitrating master, causing the block transfer to abort any data remaining to be transmitted.

The use of FIFO memories can aid in decoupling the synchronous nature of the PCI bus from the asynchronous timing of the VME platform. Thus, it is recommended that any bridging technology implement bi-directional FIFO buffers. That is, both read and write buffers should be present.

2.5.4 Interrupt Protocols

VME specifications provide for seven, prioritized interrupt lines that may be simultaneously active, while the PCI bus routes four interrupt lines to a PC motherboard's external interrupt controller. As already mentioned, single-function PCI devices may only use one of the four interrupt lines (INTA*), thus compelling PCI interrupts to be multiplexed. Additionally, PCI specifications dictate that interrupts must be acknowledged within a maximum number of clock cycles [13].

One solution to the problems imposed by these conditions is to again decouple the VME interrupter from the PCI interrupt handler, and place the majority of interrupt handling responsibility squarely on the bridge hardware itself [14]. Therefore, when a VME-PCI bridge receives a VME interrupt, it should:

1. Immediately acknowledge the VME interrupter and obtain its STATUS/ID (interrupt vector). If the interrupter is of the ROAK-type, this will silence its interrupt on the VME bus.

2. Store the IRQ vector in a PCI-side register, and assert a PCI interrupt signal (i.e. INTA*) to gain the attention of the CPU.

3. Respond to the PCI interrupt handler, and provide it with the vector of the interrupting module.

Using this algorithm, the VME interrupt acknowledgement cycle could be cleanly decoupled from the PCI interrupt acknowledgement cycle.

The next section will discuss the VME-PCI bridge technology used within Lucid, and examine its measures for coping with the concerns outlined here.

2.6 The sis1100/3100 VME-PCI Interface

The bridging technology used within Lucid is a device manufactured by Struck Innovative Systems, the sis1100/3100 VME-PCI interface. This device is composed of two modules, connected via a Gigabit fiber channel:

1. The sis3100 is a VME module residing in the first slot of a VME crate and serves as both the system controller (SC), and the VME-side of the lone bus master in the DAQ.

2. The sis1100 module occupies one PCI slot in the IOM motherboard. It is, however, composed of two cards connected by the Common Mezzanine Card (CMC) form factor:

   (a) The sis1100-CMC, which implements PCI and PC-side VME logic via a Field-Programmable Gate Array (FPGA).

   (b) The sis1100-OPT, which contains the optical communication hardware supporting the PC-side of the fiber channel link.

Figure 2.9 is a block-diagram illustrating the major components and their inter-connectivity within the sis1100/3100 device.

The interface between the custom-logic FPGA and the PCI bus is a device known as the PLX-9054 I/O Accelerator. This device is commonly used to integrate intelligent I/O machinery, such as network routers and switches, with a host system via PCI bus fabric. In the case of the sis1100 application, the PLX-9054 serves to "glue" the FPGA housing the VME and gigabit link logic with the host computer's PCI bus, thus enabling VME bus control through the host's CPU.

The PLX-9054 implements several important features to address the concerns of Section 2.5:

1. *Address-invariant byte-lane swapping* - thus, relieving the host processor from having to convert VME data from big-endian format to its native little-endian format, and vice versa.

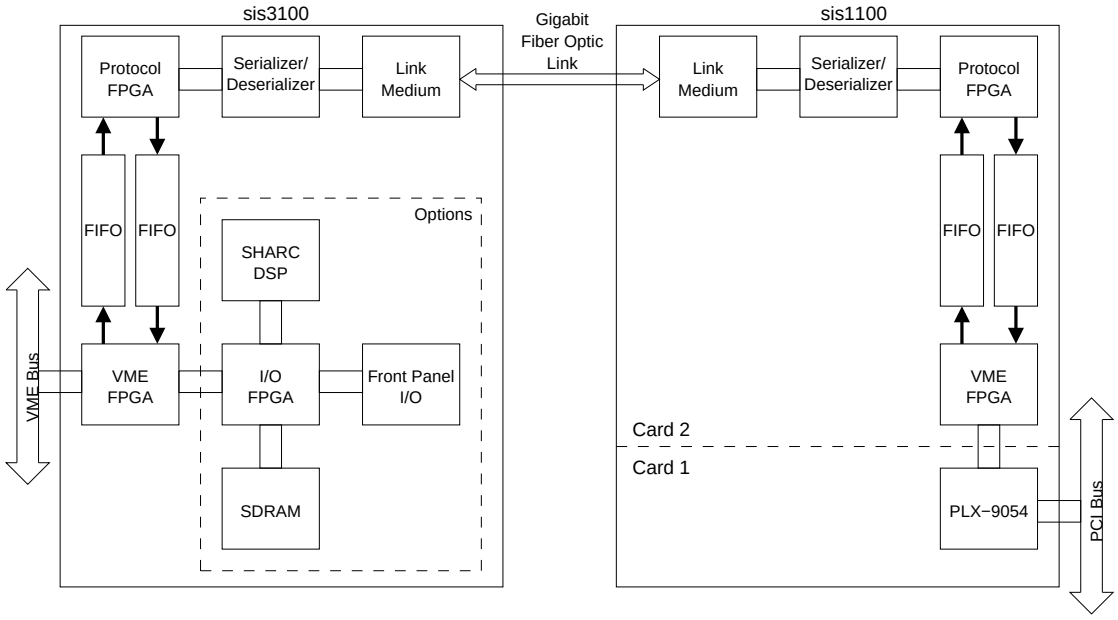

Figure 2.9: Block-diagram of the sis1100/3100 VME-PCI bridge.

2. *Address translation* - all VME addresses are accessible from the PCI-side of the link. Also available are 256 MB of VME address regions, or *windows*. These are user-configurable in 4 MB denominations. This permits the host processor to execute load and store operations to these configured regions as though the addresses were physical memory locations, local to the processor. The concept of accessing device registers through read/write operations to memory addresses is known as *memory mapping*, and it is one of the primary means by which software may interact with hardware.

3. *Read/Write FIFO's* - separate FIFO buffers for each direction of data movement enables writes to and from the PCI and VME buses to proceed at the maximum possible speed on each bus, independent of conditions on the other bus.

Additionally, the I/O Accelerator also contains two, independent DMA engines to support bi-directional movement of block data with a minimum of processor intervention. Once a DMA cycle has been initialized, no processor cycles are consumed by loading or storing data to physical memory, hence the processor is free to attend to other tasks. This capability is especially important in a single-processor, I/O-intensive application, as are the data acquisition requirements of Lucid.

In terms of the way it handles VME interrupts, the sis1100/3100 interface does *not* implement the recommended decoupling scheme of Section 2.5.4. Instead of relying on hardware mechanisms to decouple the VME interrupter from the PCI-side interrupt han-

dler, the solution adopted by the designers of the sis1100 was to relegate interrupt vector acquisition to the software domain.

Although there is no physically obvious reason why the suggested interrupt handling method of Section 2.5.4 could not be implemented on this VME-PCI interface, the physical design of the device does not easily lend itself to such a solution**.** In particular, it is the *forced* serialization/de-serialization of data across the Gigabit fiber-optic channel that suggests a software solution to the interrupt handling problem. The benefits afforded by the parallel architecture of the VME and PCI buses are seriously mitigated by the inclusion of a serial medium in the data path.

The consequences of this interrupt handling scheme are felt most acutely in the software domain and this will be examined in detail in the upcoming chapter on Lucid's software subsystems.

2.7 Summary

This chapter examined the physical devices used within Lucid, beginning with an overview of the instrumentation subsystem. This entailed a description of the available services, the components rendering those services, and their inter-relationships. The physical architecture of each of the system's major communication buses was then described using a framework of characteristics common to all buses. Examining each bus in the context of this framework permits easy comparisons.

NIM, CAMAC, and VME modules form the basis of Lucid's instrumentation subsystem. Communicating with the detector and collaborating over their respective buses, these devices provide the driving stimulus and fuel the consumption of digital information by DAQ software.

Control and data extraction from the instrumentation subsystem is provided by a common, desktop PC interfaced to the modules by a single point of control. Linking the managerial PC with the autonomous instrumentation modules is the specialized sis1100/3100 VME-PCI bridge.

The next chapter will present the design of Lucid's software systems, including its components, deployment topology, structure, and behavioral aspects.

CHAPTER 3
SOFTWARE SYSTEMS

The introduction of new hardware to upgrade the Lucid data acquisition system required extensive software support be created to take advantage of new capabilities. Thus, for each hardware change (see Figure 1.7), corresponding alterations or additions had to be made to Lucid's software infrastructure:

1. *IOM architecture shift* - The platform shift from MVME167 to IA-32 introduced an opportunity to migrate the I/O Manager's real-time operating system from pSOS+ to RTEMS. The process of adapting a piece of software to operate on a platform other than that for which it was originally designed for is known as *porting*. This activity is discussed in Section 3.4.

2. *IOM instrumentation interface* - Software supplied with the sis1100/3100 VME-PCI interface was designed to be executed on either a Linux or NetBSD-based operating system. Therefore, this piece of software had to be ported to function with RTEMS. The relevant details are discussed in Section 3.6.

3. *Support for VME64x features* - In addition to providing access to general read/write operations to VME A24, A32, and CSR/ROM address spaces, routines were added to Lucid to support the VME interrupt infrastructure, as well as BLT, CBLT, and MCST access to modules providing those features. In order to provide user access to the increased function-set available from the new VME hardware, changes were also required to Lucid's code-generation subsystem. However, because the code-generation components are not central to the topic of data acquisition, they are discussed in Appendix D.5.

The description of Lucid's data acquisition software given here is expressed in part using the visual tools provided by the Unified Modeling Language (UML). This tool graphically conveys the structural and behavioral aspects of system components. Together, structure and behavior constitute a *model* of the system under study.

Typically, UML is used to capture the design requirements of a project. The graphical model thus produced may be used to generate the corresponding source code, which in turn

instantiates the product of the design. But this process may also proceed in the opposite sense: given an existing set of source code, a graphical model may be extracted using the notation of the UML. It is in the latter sense that the UML is used throughout this thesis: to graphically document the as-built architecture of the Lucid data acquisition system.

The system model may be viewed through several levels of granularity: from a high, *architectural design* level, where the details are coarsely defined and model components and subsystems are described, through the mid-level of detail, known as *mechanistic design*, where collaborations of objects and their roles are presented, to the finest level, the *detailed-design* level, where the attributes and operations of individual classes are specified [15]. The description of Lucid's software is organized here according to these increasing levels of detail.

Following a feature review of general-purpose and real time operating systems, a component-deployment view of Lucid's software systems is given. This architectural perspective illustrates the major software components, their location, and function within the system. The discussion then turns to a detailed description of each major component deployed to the I/O manager. This includes analyses of the addition of run-time loading/linking capabilities, porting the system to RTEMS, and porting the sis1100/3100 device driver to support operation under RTEMS. Finally, mechanistic- and detailed-design descriptions of the IOM's data acquisition component, the *IOMReader*, are given.

3.1 Operating System Overview

An operating system (OS) consists of several layers of software which provide a managerial role and interface between hardware resources and the applications that require them. Programs access the services of an OS through a set of software routines available as an application program interface (API). In turn, the OS provides an environment within which programs may execute. The relationship between an application and its host operating system is one of mutual dependence: in isolation, each entity is useless.

At the core of an operating system is a program known as a *kernel*. It is the kernel that is ultimately responsible for the management of computer resources. The kernel is alternatively known as an *executive*, although the latter term is typically used in reference to small, modular systems as would be found in a real-time operating system. In this work, the two terms will be considered synonymous.

Figure 3.1 depicts the various software layers existing between application code and physical devices. In this depiction, it is implicit that each layer may only directly communicate with the layer immediately above it. Thus, the kernel communicates with peripheral

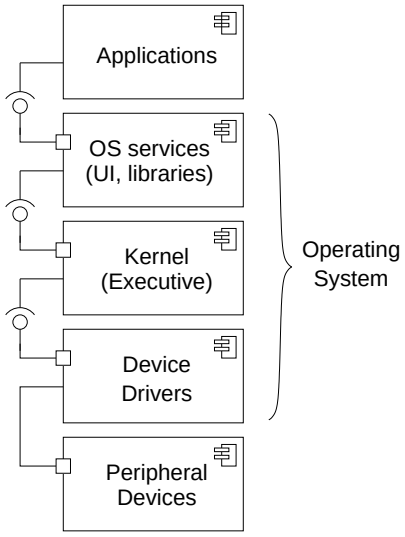

Figure 3.1: Components and interfaces of a typical operating system.

hardware via the device driver layer, application processes make requests of the kernel via a user interface (UI) or library, and so on.

3.1.1 General-Purpose Operating Systems

A general-purpose operating system, such as Linux, runs applications in the context of a *process*. Processes are synonymous with executing programs or applications: each executing program consists of at least one process. A process consists of an address-space allocated by the kernel, the program's executable code and private data, plus other system resources and machine state required to support the program's execution. Address-spaces are unique to each process: unless explicitly permitted to do so, it is an error for one process to attempt access to the address space of another.

Processes are schedulable entities: they are able to compete for, and be granted time on a central processing unit (CPU). Typically, general-purpose OS schedulers are designed to allocate CPU access according to a policy of *fairness*; no single process should be permitted to monopolize the resources afforded by a processing unit.

However, a process is not the only type of schedulable software entity. Processes may be composed of one or more *lightweight* (in terms of resource demands) executable entities, known as *threads* or *tasks*. In this work, threads and tasks shall be considered synonyms. Threads share the address-space of their parent-process and may communicate with each other using less resource-intensive means than must be used for inter-process communication (IPC). Confusingly, inter-thread and inter-process communication methods are often aggregated under the shared label of IPC. Methods available for inter-process

communication include asynchronous signals, message queues, and semaphores. These techniques will be detailed in Section 3.4.

Device drivers are software modules whose purpose is to abstract away the implementation details of device control and present a well-defined, consistent interface to applications [12]. In other words, a driver maps device-specific capabilities onto an interface common to all devices. On Unix-like systems, such as Linux or RTEMS, the device driver API closely follows that available for file access and manipulation.

Often considered part of the OS kernel proper, a driver is OS-dependent in the sense that application requests to access devices must transit the kernel. Hence, the driver-kernel interface consists of those procedures exported by the kernel to provide scheduling, file system, inter-process communication (IPC), and memory management support for hardware devices. Further details of the kernel-driver relationship will be discussed in Section 3.4.

3.1.2 Real-Time Operating Systems

Real-time operating systems (RTOS) are distinguishable as such because of the timing requirements, or *deadlines*, they must satisfy on behalf of applications: real-time systems must react to stimuli within rigid time constraints [16]. These types of quality of service (QoS) requirements define the characteristics of an RTOS.

Deadlines may be categorized as being either soft or hard. A hard deadline is one that simply cannot be missed; a late result is a system failure and may lead to catastrophic consequences. Examples of hard real-time applications include avionic controls in aircraft or rockets, and airbag deployment or anti-lock brake systems (ABS) in terrestrial vehicles.

At the opposite extreme, soft real-time requirements allow for some tolerance and flexibility with a missed deadline: a late result may still be useful and will probably not cause catastrophic failure. Applications with soft deadlines are characterized in a stochastic sense; i.e. in terms of the average quality of service they provide. Examples include cable television signal tuners and the application topic of this thesis, data acquisition systems.

In addition to characterizing the Lucid DAQ as being a soft, real-time application, portions of that system may be further categorized as being an *embedded system*. Embedded systems contain a CPU as part of a larger computing system and provide specialized services not normally found on a desktop system [15]. Real-time embedded systems interact almost entirely with other devices, and not with a human user. The I/O Manager subsystem of Lucid is such an embedded device, despite its deployment on the common desktop PC platform.

3.2 Component Overview

The execution environment of Lucid's software components is shown in Figure 3.2. This type of diagram is known in UML nomenclature as a *deployment view*, and is intended to illustrate the affiliations of software *components*, their *artifacts*, the hardware *nodes* upon which they reside, and the interconnection fabric between nodes.

In Figure 3.2, the I/O Manager and Workstation nodes are further annotated using *GNU-style* notation (*processor family-operating system*): the I/O Manager is of type *i386-rtems*, while the Workstation is of type *i386-linux*. These nodes communicate with each other via their network interface cards (NIC) over 100 Mbps Ethernet fabric, while the sis1100 <<device>> interfaces with the instrumentation subsystem over a fiber-optic link. All <<device>> nodes shown here are controlled by their host processor over a PCI bus.

Note that each component has an <<artifact>> section. Artifacts are those pieces of software physically realizing that component with which they are associated: artifacts are typically executable objects. However, they may also denote other file-types, such as documentation, or software libraries. Also, note that the dependency between the *Client-Side Daemons*, *IOMReader*, and *Code Generation* components has been stereotyped as <<creates>>. This stereotype conveys the idea that the Code Generation component is responsible for the construction of its dependents.

Observe that both the *Instrumentation Daemons* and the *IOMReader* require an interface provided by the *Instrumentation Control* component of the I/O Manager. A similar arrangement may be found between components of the Workstation node.

In the following sections a brief description will be provided for each of the components of Figure 3.2, organized according to the host hardware node.

3.2.1 Workstation (Linux) Components

Although it is not the focal point of this thesis, the role of the Workstation node is no less important in the context of the Lucid data acquisition system than that fulfilled by the I/O Manager. The full-featured Linux distribution hosted on the Workstation provides critical services that the IOM simply cannot: permanent data storage facilities, a graphical user environment, network infrastructure support, DAQ experiment administration, and software development tools.

Of the items in the following list, only the Code Generation component is covered elsewhere in this thesis in more detail than that provided here:

1. *Experiment Management* - This component includes the graphical user interface

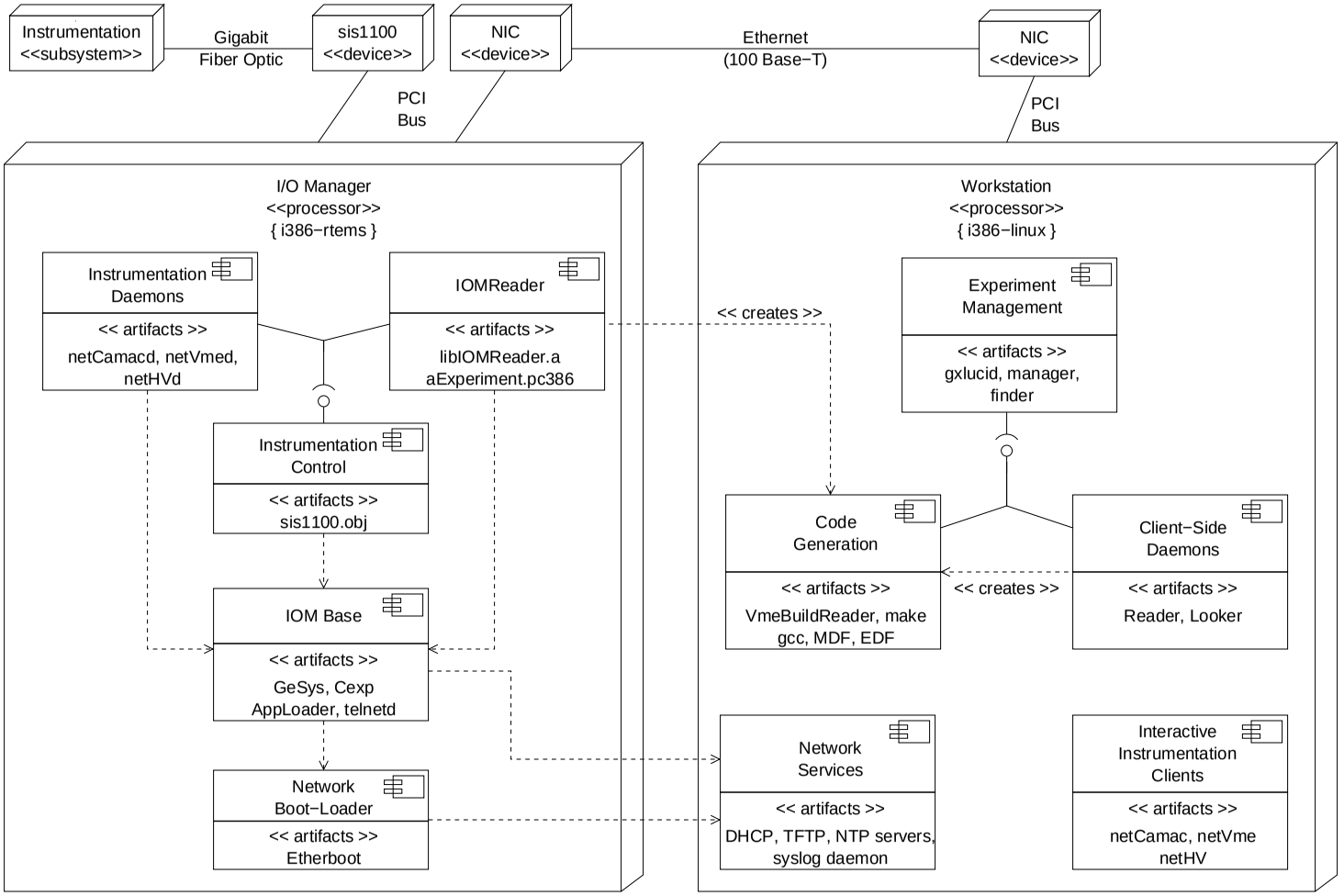

Figure 3.2: Deployment view of the Lucid data acquisition system, illustrating its hardware nodes, bus connectivity, major software components, and artifacts.

(GUI), *gxlucid*, the *finder* daemon for locating (possibly remote) experiments, and the *manager* daemon responsible for distributing and coordinating communication between the user interface and various client programs, such as the acquisition daemons and the code-generation subsystem.

2. *Code Generation* - Provided with input in the form of Experiment and Module Description files, and upon invocation by the *manager*, the *VmeBuildReader* process will produce C-language source code. The source code thus produced is transformed into executable format by the *manager's* invocation of the *make* and *gcc* compiler tools. This software will execute on the experimenter's workstation as well as on the I/O Manager (see Figure 3.2), thus instantiating data acquisition activities. A similar process is also capable of producing a *Looker* program for data analysis. However, that topic is beyond the scope of this thesis. Lucid's code generation component is treated in depth in Section D.5.

3. *Instrumentation Daemons* - The products of the Code Generation component are the processes responsible for data acquisition and analysis, the *Reader* and *Looker*, respectively. The `IOMReader` application is also produced for execution on the I/O Manager, and will be examined separately in Section 3.7. The *Reader* is a client-side proxy application, obtaining data produced by the `IOMReader`, and depositing it in a shared-memory buffer. Data consumers, such as the *Looker* or GUI-clients, pull data from the shared-memory buffer, where it may then be used in data analyses, written to permanent storage, or further distributed to additional client software.

4. *Network Services* - The distributed design of the Lucid system requires several network services be provided to support the operation of the I/O manager. A DHCP server provides a network identity (IP address) to the IOM, the location and name of the IOM's boot-file, the TFTP server where the boot-file may be obtained, as well as the identity of an network time protocol (NTP) server, from whom the IOM may obtain information to synchronize its clock. Details of DHCP and TFTP are provided in Section 3.3. The Linux system-logging facility, or *syslog*, is also used by the IOM to automate the storage and centralization of diagnostic messages produced by it.

5. *Interactive Instrumentation Applications* - These are command-line utilities, providing an interface to instrumentation modules for the purpose of interactively examining and modifying hardware module settings. Although *netcamac, netvme,* and *nethv* rely on access to routines found in Lucid's software libraries, they are intended to be used independent of Lucid. Although not shown in Figure 3.2, these command-

line clients are obviously dependent upon their complementary server applications residing on the I/O Manager (i.e. the Instrumentation Daemons component).

3.2.2 I/O Manager Components

The I/O Manager, or IOM, lies at the heart of Lucid's data acquisition capabilities. It is responsible for direct communication with the hardware systems of the previous chapter, on behalf of a remotely situated human user. The IOM serves as a proxy in this respect, brokering transactions between the user and the DAQ hardware.

The primary role of the IOM is to execute the instructions of the experimenter, responding to periodic, asynchronous, or scaler events, as specified by an Experiment Description file (EDF). The I/O Manager serves to buffer data collected from the VME and CAMAC hardware, "smoothing" the asynchronous arrival of events and providing a structured data stream to the experimenter's *gxlucid* interface.

In order to provide these services, the I/O Manager itself is comprised of several components, each of which contributes one or more vital services (see Figure 3.2):

1. *Network Boot Loader* - permits a remotely located IOM to find and download its operating environment. This component is responsible for loading the `IOMBase`: the *core* application containing the RTEMS operating system and services essential to all other IOM components. More details are found in Section 3.3.

2. *Real-time Operating System* - indicated by the *i386-rtems*, metadata tag attached to the <<processor>> stereotype in Figure 3.2, this component is realized by the RTEMS operating system. RTEMS provides priority-based, preemptive scheduling to multi-threaded applications, networking, and file system services. RTEMS is discussed in depth in Section 3.4.

3. *IOMBase Application* - serves as a foundation upon which all other RTEMS applications depend. `IOMBase` contains the RTEMS OS, its libraries, utilities, and managers. This component also contains libraries supporting run-time object file (module) loading and linking. Thus, arbitrary RTEMS applications may be launched from it: application scope is not limited to the realm of data acquisition software. Section 3.5 details the IOMBase application.

4. *Instrumentation Control* - the software interface to the VME bus instrumentation. This component is manifest in the software artifact, *sis1100.obj*, an RTEMS-port of the Linux sis1100/3100 driver and its application programming interfaces. This driver is the subject of Section 3.6.

5. *Instrumentation Daemons* - servers providing interactive access to each of the VME, CAMAC, and HV hardware subsystems. This component will not be covered in any depth here.

6. *IOMReader* - the end-product resulting from the invocation of Lucid's code generation component, *VmeBuildReader et al*, upon a target file written using the Experiment Description language (EDL). The `IOMReader` is responsible for executing user-code in response to Trigger and Event definitions, and hence, governs data acquisition activity on the I/O Manager. This component is covered in Section 3.7.

With the exception of the *Instrumentation Daemons*, each of the components enumerated above are detailed in the remaining body of this chapter.

3.3 Network Boot-Loader

There are several options available for loading an executable image onto a target machine, including loading from a hard-drive, floppy disk, or optical media. Another option, and the method used to load the `IOMBase` application, is known as network *bootloading*, or simply *netbooting*. The primary advantage of network bootloading is, of course, that the target machine can be remotely located. However, this type of loading introduces additional administrative requirements that must be met.

Netbooting requires that the target machine obtain three critical pieces of information: 1) a unique network address (e.g. 192.168.0.5), 2) the location from where to obtain a bootable operating system, and 3) a tool to load and boot the executable image. The IOM relies on the freely available, open-source software known as *Etherboot*[1] to satisfy its netbooting requirements.

Etherboot itself consists several components, including the proper driver for the network interface card (NIC) of the IOM, DHCP and TFTP client software plus a UDP/IP network stack, as well as a service to load an operating system into the target's memory for execution.

The services of the Etherboot utility are used in this context to load the `IOMBase` component, which in turn initializes the RTEMS operating system, thereby providing an execution environment for subsequently loaded components.

For the system discussed in this thesis, TFTP and DHCP servers are implemented on the same Linux workstation hosting the Lucid software. In principle, this is not a

[1] Etherboot is available at http://rom-o-matic.net or http://etherboot.org.

requirement, nor is it required that both the TFTP and DHCP servers operate from the same host. The configuration used here is merely one of convenience.

For test purposes, the Etherboot program was installed on a floppy disk, with the I/O Manager's BIOS configured to search the floppy-drive for bootable material prior to searching elsewhere. For production use, it is recommended that the Etherboot binary image be burned into an EEPROM, suitable for installation into the 28, or 32-pin socket found on most network interface cards, and intended for exactly this netbooting scenario. Note, the BIOS settings must be again adjusted to account for such a configuration.

3.3.1 Dynamic Host Configuration Protocol (DHCP)

DHCP provides the means to assign configuration parameters to networked clients, including IP addresses, domain names, the addresses of other hosts providing further services (such as a TFTP server), and name-value parameters to be inserted into the client's operating system environment.

Dynamic host configuration protocol supports three modes of operation:

1. *Automatic* - the DHCP server allocates and assigns a permanent address to a client.

2. *Dynamic* - the server *leases* an address to a client for a pre-determined period.

3. *Manual* - a client's address is determined by a network administrator and the server simply conveys this to the client.

It is the *automatic* mode of operation that is utilized within the context of the Lucid DAQ.

To begin the process of obtaining a network identity, a client will broadcast a `DHCP_DISCOVER` packet over its subnet. If the hardware address (MAC) of the client is known to a DHCP server, it responds with a `DHCP_OFFER` packet. The client will then reply with a `DHCP_REQUEST`, signifying its wish to be assigned an identity. Finally, the server responds with a `DHCP_ACK` reply, containing the the client's IP address and other information pertaining to that client.

3.3.2 Trivial File Transfer Protocol (TFTP)

As suggested by the protocol's name, TFTP is a simple protocol for file transfer between clients and servers in a networked setting: only the most basic operations of reading, writing, and error reporting are supported.

TFTP utilizes UDP over IP to effect file transfers of 512-byte (or less) packets. Client requests to read/write a file will establish intention, if the request can be satisfied, file transfer will commence. Each successfully transmitted packet is followed by acknowledgment packet. Packets below the standard size are used to denote the final packet. In this fashion, flow control and reliable transport are established over the connectionless UDP.

3.4 Real-Time Operating System: RTEMS

A 1988 study conducted by the U.S. Army Missile Command examined military software applications found in embedded and distributed multi-processing systems. One of the products resulting from this study was the commissioned development of an open-source, real-time kernel, the Real-Time Executive for Multiprocessor Systems (RTEMS)[16].

RTEMS supports hard, real-time applications and is portable amongst several processor families and their associated peripheral devices. The operating system also features a re-entrant version of the standard "C" library, support for several filesystems (IMFS, NFS, FAT), and a port of the FreeBSD TCP/IP network stack.

The RTEMS execution environment has a *flat* architecture, both in terms of its address space and processor privilege-levels. This means that application and kernel code execute in the same address space and share complete freedom of instruction execution without restrictions: there is no "user-space/kernel-space" dichotomy, as is found in general-purpose operating systems. Thus, RTEMS is a *single-process, multi-threaded* (SPMT) operating system: all threads are children of a single process, sharing the same perspective of system memory.

As noted in Section 1.4.1, RTEMS and pSOS share the common feature of implementing a standards-based API, specified by the Real-Time Executive Interface Definition (RTEID). This specification describes the services that should be offered to applications by a real-time kernel. It is through these methods that application programmers create threads, send messages between them, and assign scheduling algorithms.

The fact that pSOS and RTEMS share an implementation of the RTEID specification means that system calls available in one operating system are also available in the other, and often with similar signatures. For example, using pSOS, the directive to create a new thread has the signature, `t_create(name,priority,stack,flags,taskID)`, whereas under RTEMS, the equivalent directive signature is:
`rtems_task_create(name,priority,stack,modes,attributes,taskID)`.

Similarities in function (method) signatures were present for *all* OS directives, and greatly simplified the porting of pSOS dependencies over to the RTEMS platform. Without

this one-to-one mapping of system calls between operating systems, the task of migrating code from one to the other would have been a much more complex task.

The RTEMS RTEID implementation is known as the *Classic*, or *Native*, interface. The Classic interface takes the form of system calls, or *directives*, partitioned into logically related sets under the domain of resource *managers.* Managers are not instantiable themselves, merely serving as organizational entities, into which logically related services are collected (i.e. an interface). Figure 3.3 illustrates the RTEMS Classic API, several resource managers, and their relationship to the key components of the RTEMS executive. A second API is also offered, encompassing a subset of the Portable Operating System Interface (POSIX) specifications. However, that interface was not used in this project and will not be covered here.

For each instance of a resource under a Manager's domain, there is an associated *object control block*. For example, creation of a new task will prompt the task manager to assign a new task control block (TCB) for the resource. Those kernel services utilized by multiple managers, such as scheduling and object control blocks, are provided by a component known as the *Super Core*, or *S-Core*, in the RTEMS nomenclature. The S-Core is a largely hardware-independent component and constitutes the nucleus of the RTEMS kernel. All hardware dependencies are isolated to *board support packages* (BSP). A BSP contains code targeting a specific processor family, the *i386* PC platform for example, and those peripheral devices associated with that architecture, such as timer, interrupt, and serial-port controllers.

Upon creation, an object control block is endowed with a user-defined name, thereby providing a meaningful association for the object in the application domain. More importantly, object control blocks are also assigned an `rtems_id` tag. This identification handle permits the Super Core to locate resource control blocks in a deterministic manner: object control block look-ups proceed according to an *O(1)* algorithm, thus enabling the RTEMS executive to guarantee predictable, bounded response times when locating internal resources.

RTEMS features 17 different Managers, supporting interfaces to memory, timer, and device control, system initialization and shutdown, and inter-thread communication. The sampling of Managers captured in Figure 3.3 play vital roles within Lucid's I/O Manager subsystem and their services are detailed below.

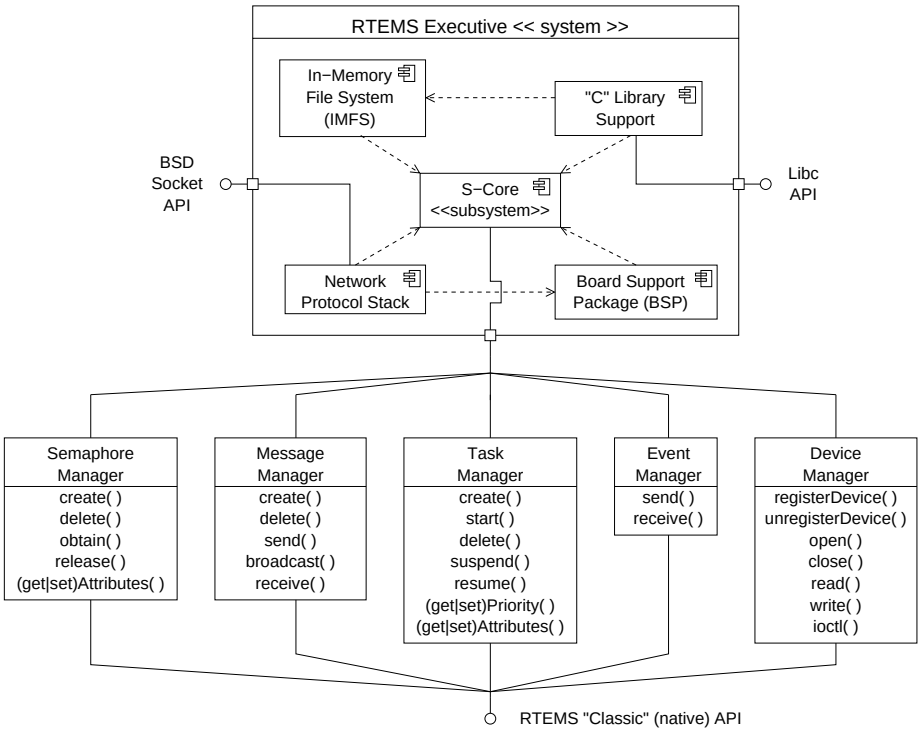

Figure 3.3: RTEMS executive *Super-Core*, it's major components and interfaces, and several managers of the *Classic* API: an implementation of the RTEID specifications.

3.4.1 Task Manager

The *Task Manager* interface provides a comprehensive set of directives for thread administration, including creation and destruction, priority adjustment, and suspension/resumption. Figure 3.4 illustrates the state set and their transitions in the RTEMS thread model. This illustration is an example of a *state diagram*, the familiar finite state machine (FSM), using the UML notation.

Internally, the RTEMS' scheduler maintains two, priority-sorted lists of threads: those in the *ready* state, and those in *blocked* state. The TCB attached to each thread contains the *context* associated with each, including its name, current priority and state, and the value of processor registers when the state was last preempted.

Tasks are scheduled according to a *priority preemptive* algorithm: the highest priority thread in the *ready* state is allocated the processor. Thus, higher priority threads will *preempt* those of lower priority. Processor monopolization by a single thread is quite likely with this algorithm: it is *unfair* by design. This assures that the most important thread (highest priority) may access the processor whenever required, and is clearly a crucial attribute in a system that must be able to guarantee response times.

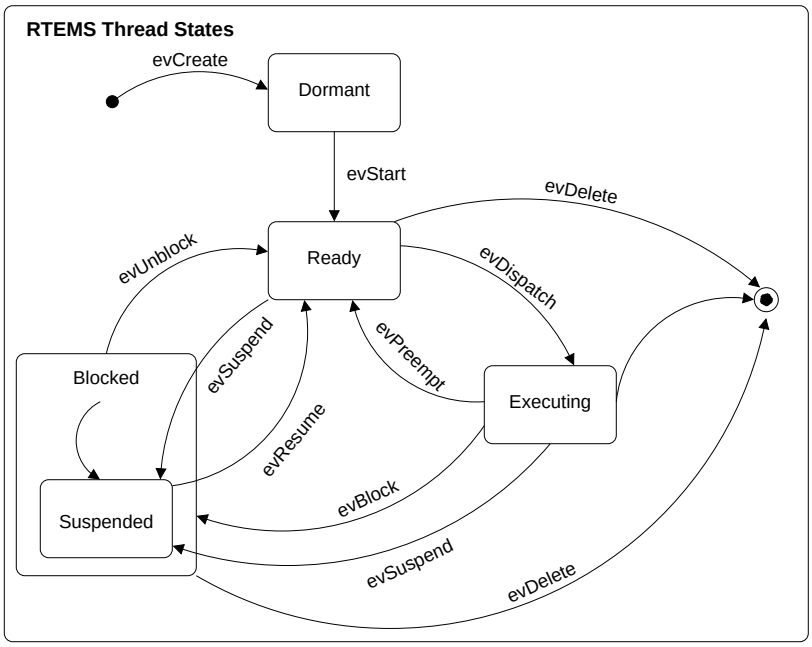

Figure 3.4: State diagram depicting the states and transitions of the RTEMS thread model.

3.4.2 Event Manager

The Event Manager is a high performance, inter-task messaging system, providing a low-overhead means of thread synchronization.

Consisting of only two operations, `send( )` and `receive( )`, this interface permits one task to send an *event set*, which would be received by another. An `rtems_event_set` is a 32-bit integer, where each bit is an *event flag* with a significance of the application developer's choosing. Threads may place themselves in the *blocked* state, waiting for a sibling to `send( )` an event set, thus "waking" the blocked thread and causing it to be placed back on the *ready*-list. Also, threads may wait for *any* or *all* events in an event-set before waking.

3.4.3 Message Manager

The RTEMS event mechanism does not permit the communication of information between threads beyond that contained by the presence of the event itself and the 32-bit event set. Also, that mechanism does not support the queueing of events: only the first of multiple event-sets sent to a thread will be received.

The RTEMS Message Manager solves both of the Event Manager's shortcomings by providing for the administration of named message *queues*. Variable length messages may

be enqueued by one task and dequeued by another. The presence of a single message in a queue is sufficient to awake any thread waiting on the queue, thus also providing a means of inter-thread synchronization.

Messages may be broadcast to several queues in unison and they may also be flushed from any queue. Messages are arranged in a queue by order of arrival (FIFO), but messages may also be marked as "urgent", enabling a last-in, first-out (LIFO) arrangement in the target queue. Similarly, threads waiting on the same queue will retrieve messages according to either a FIFO or priority-based schema. This permits higher-priority tasks to retrieve messages from a queue before their lower-priority siblings.

3.4.4 Semaphore Manager

A semaphore is a mechanism used to control access to a resource shared amongst multiple threads. In the RTEMS implementation, a semaphore is a *counting* variable: assuming it is initialized to a positive value, a thread *obtaining* the semaphore will cause it to decrement. When the thread *releases* the semaphore, its value is incremented. Any thread attempting to obtain a null-valued semaphore will block waiting and be added to a per-semaphore wait-queue, ordered according to FIFO or priority policies.

Within RTEMS, semaphores of two varieties are available:

1. *Binary Semaphore* - coupled with several distinct behaviors, restricting the semaphore's values to zero or one produces a *mutual exclusion semaphore*, or *mutex*. By encompassing access to a resource in `obtain( )`/`release( )` calls to a mutex, only one thread is permitted access at a time, thereby ensuring resource integrity.

2. *Counting Semaphore* - semaphores initialized to values greater than one. These are commonly used to control access to a *pool* of resources, for example a group of printers or data buffers, by initializing the semaphore to the same value as the number of resources in the pool. A thread wishing to access a resource must first obtain the semaphore, thereby decrementing it. When the counting semaphore reaches a null value, threads will block awaiting the resources' availability.

Mutexes and counting semaphores possess an awareness of the notion of *ownership*: they are aware of the identity of the thread that has successfully obtained them, and that thread is said to *own* the semaphore. In the case of mutexes, only the owning thread may release the semaphore. This policy is required to guarantee the concept of mutual exclusion. On the other hand, counting semaphores may be released by *any* thread, thereby supporting the role of those semaphores in synchronization schemes.

These two types of semaphore are also behaviorally different in their response to attempts by threads to re-acquire an already *owned* semaphore. In the RTEMS nomenclature, this scenario is known as semaphore *nesting*. Mutex-type semaphores permit multiple acquisitions by their owning thread, with the stipulation that each obtain must be paired with a corresponding release in order to eventually make the mutex available for competition from other threads. However, counting semaphore behavior is such that attempts by the owning thread to re-obtain a semaphore result in that thread blocking until another thread issues a release of that semaphore. Again, this behavior supports the use of counting semaphores as synchronization tools.

The use of mutexes in a system that schedules threads according to a priority-preemptive algorithm, as found in the I/O Manager, introduces certain hazards. To combat these hazards, mutex semaphores may be endowed with special behaviors. These dangers and the accepted workarounds are discussed next.

3.4.4.1 Priority Inversion

The serialization of access to resources via mutex usage creates so-called *critical* sections of code. While preventing simultaneous access, this approach invalidates the concept of absolute preemptibility: i.e. that the highest-priority thread in the *read*y-*state* will execute. However, if the critical section's duration is sufficiently brief, the disturbance to global schedulability may be minimal [15].

A priority-based scheduling scheme, coupled with critical sections of code guarded by a mutual exclusion semaphore (i.e. a mutex), may lead to a *priority inversion* scenario. This situation occurs when a high-priority thread attempts to obtain a mutex currently held by a low-priority thread and must block waiting. This situation may be further complicated if a mid-priority task preempts the low-priority task holding the mutex. As the low-priority thread cannot complete the critical section, and thus release the mutex, the high-priority thread is unable to progress and may become blocked indefinitely. If the number of threads with priorities intermediate of the high and low-priority threads is sufficiently large, this scenario may lead to a condition known as *unbounded priority inversion*, where the high-priority thread becomes delayed indefinitely.

Acceptable schema for avoiding unbounded priority inversion involves a temporarily priority elevation for the lowest-priority task holding the resource in contention. This approach guarantees that the holder of the mutex will be allowed to execute, and hence, eventually `release( )` the mutex. While several variations exist, most are based on two algorithms known as *priority ceiling* and *priority inheritance*.

The priority ceiling algorithm elevates the priority of the low-*priority* thread to that

of the highest-priority task that *may ever* contend for the shared resource, whereas the priority inheritance algorithm will assure that a low-priority task in possession of a lock will inherit the highest priority of any task that blocks waiting for the shared resource. Of these algorithms, priority inheritance is the more attractive, as that algorithm is determined entirely at run-time and without developer intervention. On the other hand, the priority ceiling algorithm requires the developer to manually configure the "ceiling priority", which may change as the system evolves.

According to one paper, there are two critical aspects to be aware of when using priority inheritance within a system: 1) the algorithmic complexity required for implementation may lead to poor worst-case temporal behavior, and 2) proper system design at the fundamental level may alleviate the need to use priority inheritance schema [17]. However, given that it is generally impossible to foresee all conditions under which code will be used, it seems prudent to use priority-inheritance where appropriate in order to protect resources in the general case.

3.4.5 Device Manager

The RTEMS Device Manager[2] provides a mechanism for dynamically registering device drivers with the in-memory filesystem (IMFS). This permits applications to access devices as thought they were files resident in the system. This "device-is-a-file" API is examined in more depth in Section 3.6.

Each device must provide a `DriverTable` structure containing the operations which will be mapped by RTEMS onto the appropriate filesystem access routines, `open( )/close( )`, `read( )/write( )`, and `ioctl( )`.

Additional information that must be provided include the device's *major* and *minor device numbers*. The major number serves as an index into a global array of devices maintained internally by RTEMS. The device minor number serves as an identifier of a particular device instance, in the case of multiple like-devices sharing common driver code.

3.5 IOMBase

So named because it forms the foundation from which arbitrary applications may be launched, the *IOMBase* component is responsible for initializing several key services re-

[2]This manager is properly known as the I/O Manager in the RTEMS literature, but the name has been changed here to avoid confusion with the Lucid software component of the same name.

quired by other components. As indicated in the deployment view (Figure 3.2), IOMBase itself is comprised of several artifacts, two of which are software packages developed at the Stanford Synchrotron Radiation Laboratory (SSRL)[3]: *GeSys* and *Cexp*. Each of these components are discussed in turn below.

3.5.1 The Generic System (GeSys) Artifact

The *Ge*neric *Sys*tem artifact, or GeSys, is responsible for initializing the RTEMS run-time environment and core services, including the network protocol stack, the TFTP-client driver and in-memory file system (IMFS), the I/O Manager's clock via the network time protocol (NTP), and network-based logging via the *syslog* facility. Most importantly, GeSys is also responsible for creating an environment conducive to loading and linking object files into an executing RTEMS system.

Executable images differ from object files in that the former have had any references to externally defined code resolved by the linker component of the compilation process, while the latter has not yet had those references resolved. Loading additional, complete executable images into an already executing system is not an option because of the high likelihood of introducing duplicate code and symbols, which may result in undefined behavior. Thus, run-time loading of code requires object files be loaded onto the executing system, subject to the caveat that code referenced by an object file must exist *a priori* on the running system.

To satisfy this dynamic loading requirement, the GeSys build system uses special techniques and tools to enforce linking code into its executable image even though the code may not be referenced anywhere within the image. In this way, the developer can force libraries of code into the GeSys executable that may be referenced by object files *yet to be loaded* into the executing system. For example, even if the `printf( )` function is not used by any code within the GeSys image, forcing that code to be included in the image allows for object files that *do* utilize `printf( )` to have that link resolved when the object is loaded. It is the developer's responsibility to decide what libraries to include, thereby tailoring the GeSys image according to memory-footprint or flexibility constraints.

3.5.2 The Cexp Artifact (Dynamic Linker)

Implicit in the preceding discussion is the existence of the tooling required to facilitate both the *loading* of object files into the memory space of an executing RTEMS image,

[3]The *GeSys* and *Cexp* components are available from http://www.slac.stanford.edu/~strauman/rtems/software.html

as well as the *linking* of unresolved references required by those object files. In conjunction with the RTEMS TFTP driver, the other SSRL-developed tool used within the I/O Manager, known as *Cexp*, provides the required loading and dynamic linking capabilities.

The C-expression interpreter, or Cexp, is a feature-rich library of utilities, offering much more functionality than suggested by its name:

1. *Symbol Table Management* - although an application's symbol table must be produced and made accessible to Cexp, doing so permits access to *all* symbols (variables and functions) in an executing program. Methods are provided by both Cexp and GeSys for producing the system symbol table.

2. *C-Expression Interpretation* - more widely known as a *shell*, this utility permits the interpretation and invocation of arbitrary C-language expressions. Similarly to the *bash* and *csh* shells, popular on Linux systems, a shell may be used interactively, via the console or a via a remote *pseudo-terminal* program (e.g. telnet), or the shell may be used to autonomously parse a script and interpret its contents.

3. *Object File Linking* - arguably the most important feature utilized within the I/O Manager, Cexp resolves undefined object file symbols using the system symbol table. The newly loaded module's symbols are then added to those maintained in the system's table, thus permitting Cexp to track inter-module dependencies.

The Cexp libraries are linked into the GeSys application, thus making the services enumerated above available within the IOMBase application. Once GeSys has completed initialization of the RTEMS network services, it fetches the names of the system symbol table (*IOMBase.sym*), system start-up (*st.sys*) and user scripts (*lucid_objs.sys*), from the execution environment. At this point, Cexp is launched with *IOMBase.sym* and *st.sys* passed in as arguments to the call, `cexp_main( )`. Cexp then fetches the system symbol table and start-up script via the TFTP driver. The start-up script, *st.sys*, defines abbreviations for several frequently used commands and instructs Cexp to load and execute the telnet daemon module file. This permits remote, command-line access to the I/O Manager.

The user script, *lucid_objs.sys*, is then loaded and parsed. This script instructs Cexp to load and initialize the object file modules containing the components, *Instrumentation Control* and *Instrumentation Daemons* (see Figure 3.2). It is at this point that the *AppLoader* artifact is also retrieved and started.

AppLoader is the tool used to retrieve the name of a Server-Side DAQ application from the Linux machine hosting a Lucid experiment session (see Figure 3.2). This network application simply listens for packets containing a string matching the pattern, *filename.pc386*. When this information is received, and assuming a module of the same name

has not already been loaded, AppLoader directs Cexp to fetch and load the module into the system. If a module of the same name already exists in the I/O Manager's symbol table, it is first unloaded prior to loading the new module of the same name.

3.6 Instrumentation Interface

The sis1100/3100 VME-PCI bridge device is a critical enabling component within the context of the Lucid data acquisition system. It is a vital link on the data path between instrumentation hardware and experimenter, directly responsible for moving data to and from VME and CAMAC modules at the behest of an operator.

The original driver supplied with the device was available in forms suitable for use with the 2.4 and 2.6 versions of the Linux kernel. Porting the Linux driver for use under RTEMS required first identifying the Linux-dependencies in the source code, then re-designing or re-implementing those portions, keeping in mind the requirements of, and services offered by the RTEMS environment. Details particular to writing device drivers for use with the Linux kernel are well documented elsewhere, and will not be discussed here (for instance, see [12]).

The driver-kernel relationship is one of many facets and inter-dependencies. Figure 3.5 is a *concurrency and resource view* of the `Sis1100Device` object, the component housing the functionality of the RTEMS-port of the sis1100/3100 device driver. This type of diagram illustrates the <<active>> objects (threads or other execution contexts) and the resources with which they interact.

While the driver provides low-level device control and logic, the kernel provides many essential tools and services upon which drivers depend:

1. *System Integration* - In order for a driver to be able to fulfill its role of device control, it must be integrated, or installed into the operating system environment. This procedure varies by operating system, but typically requires the driver to undergo some form of *registration* with the kernel.

2. *Bus Access* - The sis1100/3100 is a PCI device, and therefore requires RTEMS to provide methods to access the various address regions of the PCI bus architecture: I/O, memory-mapped, and configuration spaces.

3. *Interrupt Infrastructure* - To avoid incurring delays while the CPU waits for events from a peripheral, many devices utilize asynchronous signals known as *interrupts* to

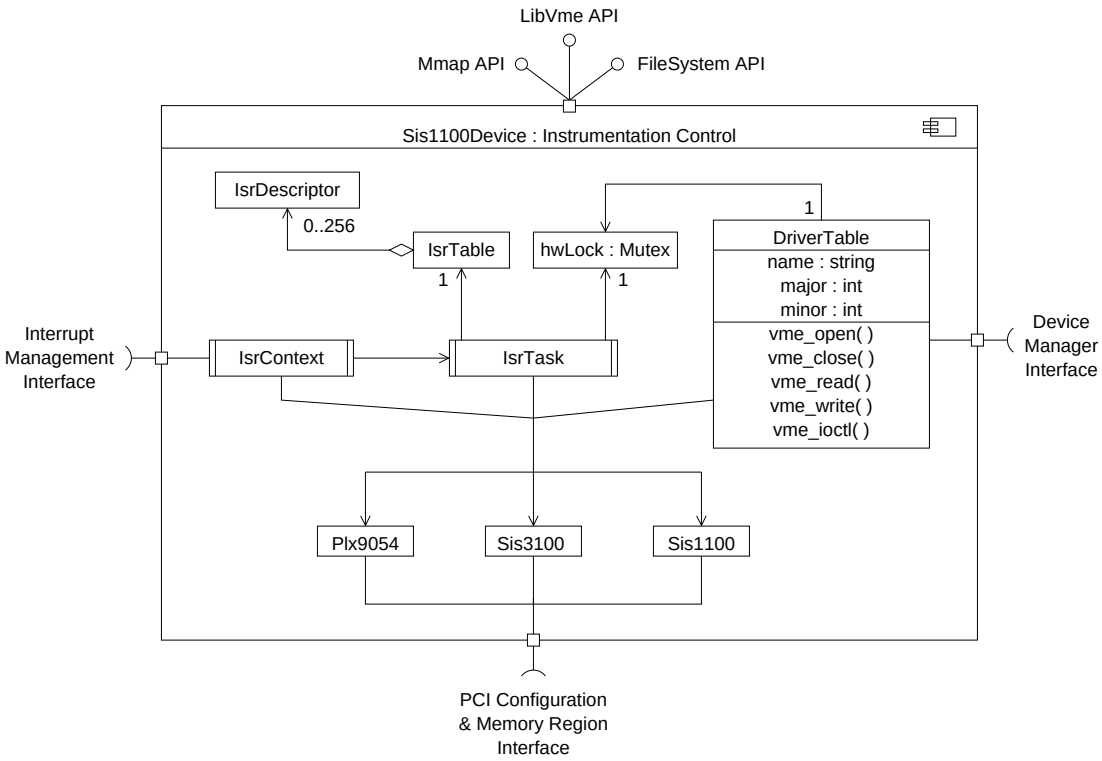

Figure 3.5: Structural diagram of the *Instrumentation Control* software component, realized by the `Sis1100Device` class. Note, the <<active>> objects (i.e. thread or interrupt context) are denoted by the double-barred boxes.

gain the processor's attention. An OS kernel is responsible for routing interrupt signals to the appropriate context, or handler, in addition to providing an infrastructure where drivers may register a software context in which to service an interrupt, .

4. *Scheduling* - Device drivers may require one or more distinct execution contexts, plus they often require the ability to schedule activities periodically, or otherwise access the timing facilities provided by the kernel.

5. *IPC* - The ability to send and receive signals is fundamental to synchronization and communication, and the Sis1100Device requires support for both synchronization and mutual exclusion.

6. *API* - Unix-like operating systems, such as RTEMS, feature an application program interface, wherein devices are abstracted as *files*: they may be opened, closed, read from, and written to. While device drivers must expose their services according to this paradigm, the OS kernel must route applications' filesystem accesses to the appropriate driver routine. However, devices with a complex set of features, such

as the Sis1100Device, may be better served by providing an alternate, more flexible API.

Each of the items enumerated above are present in some form in Figure 3.5, and are further detailed in the following discussion.

3.6.1 System Integration

Within RTEMS, driver registration is a two-step procedure, resulting in the integration of the driver into both the executive and the in-memory filesystem (IMFS). By so doing, the kernel is able to simultaneously map the driver's routines contained in its `DriverTable` onto standard C-library directives *and* make those services available to applications via the filesystem node representing the driver.

An initial call using the directive, `rtems_io_register_driver( )`, prompts installation of the driver's `DeviceTable` into an array of like-structures maintained by the RTEMS Device Manager, and known as the `IODriverAddressTable`. As a side-effect of issuing this directive, a *device major* number is allocated for the driver based on the next available entry in that table. The device major number serves as an index into the `IODriverAddressTable`, used by the Device Manager to locate the appropriate driver routines when applications access the driver's filesystem node.

Having installed the driver's `DriverTable`, the Device Manager will then invoke the driver's initialization routine from the table, if one exists. Typically, this initialization method would allocate and configure any resources required by the driver, such as threads, semaphores, and an interrupt service routine.

The final step to complete the driver registration process makes it accessible to application code by creating a file, or *node*, in the IMFS representing the device driver itself. Filesystem device nodes are created by invoking the directive, `rtems_io_register_name( )`, with arguments specifying the filesystem path, major, and minor device numbers. By convention, device files are created in the */dev* directory. In this particular case, the `Sis1100Device` is accessible from the node, */dev/sis1100_x*, where "x" is *minor device* number, representing one of possibly multiple such devices.

3.6.2 Bus Access

Assuming that the hardware platform is equipped with a PCI bus, an operating system kernel must provide facilities to access PCI configuration space. It is actually a *requirement* that the system's firmware, or BIOS, provide the software necessary to access PCI

configuration space. Required functionality is stipulated in the *PCI BIOS Specification* document, as are recommended techniques for providing those services [18].

In particular, methods must be created to permit reading and writing 8, 16, and 32-bit data, from and to PCI devices' configuration memory regions. This permits the OS kernel to *probe* PCI configuration space, thereby discovering what devices are present, where in I/O or memory-space their control and status registers are located, and how those devices' are mapped to the system's programmable interrupt controller (PIC). RTEMS provides platform-independent access to PCI configuration space with the methods,
`pci_find_device( )` and `pci_read/write_config_byte/word/dword( )`.

Recall from the discussion on PCI addressing (Section 2.4.2), devices on that bus are accessed by exposing, or *mapping*, their status and control registers into either an I/O or memory-mapped region of CPU-addressable space. In the `Sis1100Device` driver, the sis1100, sis3100, and plx9054 devices all map their registers into separate, memory-mapped regions.

The three, bottom-most objects of Figure 3.5, are software abstractions representing the register-sets of the three, discrete devices comprising the VME-PCI bridge. Each of the `sis1100`, `sis3100`, and `plx9054` classes are structures reflecting the composition of the device registers they represent. That is, each structure field *overlays* a corresponding hardware register. Thus, manipulating the bits of a particular structure field will directly manipulate the appropriate bits of the hardware register it overlays, thereby modifying or providing information regarding the device's behavior.

It must be noted that while RTEMS does provide platform-independent methods to access hardware I/O-space, it offers no such methods for memory-mapped accesses. Perhaps it is an implicit assumption that developers would simply define their own methods using pointer-based techniques. However, a standardized, platform-independent means to affect bus-access would eliminate the myriad implementations found in the RTEMS source by providing a consolidated interface for driver and application usage. For example, such an interface as is found in the `bus_space` and `bus_dma` interface of the NetBSD operating system [19].

3.6.3 Interrupt Infrastructure

For any real-time application, the operating system facilities for providing timely response to external stimuli assume a critical role. There are two means by which software may be alerted to events, or state changes, originating in peripheral devices:

1. *Polling* - in this case, software can repeatedly inquire, or *poll*, the device for state

information. While simple in principle and implementation, polling is a wasteful activity in terms of processor resource consumption: inquisition time may be better spent elsewhere.

2. *Interrupts* - this method arranges for a capable device to issue a physical signal indicating the presence of a change of state. This type of asynchronous messaging is known as an *interrupt*, as devices must route these signals through a hardware infrastructure whose purpose is to "interrupt" the CPU from its current processing task.

Once interrupted, the CPU switches from its current context (thread) to a special-purpose *interrupt-context*, where it will silence the hardware signal and, in general, execute an *interrupt service routine* (ISR) specified beforehand by the application developer.

While interrupts are more efficient in the sense that no processing cycles are consumed waiting for a device's change of state, their use may complicate software design. For example, interrupt service routines are quite restricted in the permissible set of operations they may execute. In particular, an ISR may not issue any directive (system call) that might cause the interrupt-context to become blocked while waiting for the availability of a system resource: if the interrupt-context were to block, the processor would not be able to return to the "regular" execution context, hence preventing the entire system from making any progress.

In order to return from interrupt-context to regular-context as quickly as possible, an ISR is typically designed to perform, as quickly as possible, only those tasks deemed essential to service the interrupt. Any additional processing required to service the interrupt is often deferred to a special thread, dedicated solely to interrupt support. For the purposes of this thesis, this type of thread will be referred to as an *interrupt service thread*, or IST.

While compromising the amount of useful work that may be performed in an interrupt-context, minimizing ISR duration yields at least two positive aspects:

1. minimizing the duration of an ISR minimizes a system's response-time to other possibly higher-priority interrupt signals, and

2. that period during which certain system services are forbidden has also been minimized.

The `isr` and `isrTask` objects, shown in Figure 3.5, are key items in the driver-kernel relationship. In particular, it is the `isrContext` object that most immediately executes the `Sis1100Driver`'s response to hardware interrupts originating from VME modules,

changes in fiber-optic link status, and DMA completion events. To enable interrupt response at all, the `Sis1100Driver` first requires an interface to the RTEMS interrupt management infrastructure.

The state of RTEMS' interrupt infrastructure is somewhat confused, being divided along lines according to whether a given board support package (BSP) subscribes to the "old" or "new" exception processing models. However, both models implement an Observer design pattern (see Appendix C.1): clients subscribe to be notified when a particular interrupt is received, registering a method to be called when that event occurs.

The "old" model of exception processing utilizes an *Interrupt Manager*, similar to the other service interfaces of the Classic API: applications use the directive, `rtems_interrupt_catch( )`, to attach an interrupt service routine to a hardware interrupt vector.

Although the reasoning behind the paradigm shift is unclear, the PowerPC and i386 ports of RTEMS use the "new" exception processing API. While this API offers additional flexibility, in the sense that applications may associate functions with the actions of ISR installation and removal, arguably the most useful feature of the "new" interrupt API is the ability to pass an arbitrary argument to the interrupt service routine, as evident by the ISR's signature, `rtems_isr handler(void* arg)`. Within the `Sis1100Device`, the function argument is used to pass a pointer to the driver-object itself. In the case of multiple sis1100 devices, this affords the driver's ISR the ability to access the appropriate interrupting device without having to search all such devices.

Prior to entering the interrupt service routine registered for a device, RTEMS performs a *context switch* to an *interrupt context*. Interrupt context differs from that of a thread in that an interrupt context is forcibly entered based on hardware signals, not according to the invocations of the operating system scheduler, as is the case for thread context. Since a thread may be interrupted at any time, it is the responsibility of the operating system to persist thread state across an interrupt occurrence. Also, note that since an interrupt context is not "returned to", or resumed, in the same sense as thread-based contexts are, an interrupt context cannot persist state between invocations.

Immediately upon entering the interrupt context, RTEMS executes an interrupt *prologue*. The prologue is responsible for persisting the interrupted thread's CPU register-state while the interrupt servicing is in progress, as well as silencing the interrupt at the level of the programmable interrupt controller, or PIC, situated in the SouthBridge chipset of the PC (see Figure 2.7). Following execution of the registered ISR, the RTEMS interrupt infrastructure will execute an interrupt *epilogue*. The epilogue is responsible for re-enabling the interrupt at the PIC and, if necessary, invoking the RTEMS scheduler. This

last step is critical, as an ISR will typically issue directives to unblock a waiting thread, which must then be scheduled for execution based on its priority.

Finally, it must be noted that the strategy of interrupt *prologue, service routine, epilogue* is not unique to RTEMS: NetBSD, FreeBSD, and Linux systems all employ a similar design pattern.

3.6.4 Execution Context

Within RTEMS, and with the exception of device driver interrupt service routines, driver services execute in the context of their invoking thread. Thus, if a host thread makes a call to a device driver causing it to issue a blocking directive, so too does the host thread become blocked. This scenario is similar to that found on Linux or BSD-based operating systems utilizing processor "privilege-levels", manifest as the user-space/kernel-space dichotomy: single-threaded user applications invoking driver services cause a context switch to kernel-space, where the driver code is actually executed, thus blocking the application until the device-bound operation completes.

In some instances it becomes necessary to decouple driver context from that of the caller. For example, additional interrupt processing may require a separate context from which to launch operations. This is indeed the case with the `Sis1100Device`, in which the `isrTask` thread serves as an extension to the `isrContext`: the thread represents a context from which to launch interrupt-related operations that potentially may block. The `isrTask` permanently waits for events issued by the `isrContext`. In its present configuration the `isrTask` deals with only two types of events: VME module-based interrupt requests, and changes in the state of the PC-to-VME fiber-optic link.

In the case of VME module interrupts, additional processing is required beyond what may be done from an `isrContext`: the VME interrupt acknowledgment (IACK) cycle, and the client thread's interrupt service routine must be executed from a thread context, as these activities could potentially involve blocking-directives, or require significant processing time. In addition, the IACK sequence requires the `isrTask` to access several hardware registers on the sis1100 PCI module. To prevent the `isrTask` from interfering with operations initiated by another thread, it must first obtain the mutex, `hwLock` (see Figure 3.5). Since the mutex may be held by another thread, the process of obtaining it may lead to the `isrTask` blocking.

3.6.5 Inter-process Communication

Internally, the `Sis1100Device` requires inter-process communication mechanisms in the form of mutual exclusion and signal delivery and reception mechanisms.

The `isrContext` object relies on the mechanism provided by the RTEMS Event Manager to send synchronizing signals to receptive objects. In particular, the activities of the `isrTask` are driven by events sent from the `isrContext` of the `Sis1100Device`.

Client threads utilizing VME block transfer operations are also implicitly dependent upon the delivery of events issued by the `isrContext`. In this case, the events denote block transfer completion: the `Sis1100Device` has finished transmitting or receiving a block of data via its DMA engines.

Many devices require a means of preventing state corruption incurred when multiple threads attempt to modify data structures that must remain cohesive over the duration of an operation. For example, when transferring data, the `Sis1100Device` must maintain several hardware registers in a state particular to the transfer type, and it must maintain that configuration over the duration of the transfer. In other words, the operation must execute *atomically*: i.e. without disturbance or division.

To prevent simultaneous access to sensitive data structures, the `Sis1100Device` employs a mutual exclusion semaphore, or mutex. This is the `hwLock` object of Figure 3.5. This mutex guards the entrance to atomic, or so-called *critical* regions of code within the driver: before a thread may enter the critical region, it must first obtain the mutex, or block waiting until it becomes available. Upon exiting the critical region, the mutex is released. In this fashion, mutually exclusive access to atomic sections of code is strictly enforced, thereby preventing corruption of critical data structures.

It should be noted that client code utilizing the the VME block transfer capabilities will block waiting for completion *while holding the* `hwLock` *mutex*. This is generally considered a poor practice, as other threads may be indefinitely prevented from gaining access to resources protected by the mutex. However, the nature of the sis1100/3100 device is such that this design is unavoidable: if the thread performing DMA were to release the mutex prior to waiting for the operation to complete, the possibility is introduced whereby another thread may corrupt structures critical to the DMA operation in progress. It is important to also note that *asynchronous*, DMA-based I/O is impossible with this device, and for the same reason: the nature of the device is such that, once a DMA-based operation is commenced, the initiating thread must retain *sole ownership* of the device for the duration of the operation.

However, to combat the possibility of indefinite postponement while the `hwLock` mutex

is held, the lock-holding thread waiting for the DMA operation to complete will only wait up to a user-configurable maximum period before terminating the activity and releasing the mutex for competition from other threads.

3.6.6 Application Programming Interfaces

The application-programming interface provided with the Linux/NetBSD version of the sis1100 device driver was based on a three-layered design. This API was leveraged for inclusion in the RTEMS version of the driver.

The lowest layer consists of a set of private routines used by the driver itself to fulfill services such as reading, writing, interrupt management, and registration with the host operating system. Use of these routines is transparent to applications, as they are invoked only by the second layer of the API. This second layer is formed by those routines which map the driver's mechanisms into the traditional file-system API of Unix-like operating systems. The highest API layer is formed by encompassing the file-system API in adapter routines featuring more "user-friendly" function signatures. Each interface is detailed in the following discussion.

3.6.6.1 File System API

This API encompasses the traditional Unix model of abstracting character-devices as a sequence of bytes: i.e. a file. Applications first obtain a file-descriptor via the `open( )` system directive, which is required for all subsequent device transactions:

```
int fd = open("/dev/devicename", flags);
```

Using the file descriptor, an application may then set the device to begin reading/writing from a position in the sequential byte-stream. Practically, this amounts to instructing the driver to direct the transfer at an initial offset, or address, in VME space. For example:

```
off_t offset = lseek(fd, 0xFFFF0000, SEEK_SET);
```

By default, the sis1100 driver is configured to direct VME access to the A32/D32 address-space, so the `lseek( )` call, above, will enable such access beginning at the VME address, 0xFFFF0000. Access to other address-spaces may be arranged by an appropriate `ioctl( )` directive prior to issuing the `lseek( )`.

Finally, the transaction itself may accomplished by providing a pointer to an appropriate buffer, the number of bytes to be moved in the transfer, and finally issuing a `read( )/write( )` call:

```
int n = read(fd, &aBuffer, numBytes);
```

The driver will then carry out the appropriate transfer, using the fastest method at its disposal based on the number of bytes to be transfered: i.e. using either single-cycle, or DMA-type accesses. This decision is based on fact that some overhead is necessary to configure the device for DMA-transfers and hence, it may be more economical, in terms of the duration required, for transfers of small quantities to use single-cycle VME access. This idea is covered at length in Chapter 7 (performance measurement analyses).

3.6.6.2 LibVME API

The file system API, discussed above, becomes somewhat awkward to use in the context of frequent and variable VME accesses: at least three separate methods must be invoked. In an effort to combat this difficulty, another distinct library of routines was provided with the Linux-version of the sis1100/3100 driver, and is manifest in the RTEMS-port of the driver in the software artifact, *libsis1100_api.a*. Alternatively, this application programming interface is also known as the *LibVME API*.

The routines of this API largely consists of adapters, or "wrappers", around the various `ioctl( )` calls required to configure the driver for different access methods to VME spaces. For example, after obtaining access rights to an instance of the driver via an `open( )` directive, the LibVME routine to execute a single-cycle write to A32/D16 space is:

```
int error = vme_A32D16_write(fd, vmeAddress, data);
```

and, a block transfer from an A24/D32 device is:

```
error = vme_A24_BLT32_read(fd,vmeAddr,&buffer,nWords,&nWordsReturned);
```

where the former function takes, in addition the file descriptor returned from `open( )` and the VME base-address, a pointer to a buffer where returned data may be stored, the number of words desired in the read, and returns the number of words *actually* read.

3.6.6.3 Memory-mapped API

As shown in the structured class diagram, Figure 3.5, the `Sis1100Device` features a third interface by which applications may perform read/write accesses to the VME bus; the memory-mapped, or Mmap API.

The device features a table of 64 individually-configurable memory regions, or "windows". Each of these 4 megabyte-wide windows map a region of VME address-space into an equally sized region of the host PC's address space. Thus, each region provides a "window" into VME address-space, which is accessed by software using simple pointer dereferencing operations.

In order to use this mechanism, application code must first obtain access to an instance of the `Sis1100Device` using the `open( )` system-call. Next, a routine provided in the LibVME API is required to configure a window for use. This routine has the prototype:

```
int vme_set_mmap_entry(int fd, uint32_t vmeBaseAddr, uint32_t am,
uint32_t hdr, uint32_t size, uint32_t *pcBaseAddr);
```

where, `vmeBaseAddr` is the desired base-address in VME space and `pcBaseAddr` is a pointer to be returned by the function, used by the application for VME address-space accesses. Additional required parameters include the VME address modifier assigned to the region, and the desired length of the region (up to 256 MB). The `hdr` parameter, or header, is required by the sis1100 in order to fully qualify the type of transactions permitted through the memory-mapped region. This parameter is essentially a bit-field, whose structure may be found in the sis1100/3100 documentation.

3.6.7 Interrupt Interface

A two-tiered approach to interrupt service was implemented for the RTEMS-port of the sis1100 device. This choice was motivated by both the virtues of such a design and also by the requirements dictated by nature of the device itself.

In Chapter 2.6, and also in Section 3.6.3 above, attention was drawn to the design of the sis1100 VME-PCI interface being such that the VME interrupt acknowledgment sequence had to be relegated to the domain of software. Furthermore, the IACK sequence *cannot* be performed from an interrupt-context (ISR), as the device may be in a state such that the transactions required to perform the IACK could disrupt a transaction in progress, or even cause the device to enter a forbidden state, or *wedge*. In light of these restrictions, one of the primary functions of the `Sis1100Device`'s IST is to execute the VME IACK sequence on behalf of the device driver, thereby discovering the vector identity of interrupting VME modules.

Illustrated in Figure 3.6, the interrupt-handling facilities of the `Sis1100Device` are designed according to an *Observer* pattern (see Appendix C): client applications register their interest in a particular interrupt vector by registering an `IsrDescriptor` object

with the driver. The vector itself serves as an index into an array of `IsrDescriptor` objects, represented in Figure 3.6 by the `isrTable`. A key feature of these descriptors is their `execIsr( )` method, invoked by the IST after the appropriate interrupt has been acknowledged. This is an example of the *push* paradigm, discussed in the previous chapter: the interrupt service thread *pushes* data to descriptor object, which executes some actions on the clients behalf.

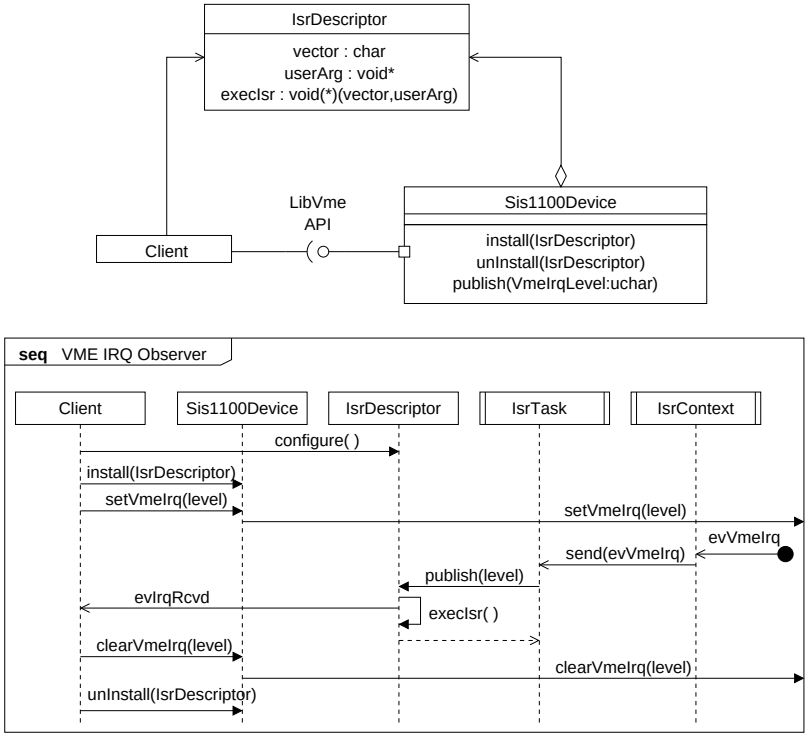

Figure 3.6: Client-`Sis1100Device` interaction during instrumentation interrupts. Note, the sequence shown here assumes the interrupting module is of the ROAK variety.

Typically, the `execIsr( )` method would simply clear the interrupting condition on the instrumentation module, and perhaps deliver an event to a master thread for further action. However, the `execIsr( )` method is subject to two caveats as a result of its location in a time-sensitive region of code. First, the method should not issue any directive that may block, although it is certainly free to do so. Blocking at this point in the driver's execution sequence would delay its response to any other pending interrupts. Second, if the interrupting VME module is of the *RORA*-type, the method *must* access the module to silence the interrupt. If this action is not taken, the interrupt response sequence will enter into an infinite loop, as the interrupt signal will never terminate.

3.7 The IOMReader Component

As portrayed in the deployment view of Figure 3.2 and discussed in Section 3.2, the `IOMReader` component is the result of linking a user's compiled Experiment Description file with the data collection template contained in the artifact, *libIOMReader.a*. The structure and behavior of the latter artifact is the topic of this section.

Together, the `IOMReader` and Client-side daemons realize a variation of the Proxy design pattern (see Appendix C), publishing event types according to an experimenter's specification, *pushing* data from the I/O Manager to Workstation, where the *pull-model* of data dissemination is used by clients to obtain event data from Lucid's Reader daemon (see Appendix C.2).

Figure 3.7 illustrates the collaborative objects participating in this design pattern. This figure is a snapshot of activities during an active data acquisition session, as initiated by an experimenter from the *gxlucid* interface: i.e. the "Online-mode" of Lucid. Recall from Section 1.3, when in Online-mode, data is acquired by and forwarded from the I/O Manager. The `Reader` daemon, executing on a Linux workstation, then places `DataRecords` in a shared-memory buffer, and signals interested `Consumers` that new data is available. In Offline-mode, the `Reader` sources data from a file previously obtained during an online-session.

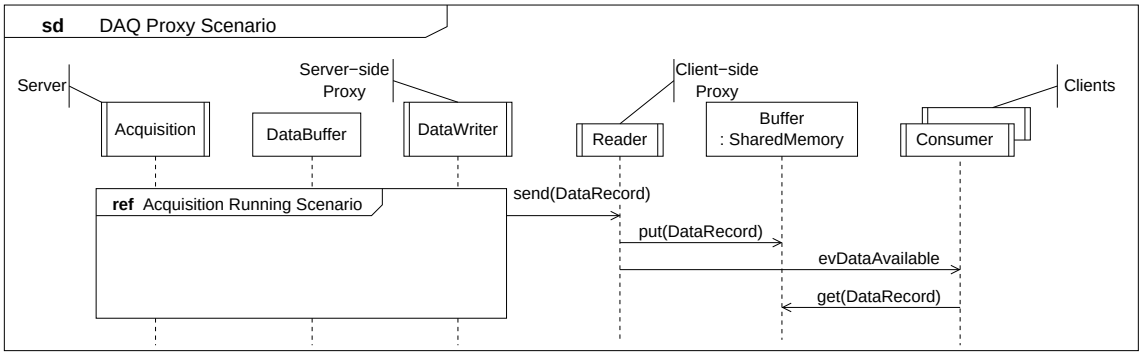

Figure 3.7: Collaborative objects participating in the Proxy design pattern implemented within Lucid.

The activities of the server-side objects are not explicitly shown in Figure 3.7. In their stead is a reference to a distinct sequence diagram, the *Acquisition Running Scenario*, denoted by the **ref**, or *reference* operator of that interaction sequence. The notational ability to decompose sequences into arbitrarily-nested fragments is a feature introduced with the UML 2.0 specification to address issues of scalability within sequence diagrams [15]. The **sd** operator identifies the primary interaction fragment (i.e. *DAQ Proxy Scenario*),

whereas the **ref** operator names a referenced sequence diagram shown elsewhere. In this case, the *Acquisition Running Scenario* is shown in Figure 3.10, and discussed in Section 3.7.2.

The implementation of a Proxy pattern here affords transparent data and event distribution to `Consumers`, regardless of whether Lucid is used in Online or Offline mode. Thus clients, such as the `Looker`, are oblivious to differences in the operating mode of Lucid. The absence of dependency on the source of data and events permits Lucid's users to employ a consistent interface and infrasturcure for data acquisition, as well as data analysis activities.

Note, the `IOMReader` does not adhere strictly to the description of the *Proxy* pattern, as detailed in Appendix C.2: a dynamic subscription mechanism is unavailable. The fact that events may be conditionally triggered by other events complicates the addition of this feature. In any case, clients wishing to unsubscribe from an event must simply remove the corresponding trigger definition from the Experiment Description file, re-compile, and reload the new `IOMReader` application.

3.7.1 Application Structure

Structurally, an IOMReader is a template composed of the elements shown in Figure 3.8. This resource and concurrency view illustrates the server-side objects comprising the data acquisition framework and Proxy pattern collaborators.

In the Online-mode, all interactions between the IOM and workstation occur over four TCP/IP port connections governed by five threads, which are denoted by their bold outline in Figure 3.8. These threads are described below, in order of decreasing thread-priority:

1. *Logger* - blocks waiting for `LogMsg` object arrivals to the `LogQ` message queue. These messages may contain error or status conditions, and are retrieved from the queue and forwarded to the machine designated as the I/O Manager's log-server. These messages are routed to *gxlucid*, where they are displayed for the experimenter as well as persisted to file to aid future diagnostics.

2. *Command Reader* - listens on the `Command Port` for incoming instructions from the experimenter's *gxlucid* Online session. This thread places `Command` objects in the `CommandQ` message queue, and notifies the `Acquisition` thread of the new instructions by sending it an `evCmd` event. The retrieved commands include orders to start, stop, pause, or resume data acquisition activities, but may also include user-defined directives. Thus, `Command` objects determine the operating state of the IOMReader application.

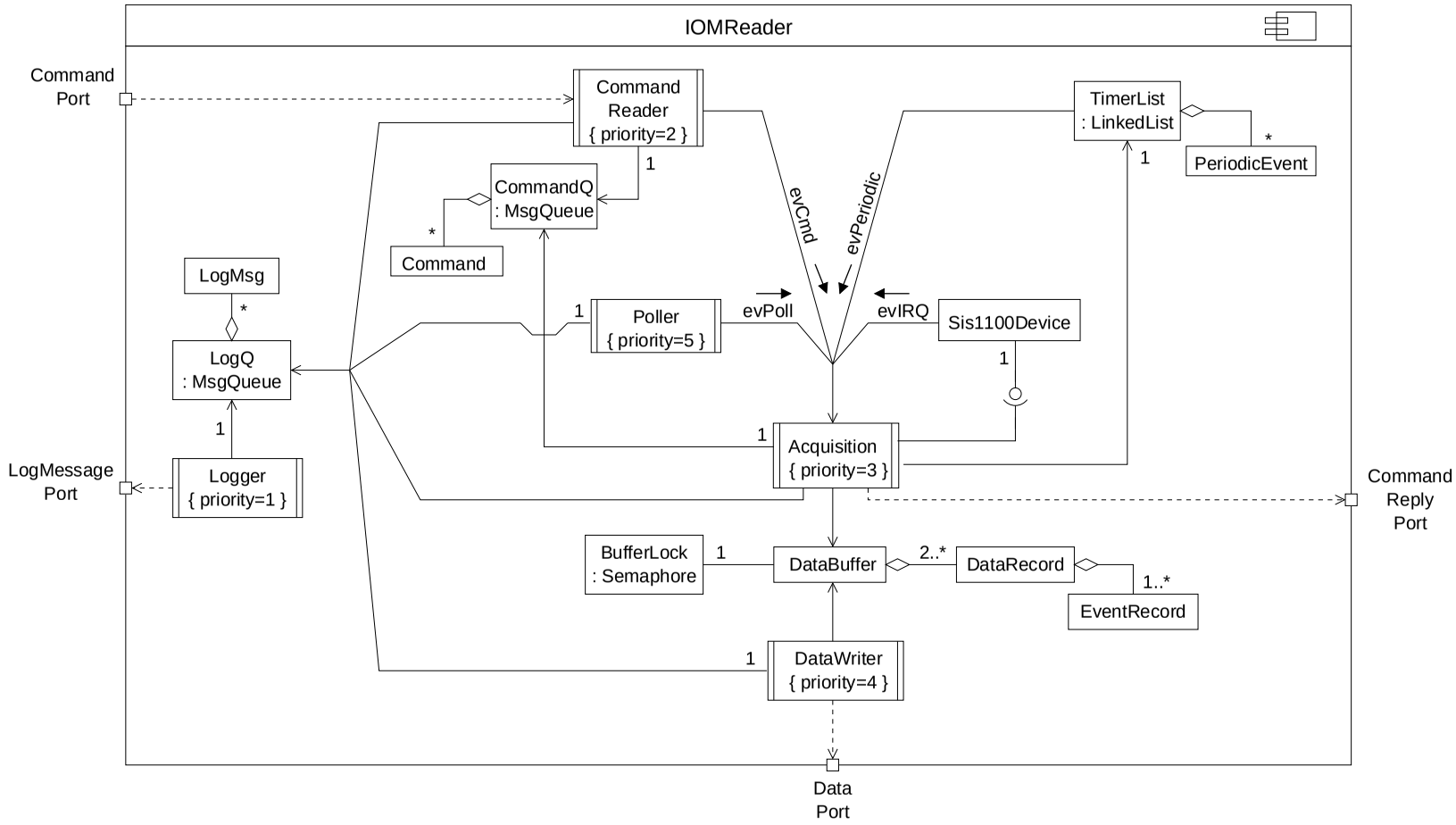

Figure 3.8: A concurrency and resource view of the `IOMReader` component. Priorities are indicated within each thread: numerically lower values indicate greater priority.

3. *Acquisition* - the engine driving data acquisition activity on the I/O Manager. This thread maintains an instance of the `Sis1100Device` for communicating with instrumentation modules, and provides the execution context required to service the operations specified in an Experiment Description File. Given its critical role, a detailed view of the `Acquistion` thread is presented below, in Section 3.7.2.

4. *DataWriter* - transmits a stream of `DataRecord` objects collected by the `Acquisition` thread, over the `Data Port` to a receptive `Reader` daemon on the workstation node. When engaged in acquiring data, the interactions between the `DataWriter` and `Acquisition` threads are the focus of activity on the IOM, and so are treated together, in depth, in Section 3.7.2.

5. *Poller* - the lowest priority thread in the `IOMReader` application. As frequently as it is able, the `Poller` sends `evPoll` events to the `Acquisition` thread. Thus, `Acquisition` is prompted to service those events in a continuous, *polling* manner. If an `IOMReader` application contains no polling-events, this thread destroys itself immediately after it has started.

Subscription to events published by the IOMReader is via the *trigger-definition* specified in an Experiment Description file. For example, the EDL statement, `trigger aEvent every 0.1 seconds`, registers a client's subscription with the `Acquisition` thread to execute the callback, `aEvent`, every tenth of a second. This mechanism permits clients to register callbacks upon the occurrence of any of the event-types published by the `Acquisition` thread.

3.7.2 The Acquisition and DataWriter Threads

Figure 3.9 illustrates the behavior of the finite state machine realized by the `Acquisition` thread. Although not indicated in that figure, it is important to note that these states operate concurrently with the state-set imposed by the RTEMS thread-model (refer to Figure 3.4). Thus, the states presented in Figure 3.9 are actually *composite* states: `Acquisition` may be in the states of Running-Blocked, Suspended-Blocked, Stopped-Executing, etc.

As implied by its placement at the center of the resource and concurrency view (Figure 3.8), the `Acquisition` thread is a key element of the `IOMReader` component. It is the receptacle of `Triggers` and the execution context for `Events` specified by an Experiment Description file. `Acquisition` is an event-driven FSM, responding to the following data acquisition-related event-types (see Figures 3.8 and 3.9):

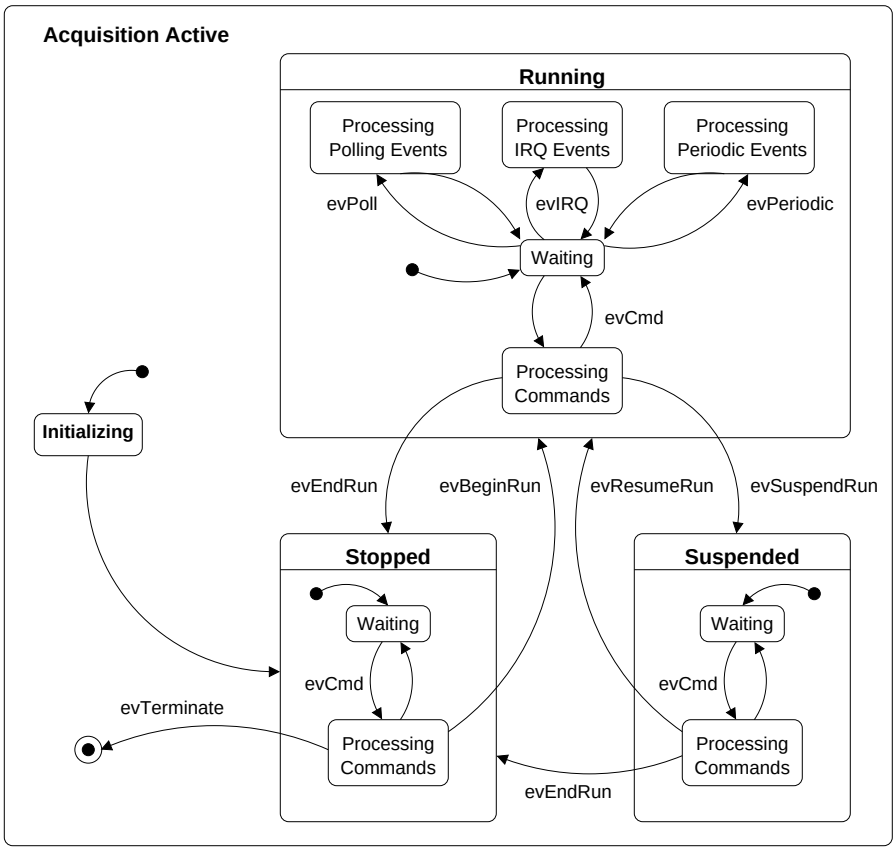

Figure 3.9: Diagram of the `Acquisition` thread's FSM structure and behavior.

1. *evIRQ* - triggered asynchronously from VME or CAMAC hardware interrupts, these events are delivered by interrupt service routines registered with the `Sis1100Device`.

2. *evPeriodic* - periodically generated timeouts, where client callbacks requiring periodic servicing are handled. These events may be triggered up to a maximum frequency of 100 Hz.

3. *evCmd* - command events originating when a user manipulates various interface widgets in the *gxlucid* UI, or triggered by other software events. These are the only event-types for which `Acquisition` will issue a success/failure response over the `Command Reply` port.

4. *evPoll* - polling events dispatched from the `Poller` thread. Because that thread runs at the lowest priority of the thread-set comprising an `IOMReader` application, polling events will be sent and handled only as quickly as permitted by the `Acquisition` thread's other obligations.

Additionally, there are four special `evCmd` event-types with which users may associate

actions to be committed. These are activated when different buttons of the *gxlucid* UI are depressed (see figure 1.6), and serve to transition the `Acquisition` FSM between the Stopped, Suspended, and Running states of Figure 3.9:

1. *evBeginRun* - the first event executed when the "play-button" of the UI is engaged. Also, enables instrumentation interrupt-generation, if required by the Reader-file.

2. *evEndRun* - executed when the "stop-button" is depressed. Disables interrupts and places `Acquisition` in the "stopped" state.

3. *evSuspendRun* - triggered when the "pause-button" is depressed. Disables interrupts and suspends data acquisition.

4. *evResumeRun* - triggered when the "pause-button" is dis-engaged. Re-enables instrumentation interrupts and omits all expired periodic events.

Once `Acquisition` has been transitioned into the `Running` state, the majority of the `IOMReader`'s activity is contained within interactions between the `Acquisition` and `DataWriter` threads. In light of this fact, it is worthwhile to closely examine the details of these threads' relationship.

Figure 3.10 is a snapshot of activities on the IOMReader during a typical Online session: the `Acquisition` thread is obtaining instrumentation data with VME block transfers, formatting and storing each event in the `DataBuffer`, and alerting the `DataWriter` to the event's presence. Within Figure 3.10, the **par** operator denotes *parallel*, or concurrent activities housed within its interaction fragment. In this case, the concurrent sequences are the execution contexts of the `Acquisition` and `DataWriter` threads. Within the same interaction fragment is the **loop** operator, indicating that sequences within will execute over some number of iterations, depending upon the evaluation of the loop's terminating condition. Here, the interaction fragment will repeat `while AcquisitionState == Running`.

Internal to the concurrent execution fragment are two additional, and closely related operators: **alt** (*alternative*) and **opt** *(optional)*. The *alternative* operator may be thought of as an `if...then...else` conditional apparatus: only one of the alternatives, delineated by horizontal, dashed-lines, may evaluate to true. Similarly, the *optional* operator is analogous to a simple `if...then` conditional statement: only if its guard statement evaluates to true will the sequence fragment execute.

In addition to its primary role of gathering data from instrumentation modules, the `Acquisition` thread also serves as an *event builder*. This activity involves assembling

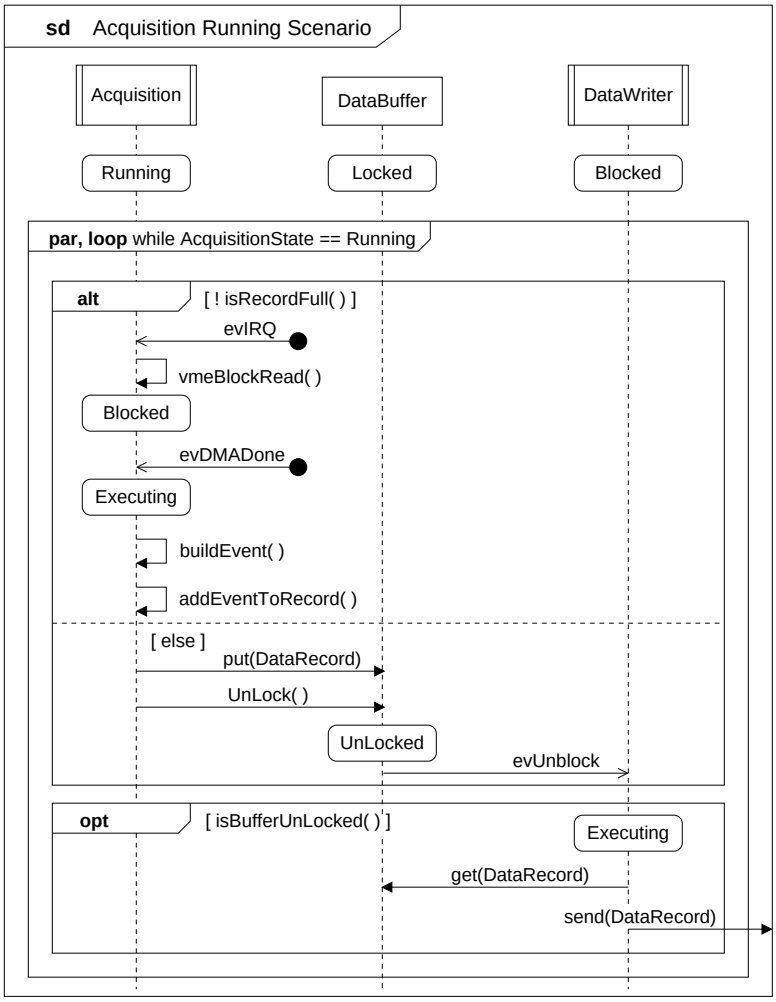

Figure 3.10: Sequence diagram of `Acquisition-DataWriter` thread interactions.

the data obtained from each event callback into `DataRecord` objects: each distinct event defined in a user's Reader-description file may potentially require the I/O Manager to save data acquired during the event's execution. To ease the duties of the `Acquisition` thread in this respect, each `Event` object (see Figure D.3) serves as its own container for data collected during its execution. `Events` are self-describing in the sense they identify, house, and describe their contents. All that remains is for the `Acquisition` thread to differentiate among instances of the same event-type. This is accomplished by associating with each Event a unique, monotonically increasing sequence number.

`Event` objects are themselves housed in super-structures known as `DataRecords`. These objects comprise the information stream sent from the I/O Manager to the Linux workstation, where they are distributed to data consumers. `DataRecords` are sourced from the `DataBuffer`, a circularly-structured resource pool of `DataRecord` objects. This buffer

is shared by both the Acquisition and DataWriter threads: the former adds objects to the buffer, while the latter removes them for transmission over the network.

Because the DataBuffer constitutes a shared resource, access to it is guarded by the BufferLock, a counting semaphore. This semaphore is used as synchronization element, coordinating the activities of each thread accessing the buffer. Although configured to have a maximum value of one, the semaphore is created with an initial value of zero, meaning the semaphore is initially "owned" by its creator, the Acquisition thread.

From Figure 3.10, the DataWriter is initially blocked, waiting to obtain the BufferLock. When the Acquisition thread has collected a full DataRecord of Events, it releases the semaphore, causing the RTEMS scheduler to place DataWriter on the ready-list of threads eligible for execution. When the DataWriter is granted use of the processor, it removes a DataRecord from the head of the circular buffer, transmits it over the network, marks that record as being again available, and then attempts to acquire the BufferLock again. Since that semaphore is of the counting variety, attempts by the "owning" thread (i.e. DataWriter) to re-acquire it result in that thread blocking until another thread again releases the semaphore.

Within the scenario portrayed in Figure 3.10, the Acquisition and DataWriter threads are essentially the only two threads competing for the processor. Given that Acquisition has the greater priority attribute, the DataWriter will only be allowed to execute when Acquisition blocks itself, waiting for the delivery of events. Thus, it becomes critical the Acquisition thread blocks at certain "scheduling points" in its execution context in order for the DataWriter to make progress in its own context. From Figure 3.10, Acquisition blocks itself when it issues a vmeBlockRead( ) directive, and must wait for the DMA transaction to complete. Therefore, the use of VME block transfers is beneficial not only in the sense of their economy of data movement, but also in the sense that they provide a critical "scheduling point", thereby affording the DataWriter thread an opportunity to execute its services.

3.8 Summary

This concludes Part I of the thesis. Having examined the hardware and software components of the Lucid data acquisition system, the discussion of Part II will focus on performance aspects of the system. Following an introduction to the concept of *dead time* and its effects, those ideas are then re-examined using the mathematical framework afforded by *queueing theory*. With this analytical tool, the schema utilized to obtain performance metrics for the Lucid DAQ is presented, followed by an analysis of those measurements.

CHAPTER 4
DEAD TIME

Most components of a data acquisition system have a minimum amount of time in which two input signals may be resolved as separate events. If one input is being processed by a device and another input arrives before the minimum required temporal separation, the second event may be lost, or cause a pile-up effect within the device. This minimum required temporal separation of input signals is known as the *dead time* of a device. More precisely, dead time may be defined as a time interval, following a registered (or a detected) event, during which the counting system is insensitive to other input events [20]**.** The loss of, or the pile-up of events are dependent upon the timing characteristics of the device providing service to the input signal.

In addition to the implications of the loss of input events, dead time distorts the statistical distribution of the interval between input events, possibly invalidating the use of an assumed distribution and its moments in correction calculations, as applied to the physical property under measurement. These effects are important to account for in so-called *counting experiments*, where the flux of reaction particles must be precisely known. Experiments like the absolute cross-section measurements performed using the *Blowfish* neutron detector with the Lucid DAQ are examples of such counting experiments.

In this chapter, the effects of dead time will be examined using mathematical descriptions based on two, commonly used models of dead time behavior. Also, since dead times almost never occur in isolation in practice, the effect on measured count rates of a series arrangement of two elements will be discussed. Finally, two techniques commonly used for the measurement of dead time in detectors and fast, electronic circuit elements is presented.

4.1 Dead Time Models

Dead times are typically modelled as being of either of two varieties, according to the effect produced by events that follow one another in a period of time that is less than the resolving time of the system:

1. *Non-extendable*, or *non-paralyzable* - those in which events with an interarrival period of less than the dead time of the system, τ, are lost and have no other effect on the system. That is, the system is insensitive to the arrival of an event, less than τ seconds after the preceding event. For example, ADC devices used in nuclear physics measurements typically have a conversion period on the order of tens of microseconds, where any further input is ignored until the current conversion is completed. Thus, the ADC device is effectively "dead" for this conversion period.

2. *Extendable*, or *paralyzable* - those in which the arrival of a subsequent event during a dead time period extends the dead time of the system from the time of the second event's arrival by another period, τ. This causes a prolonged period during which the events subsequent to the first are not recognized. Discriminators operating in the "updating output" mode display this behavior. A device displaying paralyzable dead time behavior may find use as a timing mechanism in certain experimental configurations, extending an input-gate width, for example. However, in the context of counting experiments, such as measuring the absolute cross-section of a reaction, a device with paralyzable dead time characteristics can only be detrimental, and is something to be avoided.

Single-channel systems characterized by paralyzable and non-paralyzable dead time responses to random-rate input signals are illustrated in Figure 4.1.

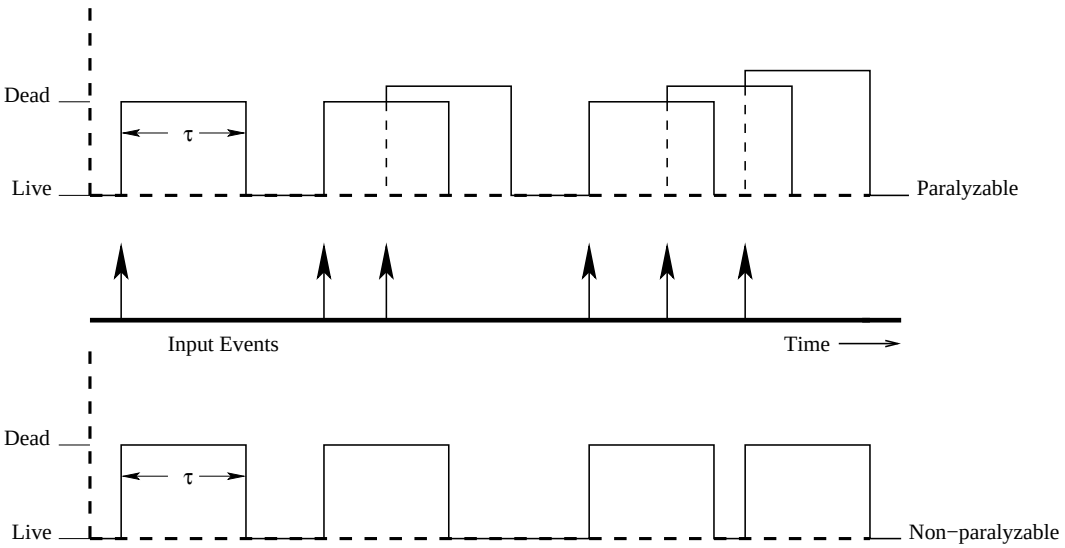

Figure 4.1: Illustration of *paralyzable* and *non-paralyzable* dead time behavioral models. Of the six input events, the non-paralyzable system resolves four, while the paralyzable system resolves three. Adapted from the figure given in [21].

4.2 Mathematical Description

The mathematical characteristics of non-paralyzable and paralyzable dead time systems are described in the following sub-sections.

4.2.1 Non-Paralyzable System

To quantify the effect of a non-paralyzable system, let the average input rate seen by a device be λ_{in}, and assume that the time interval between input events is exponentially distributed. Then, the system will lose a fraction of input events due to dead time, τ, producing an output rate of λ_{out}. The probability of lost events is the product $\lambda_{out}\tau$. Thus, $\lambda_{in} - \lambda_{out} = \lambda_{in}\lambda_{out}\tau$ is the rate of lost events, and the transfer function for such a system is:

$$\frac{\lambda_{out}}{\lambda_{in}} = \frac{1}{1+\lambda_{in}\tau} \tag{4.1}$$

4.2.2 Paralyzable System

By definition, the dead periods of a paralyzable system are not of fixed length, and so the output rate may not seem as straight forward to quantify. However, the key observation with a paralyzable system is that the output rate is equal to that portion of the input events with interarrival times that are greater than the dead time [22]. That is,

$$\lambda_{out} = \lambda_{in} \cdot P\{t > \tau\} \tag{4.2}$$

where $P\{t > \tau\}$ is the probability that the interarrival time, *t,* will be greater than the dead time, τ. If events arrive at random intervals, at an average rate of λ , the probability of *n* events arriving within a time interval, *t*, follows the Poisson distribution:

$$P\{n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{4.3}$$

The probability density function associated with processes described by such a Poisson distribution is given by:

$$p(t) = \lambda e^{-\lambda t} \tag{4.4}$$

where λ is the average rate of event arrivals (or, equivalently, the reciprocal of the average period between adjacent input events). Therefore $P\{t > \tau\}$ may be found by integrating

Equation 4.4 over the region $\tau \rightarrow \infty$:

$$P\{t > \tau\} = \int_{\tau}^{\infty} \lambda e^{-\lambda t} dt = e^{-\lambda\tau} \tag{4.5}$$

Substituting the above result into Equation 5.3 will yield the average output rate for a system with paralyzable dead time behavior:

$$\lambda_{out} = \lambda_{in} e^{-\lambda_{in}\tau} \tag{4.6}$$

The average output rate versus the average input rate is shown below in Figure 4.2 for both the paralyzable and non-paralyzable dead time models. For low input rates, the two models show almost identical behavior. However, at higher input rates, their behaviors differ considerably: the non-paralyzable model asymptotically approaches an output rate of τ^{-1}, whereas the paralyzable model shows a maximum output value of $(\tau e)^{-1}$ at input rates of τ^{-1}. In addition, the paralyzable model may yield the same output rate for two values of input, depending upon which side of the maximum the input rate falls on. Care must be exercised to ensure that the input rate measured corresponds to the input rate on the correct side of the maximum.

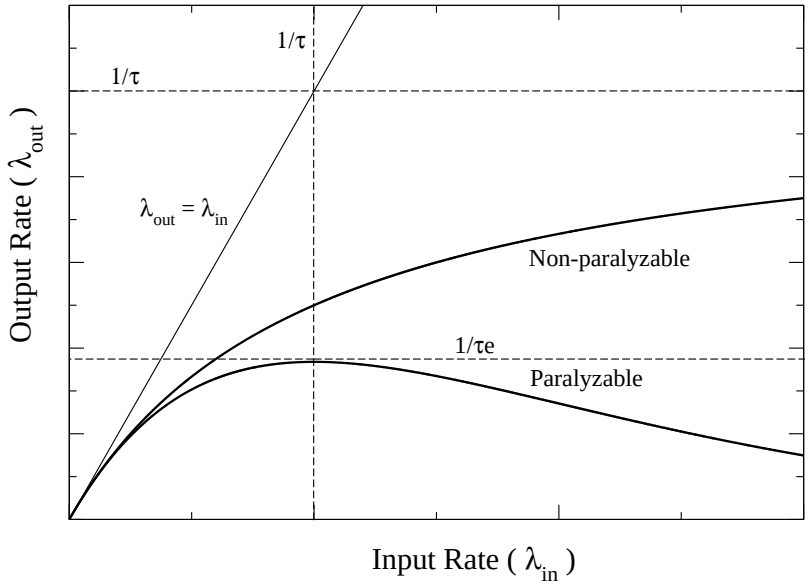

Figure 4.2: Output rate versus input rate for two systems with equally valued dead time, but different models of dead time . Adapted from [21].

4.3 The Effects of Dead Time

Dead time losses are contributed by all elements in a detector system, from the actual detector, to the electronics modules which process the the detector signals, to the computer system that extracts the data produced by "upstream" elements. Each contributing element must be analyzed to discover the extent of its effect. There is no general method available to calculate the combined dead time effects from several system elements, and each system presents different effects. Dead time affects three main areas of experimental data: output count rates, event interval densities, and hence, the event counting statistics. Each of these will be examined in turn.

4.3.1 Output Count Rates

The reduction of count rates, through the loss of input events, is perhaps the most significant effect of dead time [22]. Original count rates for each dead time model may be calculated from the equations given above, in Section 4.2. The uncertainties associated with the measurement of physical quantities scale as the inverse square root of the number of individual measurements, N: i.e. $\bar{\sigma} \propto \frac{1}{\sqrt{N}}$ [23]. Thus, in order to obtain a requisite degree of statistical uncertainty, an experimenter must make a sufficient number of measurements. Since losses of data due to dead time will occur in any counting experiment, the dead time of a data acquisition system may effectively dictate the duration of an experiment.

This idea may be clarified by an example. Consider the following scenario, illustrated in Figure 4.3: System A has dead time, τ_A, and System B has dead time, $\tau_B = N\tau_A$, where

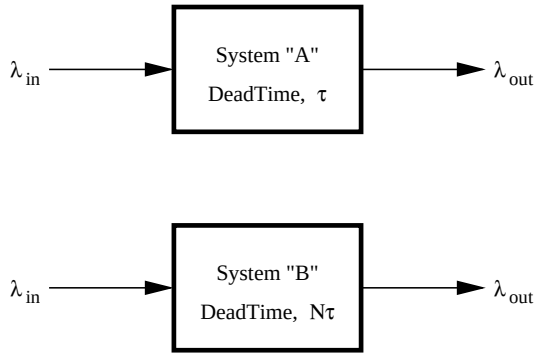

Figure 4.3: Block diagram of simple systems to illustrate the effects of dead time on output rates as a function of input rates.

N is an integer and $N \geq 1$. For simplicity, both dead times are of the non-extendable type

and are measured in seconds. Then, the ratio of output rates for the two, hypothetical systems will be :

$$\frac{\lambda_{out}^{A}}{\lambda_{out}^{B}} = \frac{\frac{\lambda_{in}}{1+\lambda_{in}\tau_A}}{\frac{\lambda_{in}}{1+N\lambda_{in}\tau_A}} = \frac{1+N\lambda_{in}\tau_A}{1+\lambda_{in}\tau_A} \tag{4.7}$$

The effect of this behavior is illustrated in Figure 4.4, for Systems A and B as a function of $\lambda_{in}\tau_A$, for several values of N, with τ_A equal to a unit dead time. The intuitive result of

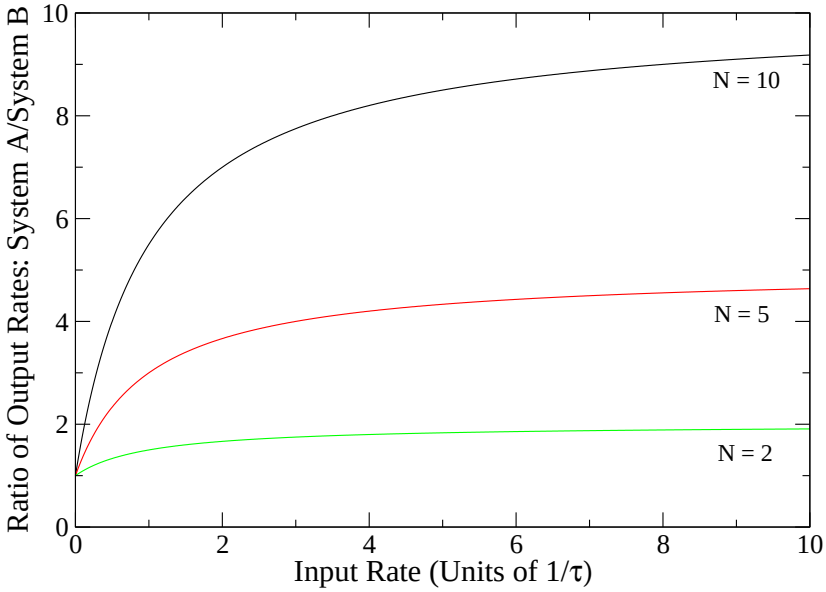

Figure 4.4: Plot of the ratio of output rates, from Equation 4.7 for Systems A and B.

this behavior is that System B will require more time than System A to obtain an equivalent number of events, in order to produce the same degree of statistical uncertainty in a measurement. The difference in time requirements asymptotically approaches the ratio of the dead times for the two systems. Therefore, because a greater number of measurements results in less statistical uncertainty, and given that facility beam-time is a precious commodity, it is most beneficial if a given experiment can be conducted in a minimum amount of time. For this reason, minimization of dead time for a given experiment is critical.

4.3.2 Interval Densities

Minimization of dead time alone is an incomplete solution, as dead time will affect other experiment considerations as well. The distribution of intervals between input pulses is one such factor influenced by dead time.

The probability density for the time interval between events is often assumed to follow that of a Poisson process, as given above in Equation 4.4. However, introduction of a dead

time will perturb the interval density distribution. In particular, a dead time will cause the density to vanish for interarrival times $t < \tau$, such that the interval density is truncated to allow only those events that are separated by at least the duration of the dead time [20].

The practical implication is that the interval density perturbation caused by the presence of dead time in the signal path will invalidate any assumptions made as to the Poisson-nature of the input signals. This may have consequences for calculations that assume a negative-exponential interval distribution for signal properties "downstream" of the dead time, such as mean or variance calculations used for event count-correction purposes. The error propagated to a calculated quantity affected by dead time is often as important (or even more so) as the magnitude of the dead time itself [22].

4.3.3 Counting Statistics

A system's dead time preferentially filters out events occurring within a time less than τ seconds after the preceding event. Thus, dead time has the effect of distorting the interval distribution of incoming events from an assumed exponential distribution. Clearly, this effect is a consequence of the dead time's perturbation to the interval density distribution of input signals.

The losses incurred from this filtering effect serve to modify the counting statistics to something other than the often assumed Poisson behavior. The degree of the distortion is insignificant for small losses (i.e. $\lambda_{in}\tau < 0.2$), but serve to reduce the variance expected in repeated measurements if the losses are not small [21]. That is, the Coefficient of Variation (variance/mean) is no longer unity, as it would be for the case of a Poisson process, but is in fact less than one.

4.4 Series Arrangements of Dead Times

Each channel of signal propagation in a data acquisition system typically features at least two or more elements arranged in series, each of which contributes dead time to the system. Clearly, it would be useful to be able to quantify the effects introduced by such a circuit, as illustrated in Figure 4.5. In particular, a description of the expected output rate as a function of the original input signal rate would be useful.

A tendency exists to consider only the effect of the dead time with the longest duration . However, this may be acceptable only if the longest dead time occurs first in the signal chain [20]. In a typical arrangement, the first element will have a shorter dead time, τ_1 than a subsequent element, τ_2 , in the signal propagation chain. Ordinarily, the dead time

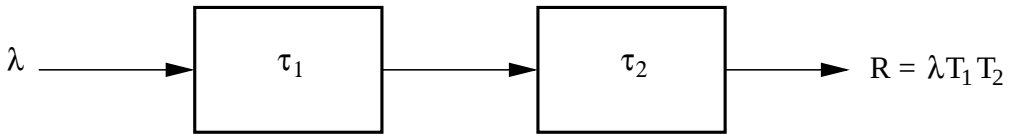

Figure 4.5: Series arrangement of two elements with dead times τ_1 and τ_2, where $\tau_2 > \tau_1$. The rate of input pulses is λ , and output pulses is R.

ratio, $\alpha = \tau_1/\tau_2$, is such that $0 < \alpha < 1$. In this scenario, the influence of τ_1 may be considered as a correction to be applied to the output count rate.

The relation between the input (λ) and output (R) count rates may be defined as the transfer function:

$$T = \frac{R}{\lambda} = T_1 T_2 \tag{4.8}$$

where T_2 is the *transmission factor* for τ_2 in the absence of the first dead time, while T_1 accounts for the perturbation due to τ_1. That is, $T(\tau_1, \tau_2)$ is a function of both dead times.

For circuit elements of non-extendable and extendable dead times, there are four possible arrangements, and T_2 is given by the transfer functions expressed by Equations 4.1 and 4.6, respectively, in Section 4.2. Although the case of two, non-extendable dead times in series presents the most complex expression of the four possible cases, it is the only scenario of relevance to the DAQ discussed in this thesis. Defining $\rho = \lambda\tau_2$, then the system's transmission factor is given by [22]:

$$T = (1+\rho)\left[\sum_{k=0}^{N} x_k\right]^{-1} \tag{4.9}$$

where $N = \lfloor 1/\alpha \rfloor$ [1], and x_k is the expression:

$$\begin{aligned} x_k = \left(\tfrac{e^{-s_k}}{k!}\right)\left\{[1+(1+\alpha)\rho - s_k]\, s_k^k + e^{\alpha\rho} s_{k+1}^{k+1}\right\} \\ +(k+1)(1+\alpha\rho)\,[Q(k,s_{k+1}) - Q(k,s_k)] \end{aligned} \tag{4.10}$$

with $s_k = \max\{(1-k\alpha)\rho, 0)\}$, and $Q(n,m) = \sum_{j=0}^{n} \frac{m^j}{j!} e^{-m}$. Plots of the numerical solution of Equation 4.9, expressing $R = 1 - T$ as a function of α may be found in [22].

4.5 Measurement of Dead Time

The need to apply accurate corrections to formulae influenced by dead time effects requires precise knowledge of the type and numerical value of the dead time involved. A

[1] The symbol, $\lfloor x \rfloor$, denotes the *floor* operator: the largest integer less than or equal to x.

traditional approach has been the *two-source method*, but this technique requires radioactive sources with specific properties (detectable radiation type and sufficient decay rate) and is only capable of 5-10% precision [22]. A more contemporary technique, utilizing dual oscillators (pulsers) of differing frequency, is both less cumbersome and much more precise. This *two-pulser* technique is also discussed below.

4.5.1 Two-Source Method

In addition to the difficulties noted above, the two-source method requires knowing, *a priori*, the type of dead time involved and also has strong geometrical dependencies. In the formulae given below, a non-paralyzable dead time model is again assumed.

The basis of this technique is the idea that, given the impossibility of the knowing the *true count rate* of a source, the *observed count rate* may be used to infer the dead time of a circuit. Rearranging Equation 4.1 to obtain the true, input count rate, n, in relation to the observed count rate m and the assumed constant dead time, τ_D :

$$n + z = \frac{m}{1 - m\tau_D} \tag{4.11}$$

where z is the zero-source count rate (i.e. background signals). If the product, $m\tau_D \ll 1$, such as in the case of a device with a small dead time, then a series expansion of Equation 4.11 yields (to first order):

$$n + z \approx m\left(1 + m\tau_D\right) \tag{4.12}$$

When the circuit is exposed to two sources simultaneously, it is assumed to obey the relation, $n_{12} = n_1 + n_2 + z$. Therefore, given the following system of equations

$$\begin{aligned} n_1 + z &= m_1\left(1 + m_1\tau\right) \\ n_2 + z &= m_2\left(1 + m_2\tau\right) \\ n_{12} = n_1 + n_2 + z &= m_{12}\left(1 + m_{12}\tau\right) \end{aligned} \tag{4.13}$$

the dead time of the circuit is found as [5]:

$$\tau = \frac{m_1 + m_2 - m_{12} - z}{m_{12}^2 - m_1^2 - m_2^2} \tag{4.14}$$

from direct measurement of the individual-source observed count rates, the combined-source count rate, and the background radiation rate.

However, as mentioned, this technique suffers from practical difficulties, such as source-detector positioning, inter-source scattering effects, and an analytic form unfavorable for

error propagation calculations [24].

4.5.2 Two-Pulser Method

In a modern variation on the two-source method, the input pulses supplied by two decaying sources are replaced with a superposition of pulses from two electronic oscillators (pulsers), of frequencies ν_1 and ν_2, with $\nu_1 > \nu_2$ (see Figure 4.6). The advantages to this technique of dead time measurement are simplicity and accuracy: dead time may be measured to a precision of better than 10^{-3} in only a few minutes using little more than two pulsers and three scaler channels [22]. In addition, variation of ν_1 permits identification of the *type* of dead time present for the system under test (i.e. extendable or non-extendable).

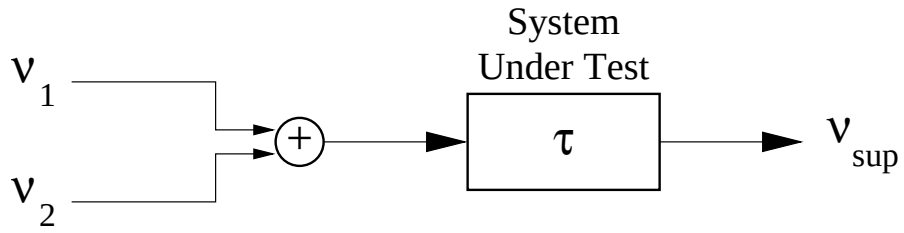

Figure 4.6: Illustration of system dead time measurement using the two-pulser method.

If ν_{sup} is the mean frequency of the superimposed pulses after having been filtered through a dead time, τ, then it may shown that for a non-extendable dead time:

$$\nu_{sup} = \begin{cases} \nu_1 + \nu_2 - 2\tau\nu_1\nu_2, & 0 < \tau < T/2 \\ \nu_1, & T/2 \leq \tau < T \end{cases} \tag{4.15}$$

Whereas for an extendable dead time:

$$\nu_{sup} = \begin{cases} \nu_1 + \nu_2 - 2\tau\nu_1\nu_2, & 0 < \tau < T \\ 0, & \tau > T \end{cases} \tag{4.16}$$

Where T is the period given by the reciprocal frequency of the faster oscillator: i.e. $T = 1/\nu_1$.

Referring to Figure 4.7, the region $\nu_1 < (2\tau)^{-1}$ yields a value for the dead time that is independent of type. For the region $\nu_1 > (2\tau)^{-1}$, the observed count rate, ν_{sup} gives a simple, graphical means of determining the type of dead time [22].

Denoting the number of registered pulses in the sampling period, t, as n_1, n_2, and n_{sup},

the dead time is then simply:

$$\tau = \frac{t}{2}\left(\frac{n_1 + n_2 - n_{sup}}{n_1 n_2}\right) \tag{4.17}$$

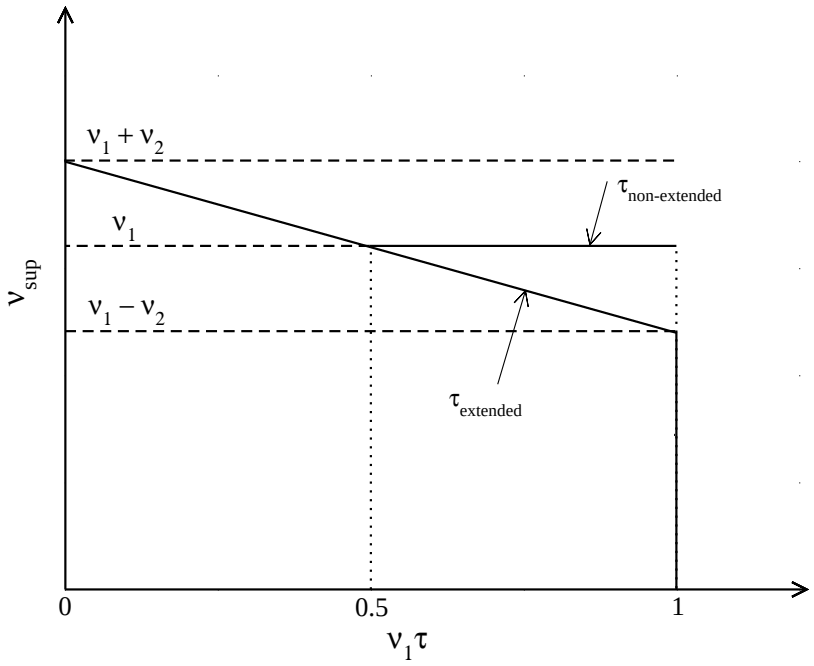

Figure 4.7: Experimental count rate, ν_{sup} , as a function of ν_1 , for the *two-pulser* method of dead time measurement. Both ν_2 and τ are constant. Figure adapted from [20].

4.6 Concluding Remarks

The material discussed in this chapter was presented from a nuclear physics perspective, and consequently, the subject of data acquisition system performance was delivered in such a way as to highlight those areas of immediate concern to the experimenter: i.e. those areas that most directly affect accurate determination of the physical property under measurement. Elements of system design, such as practicality and scalability have not been addressed. Also, much of what has been presented was done with the idea of application to the study of radioactive material decay-rate measurements. In particular, the dead time measurement techniques discussed may be especially suited for this application. However, much of the results presented remain applicable to counting experiments in general.

The characterization of a system's dead time as paralyzable or non-paralyzable, and the uncertainties associated with dead time corrections are of primary concern to experimentalists . Therefore, careful measurement of a system's dead time is required to accurately describe the uncertainties used in the experimenter's rate-correction calculations. The characterization and measurement of dead time for the Lucid data acquisition system are the subjects of Chapter 6.

In much of the discussion of this chapter, the assumption of a constant dead time has been implicit. While perhaps valid for the decision-logic modules present in the initial stages of a data acquisition system, this assumption cannot realistically be expected to hold for those components that perform higher-order processing of the system's data stream: i.e. the computer processor(s). The temporal variability of dead time must be accounted for to provide an accurate view of a system's capabilities. In the next chapter, a complementary and more general approach will be taken toward the subjects of data loss and system performance analysis. This approach will allow for time-varying input intervals *and* dead times.

CHAPTER 5

INTRODUCTORY QUEUEING THEORY

The branch of applied probability theory known as queueing, congestion, or traffic theory, provides a framework with which to analyze systems that are subject to demands for service whose frequency and duration may only be described in probabilistic terms. Queueing theory yields a mathematical description of such systems where the times at which requests for service and the duration of the service cannot be predicted, except in a statistical sense [25]. These systems are driven by demands for service and the duration of that service, both of which are generally described by probability distributions: i.e. they are random variables.

Data acquisition systems are typically composed of several sub-systems, each of which provides a service, or function, according to the demands made on its input terminal(s). The system discussed in this thesis is composed of both hardware and software sub-systems: detectors (signal generators), signal conditioning electronics, analog-to-digital conversion modules, and general-purpose computers executing various software processes in support of the entire system. Each of these elements responds to input stimuli by providing a "service", which persists for some finite period. Complexity in the analysis of such systems arises from variability in the service and input interarrival times, the problems of resource sharing and resource allocation, and from the geometry of the interconnections, providing data flow between each of the sub-systems [26].

Given the above complexities, computer systems, and data acquisition systems in particular, are excellent candidates for mathematical modelling using the tools afforded by queueing theory. A mathematical description permits exploration of the relationships between the demands for service placed on a system and the delays, or losses, experienced by the users of that system [27]. In addition, a mathematical approach permits calculation of several quantities of interest, such as the fraction of lost input events, system utilization and throughput, and may provide insight into areas of the system where resource contention is strongest, with the goal of optimizing system performance.

5.1 Queueing Notation

Like any area of specialization, queueing theory carries with it some terminology and definitions that are unique to the field. These will be introduced in the following discussion.

Computer systems may be modelled as "service centers", providing some function, or functions, in response to the arrival of requests for service, or events. In order to describe such a system, several characteristics of that system must be specified [28]:

1. *Arrival Process*: the periods between the arrival of input events (customers), or interarrival times, are generally assumed to be independent and identically distributed (IID) random variables. The so-called Poisson arrival process is the most commonly assumed. This assumption will be used frequently in the remaining discussion unless explicitly stated otherwise. A Poisson process implies that the interarrival times are IID, and obey a negative exponential distribution.

2. *Service Time Distribution*: the distribution of time intervals required by the server to fully process the demands of a request for service, as initiated by the arrival of a customer to the system (input event). Again, the most commonly assumed distribution is the negative exponential distribution. The prolific use of this distribution may be more attributable to its analytical ease of use, than its precise depiction of service times, particularly where computer systems are concerned.

3. *Number of Servers*: if a service center has multiple identical servers to handle the requests of customers, then those servers may be considered part of a single queueing system, with the input load distributed amongst them in some arbitrary fashion.

4. *System Capacity*: this characteristic specifies the maximum number of customers that may be present in the queueing system. The system capacity includes the number of customers waiting for service (enqueued), as well as those already receiving service. As an example, a single-server queueing system with finite queue capacity, B, would possess a maximum system capacity of $B+1$. All real systems have finite queue capacity, but the assumption of infinite queue capacity often results in mathematical simplifications, and the approximation may be justified when the system capacity is very large.

5. *Population Size*: if, in its lifetime, the queueing system supports serving only a finite number of customers, then this is indicated by the population size. Again, system analysis may be simplified by assuming that the customer population is infinite in size. Indeed, this is often the case for real systems.

6. *Service Discipline*: the algorithm that determines in what order the customers receive service. The most common discipline is First Come First Served (FCFS), and this will be the discipline assumed in the remainder of the discussion, unless otherwise noted. Other common service disciplines are Round Robin (RR) and Last Come First Served (LCFS). This parameter also includes the disposition of customers who arrive at a queueing center and find the center already occupied; typically a customer may either enter a queue to wait for service, or immediately depart from the system. These dispostions are known as Blocked Customer Delayed (BCD) and Blocked Customer Cleared (BCC), respectively.

Queueing theorists have developed a shorthand notation for specifying the above six parameters, known as Kendall notation. With this method, the queueing system's six parameters are specified in the form *A/S/m/B/K/SD*, where each entry corresponds to one of the above, six characteristics. Here, *A* is the interarrival distribution, *S* is the service time distribution, *m* is the number of servers, *B* is the number of buffers (system capacity, queueing capacity), *K* is the population size, and *SD* is the queue's service discipline.

The statistical distributions for interarrival and service times are conventionally denoted by single-letter symbols. Three of the most commonly used distributions and their symbols are shown below, followed by a simple description:

M	Exponential
D	Deterministic
G	General

An exponential distribution denotes one characterized by a negative exponential; i.e. a Poisson process. Deterministic distributions denote constant times with zero variance, in the statistical sense. A general distribution is one that may not be characterized by a (relatively) simple mathematical expression. However, a general distribution must have a finite mean and variance. Queueing results that are valid for a general distribution are valid for all distributions.

Conventionally, only the first three or four parameters of the Kendall notation are specified, *A/S/m/B*. For example, a queueing system characterized by negative exponential interarrival and service time distributions, *m* identical servers, infinite system capacity, and a FCFS service discipline would be indicated as a *M/M/m* system. An identical system with a finite number of input buffers (queue positions), *B*, would similarly be expressed as an *M/M/m/B* system.

5.2 Stochastic Processes

The analyses of queueing systems depends on the use of random variables that are functions of time. Typically, these variables are used to describe discrete properties, such the number of jobs in system or enqueued, or to describe continuous properties, such as the job interarrival and service time distributions. Such random functions of time are known as stochastic processes [28].

Two types of stochastic processes have already been mentioned: those whose variables may assume any real value (continuous), and those that may assume only integer values (discrete). Other processes commonly used in queueing theory are:

1. *Markov Processes*: those whose evolutionary behavior depends only the present state of the system. That is, the future state of the system is independent of its past states; there is no requirement of the knowledge of how long the system has been in its current state. This is possible only if the time-evolution of the system posseses a "memoryless", negative-exponential distribution. Thus, the "M" used in Kendall notation.

2. *Birth-Death Processes*: also known as arrival-departure processes, these are discrete-state Markov processes whose state transitions are limited to immediately adjacent states. For example, a system in state n may transition to state $n-1$, or to state $n+1$.

3. *Poisson Process*: if the times between events are independent and identically distributed (IID) according to a negative exponential distribution, the number of events in the interval $(t, t+\Delta t)$ obey a Poisson distribution, of the form:

$$f(n) = P\{N = n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{5.1}$$

   This discrete expression yields the probability of n events occuring, given that the average rate of events is λ. Poisson processes are often used in queueing theory because they describe memoryless processes, such that the time of occurance of an event is independent of the time of occcurance of the preceeding event. Application of this principle to queueing system event interarrival times permits closed-form, analytic solutions to be found in many cases.

5.3 Erlang's Loss Equation

The rate at which physics events are processed by the Lucid DAQ governs the experiment's duration and precision. The reaction rate of a photo-nuclear physics experiment, such as those performed at the HIGS facility, is statistical in nature, since the process is governed partially by quantum-mechanical laws, and partially by characteristics of the accelerator's γ-ray beam. The beam-dependent factors include the pulse duration, frequency, and intensity.

The reaction of interest may occur very infrequently, relative to the rate of uninteresting background events, or noise. Thus, an acquisition system must attempt to collect as many input events as possible, and delineate which events are interesting from those which are merely noise. However, due to dead time considerations, some fraction of events will be lost forever, with no chance of being processed. This effect will be felt most acutely by the loss of the (relatively) rare "interesting" events. The end result will be an increase in the experiment's duration in order to obtain some requisite level of statistical precision. In more practical terms, the increased experimental time requirements translates directly into increased consumption of facility beam-time.

One of earliest applications of queueing theory was to the field of teletraffic engineering in the early part of the twentieth century [25]. A study was conducted to deduce the number of operators required at a call center, such that the probability of a customer's call being denied service due to all operators being occupied was below some acceptable level. This same logic may be applied to computer systems by substituting applications, or servers, for the role of the operators, and the inputs to the applications playing the role of the customers.

One of the results of the above study was the development of a mathematical expression that enables predictions to be made as to the proportion of calls (customers) lost as a function of the demands made on the system. This expression is known as *Erlang's loss formula*, after the researcher who conducted the teletraffic study. The method of Erlang's approach is sufficiently general that it has been applied to fields as seemingly diverse as urban resource planning and computer systems performance analysis. It will be applied here to the analysis of the Lucid data acquisition system in particular.

The following discussion illustrates several of the principles commonly encountered when analyzing systems using a queueing theoretic approach. In particular, the ideas of state transitions and statistical equilibrium will be examined. This approach follows that developed in [25].

Consider the system consisting of a fixed number of servers, m, illustrated by the state-

transition diagram, Figure 5.1. Let the system be in state E_j, when there are j customers receiving service in the system, where $j = 0, 1, 2, \ldots, m$. Also, let P_j be the proportion of time that the system spends in state E_j. Initially considering only "upward" state transitions, if demands for service arrive at an average rate, λ, the system will experience an average rate of transition from state $E_j \rightarrow E_{j+1}$ of λP_j, since the proportion of time spent in state E_j is P_j. Note, for the special case of $j = m$, the transition from $E_m \rightarrow E_{m+1}$ is impossible, as only m servers exist, hence the rate of this state transition is zero. This corresponds to the situation where a customer arrives to find the queueing system fully occupied and is denied service. In practical terms, the system has incurred dead time in this instance. To summarize, the rate of upward state transitions, $E_j \rightarrow E_{j+1}$, is λP_j for $j = 0, 1, \ldots, m-1$, and is zero for the $j = m$ case.

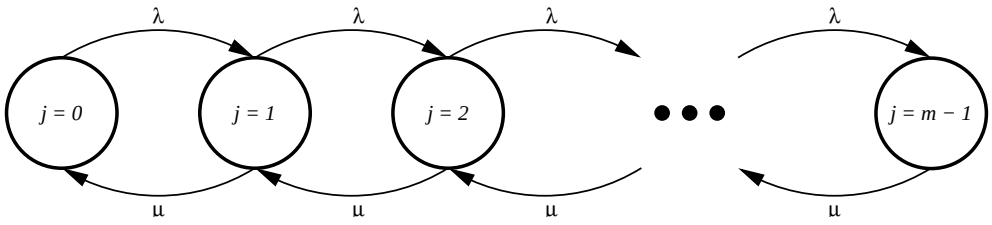

Figure 5.1: State transition diagram of an *M/M/m/m* queueing system. The system is entirely characterized by the number of customers present in it, j. The rate of transitions between adjacent states is denoted by the symbols on the inter-state arcs.

The average downward state transition rate, $E_{j+1} \rightarrow E_j$, may be found in a similar fashion by considering the rate at which an individual server completes service for one customer. Letting the average rate of service completions be μ, then for the $(j+1)^{th}$ state, E_{j+1}, where the system spends a proportion of its time equal to P_{j+1}, downward transitions, $E_{j+1} \rightarrow E_j$, will occur at a rate of $(j+1)\mu P_{j+1}$ per unit time (for $j = 0, 1, \ldots, m-1$).

Under steady-state conditions, the rate of upward state transitions may be equated to the rate of downward transitions, thus utilizing the familiar concept of conservation of flow. A stochastic system exhibiting such time-independence is said to be in a state of *statistical equilibrium*. For the $j = 0, 1, \ldots, m-1$ states of the system considered above, this equilibrium is expressed by the equation:

$$\lambda P_j \quad = \quad (j+1)\mu P_{j+1} \tag{5.2}$$

Solving this equation recursively will express the P_j-*th* state in terms of the initial state, P_0, where the system is idle:

$$P_j = \frac{(\lambda/\mu)^j}{j!} P_0 \tag{5.3}$$

Utilizing Equation 5.3, P_0 may be found by imposing the normalization condition that the set of proportions, $\{P_j\}$, must sum to unity:

$$P_0 + P_1 + \cdots + P_{m-1} = 1 \tag{5.4}$$

$$P_0[1 + \frac{(\lambda/\mu)^1}{1!} + \frac{(\lambda/\mu)^2}{2!} + \cdots + \frac{(\lambda/\mu)^{m-1}}{(m-1)!}] = 1 \tag{5.5}$$

$$P_0 = \frac{1}{\sum_{k=0}^{m} \frac{(\lambda/\mu)^k}{k!}} \tag{5.6}$$

Expressing the ratio of average input rate to average service rate, λ/μ, as ρ, the proportion of servers that are busy with j customers may now be expressed as

$$P_j = \frac{\frac{\rho^j}{j!}}{\sum_{k=0}^{m} \frac{\rho^k}{k!}} \tag{5.7}$$

The quantity, ρ, is known as the *offered load*, and constitutes a measure of the demand placed on a queueing system. Although ρ is dimensionless, it is assigned units known as *erlangs,* denoted as [erl]. Equation 5.7 is commonly known as Erlang's loss formula, or the Erlang B-formula and, for an *M/M/m/m* system, may be expressed as:

$$B(m,\rho) = \frac{\frac{\rho^m}{m!}}{\sum_{k=0}^{m} \frac{\rho^k}{k!}} \tag{5.8}$$

This expression is plotted in Figure 5.2, for several values of service channels, m.

It can been shown that Erlang's loss equation is valid for the case of Poisson-distributed input processes and for *all* distributions of service time, provided a finite mean and variance exist [27]. That is, Equation 5.8 is valid for all *M/G/m/m* queueing systems. In addition, the Erlang B-formula has no restrictions on the values that ρ may assume, whereas many queueing system models are subject to the constraint, $\rho < 1$.

The usefulness of Erlang's loss formula lies in its ability to predict the fractional loss of input events from a system service facility. Also, the relationship between input event interarrival times and facility service times is simply and concisely accounted for in the single parameter, ρ, the offered load to the system. The two components of the offered load are often accessible to measurement, thus providing verification of an analytic, "paper model", while enabling identification of system limitations and therefore, areas where improvements may be made.

For the case of a single server ($m = 1$), Equation 5.7 is evaluated below to discover the

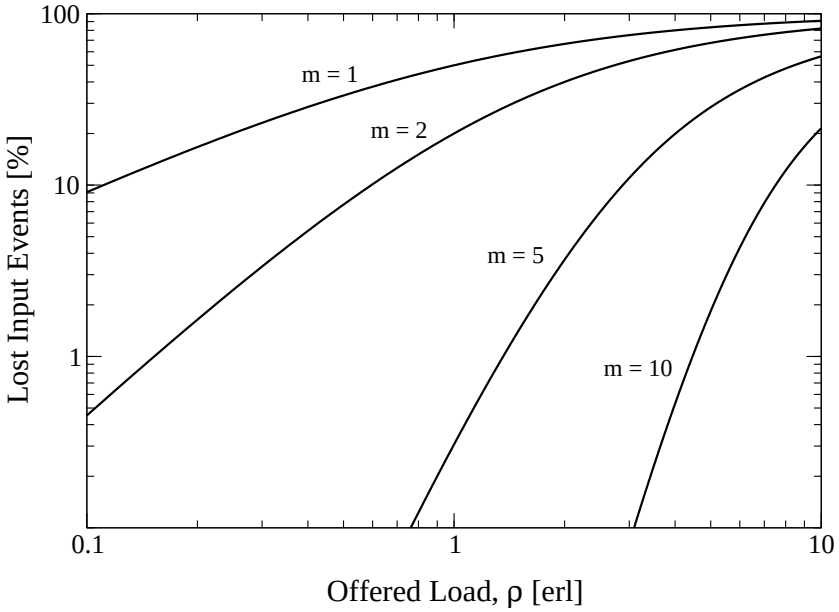

Figure 5.2: Log-Log plot of Erlang's B-formula for m = 1,2,5, and 10 service facilities.

fraction of time during which the system is idle, the probability of state P_0. This probability may be equivalently interpreted as the fraction of input events receiving service or, as the proportion of input events *not* lost. Interestingly, the expression for P_0 is identical to that found in Section 4.2, Equation 4.1, for the ratio of average output to input event rates:

$$P_0 = \frac{1}{1+\rho} = \frac{1}{1+\frac{\lambda}{\mu}} \tag{5.9}$$

5.4 The M/M/m/B Queueing Model

Having examined the *M/G/m/m* queueing model above, another common analytical model and its characteristics will be briefly presented here. This is the *M/M/m/B* queueing system. This model is really just the general case of the *M/M/m/m* model, with a finite queue capacity, B, where $B > m$, for m, the number of servers in the system.

In terms of digital electronics components, a queue may be physically realized as a memory module, or buffer, where digital information is stored for later access. An analog electronic buffer may be realized in the form of a capacitive circuit, or a long transmission cable. However, the storage lifetime of an analog buffer is generally far shorter in duration than that of a digital information buffer. This characteristic typically limits the usefulness of a transmission line as an analog signal buffer. However, capacitive circuit buffers are used in early-stage trigger decision modules in at least one particle physics DAQ [29].

Within the Lucid data acquisition system, buffering is available in the Dual-Port memory of the the VME ADC and TDC modules, as well as in the RAM of the I/O Manager and Workstation computers.

From the state-transition diagram of Figure 5.3, the steady-state probabilities of having n "customers" present in the system is:

$$P_n = \begin{cases} \frac{\rho^n}{n!} P_0 & n = 1,2,3,\ldots,m-1 \\ \frac{\rho^m}{m! m^{n-m}} P_0 & n = m, m+1, m+2 \ldots, B \end{cases} \tag{5.10}$$

where the probability of an idle system, P_o, may again be obtained by utilizing the normalization condition, $\sum P_n = 1$. The results of this calculation yield:

$$P_0 = \left(\sum_{n=0}^{m-1} \frac{\rho^n}{n!} + \frac{\rho^m}{m!} \left(\frac{1-(\rho/m)^{B-m+1}}{1-(\rho/m)} \right) \right)^{-1} \tag{5.11}$$

Note that for finite capacity systems, there is no requirement that $\rho < 1$; the system is

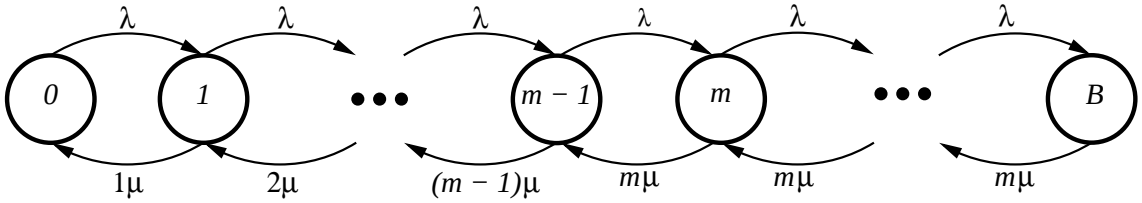

Figure 5.3: State-transition diagram for the *M/M/m/B* queueing system.

stable for all finite values of offered load.

Again, for the single-server case, where $m = 1$, and a finite buffer of capacity, B elements, the probability of finding the system in the n^{th}-state is given by:

$$P_n = \begin{cases} \frac{\rho^n(1-\rho)}{1+\rho^{B+1}} & n = 0,1,2,\ldots,B \\ 0 & otherwise \end{cases} \tag{5.12}$$

5.5 The *G/G/m* Queueing Model

Few useful results exist for queueing systems of the *G/G/m* model. The lack of independence between service and interarrival times leads to distributions that are no longer "memoryless", and the system is no longer a function of a single parameter (number of jobs in system). The most useful results are asymptotic bounds on $\overline{W_q}$, the average time a job spends waiting in the queue before service commences. This result may then be used

with *Little's Law*, which is valid in general, to provide bounds on average queue length, sojourn time, etc [30].

5.6 Queueing Networks & Tandem Queues

Data acquisition systems are composed of numerous, interdependent hardware and software subsystems. These systems may feature parallel and/or serial inter-connections to support the flow of information. In general, from a queueing theoretic perspective, these queueing networks are difficult to analytically model. These complications are a result of having to combine the results and analytic techniques applicable to individual components and derive conclusions which accurately describe the entire system under study. Given the countless combinations of queueing system components, characteristics, and interconnections, there are few general results from queueing theory that are applicable exclusively to networks of queues. However, some complexity may be alleviated if the systems possess analytically tractable features, such as Poisson input event interarrival and service times and infinite queue capacity [27].

Perhaps the most important simplification that may used in the analysis of networks of queues applies to those that may be modeled as being composed of $M/M/m/\infty$ systems:

> If the arrival process to a $M/M/m/\infty$ queueing system is Poisson with parameter λ, then under steady-state conditions ($\lambda < m\mu$), the departure process from the queueing system is also Poisson with parameter λ [27].

This property is known as the *equivalence property* for $M/M/m/\infty$ queueing systems. The implication of this result to a network consisting of N inter-connected, $M/M/m/\infty$ queueing systems is that each of the N components may be analyzed independently, with the aggregate behavior given by the product of the results from each individual component.

If any of the network's queues possess only finite capacity, the equivalence property no longer holds, and the above analytical simplification is not possible. However, the queueing network may lend itself to an analytical solution via examination of the system's state-transition diagram. This approach is subject to the constraints that each queuing system in the network possess negative exponential service time distributions and the arrival process to the system be Poisson in nature. The technique is as follows [27]:

1. Construct a state-transition diagram illustrating all possible states and steady-state transitions for the entire network.

2. Write and solve the conservation of flow equations in and out of each possible state for the entire network.

Proceeding in this manner, the steady-state probabilities associated with each state may be obtained. From this information, several quantities of interest may be calculated, such the average number of jobs in the system (queue lengths) and the fraction of potential users lost (i.e. the dead time).

As an example, consider the two-station tandem queueing facility illustrated in Figure 5.4. The input to the first station forms a Poisson process, with average rate λ . For simplicity, both stations feature service-time distributions in the form of a negative exponential function, with the identical average rate of μ . Neither station has a system capacity greater than unity.

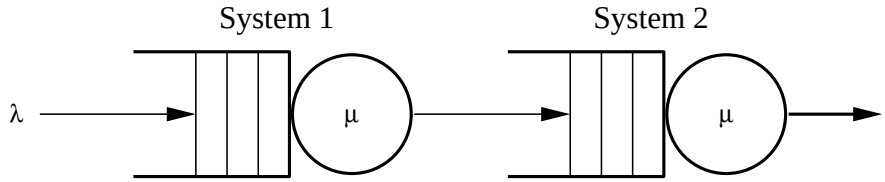

Figure 5.4: Schematic of two-station tandem queueing system, with jobs arriving at the average rate of λ , and identical average service rates, μ .

Service is provided sequentially in this system; new arrivals must visit both stations. Jobs completing service at the first station will proceed to the second station *only if the second station is not already occupied*. Therefore, the presence of a job at the second station will *block* the first station from providing service to a customer newly arrived to the system until the service at station two has been completed.

Unlike the one-dimensional queueing systems examined previously in Sections 5.3 and 5.4, this system requires two variables to describe the number of jobs (customers) in it:

- 00 - the “Idle” state
- 10 - the first station is busy.
- 01 - the second station is busy.
- 11 - both stations are busy.
- b1 - station one is idle, but *blocked* from accepting a new job because of the job in service at station two.

The following equations may be obtained by direct examination of Figure 5.5, and represent the "conservation of flow" for each possible state:

$$\begin{aligned} \lambda P_{00} &= \mu P_{01} \\ \lambda P_{00} + \mu P_{11} &= \mu P_{10} \\ \mu(P_{10} + P_{b1}) &= (\mu + \lambda) P_{01} \\ \lambda P_{01} &= 2\mu P_{11} \\ \mu P_{11} &= \mu P_{b1} \end{aligned} \tag{5.13}$$

where λ and μ are as previously described, and the state probabilities are given by P_{ij} . This set of five equations in five unknowns may be simultaneously solved by utilizing any four of the equations in conjuntion with the normalization condition for the state probabilities, given below:

$$\sum_{i,j} P_{ij} = P_{00} + P_{10} + P_{01} + P_{11} + P_{b1} = 1 \tag{5.14}$$

Proceeding in this way, the steady-state probabilities, P_{ij} may be found and used to com-

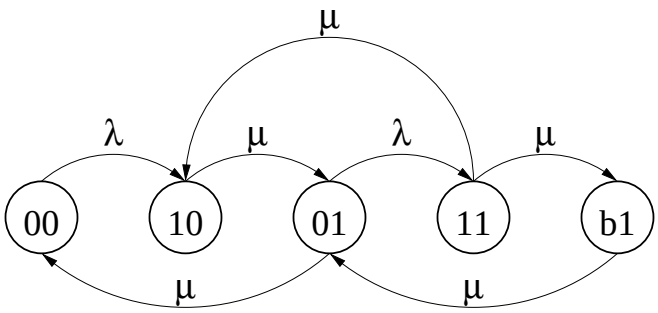

Figure 5.5: State-transition diagram for the two-element tandem queueing system with blocking.

pute properties of interest for this tandem system. One such important property is the portion of time that the tandem system is unable to accept new input events, or equivalently, the fraction of potential users lost to the system (i.e. the system's dead time). This portion is simply the sum of the probabilities corresponding to states where the system cannot accept new input [27]:

$$f = P_{10} + P_{11} + P_{b1} = \frac{3\rho^2 + 2\rho}{3\rho^2 + 4\rho + 2} \tag{5.15}$$

where the traffic intensity is defined in the usual form, $\rho = \lambda/\mu$. Finally, it should again be noted that such finite-capacity queueing systems are stable in the case of $\rho \geq 1$.

An analysis of the state-transition diagram representing the interaction of the two-component, Lucid sub-system consisting of the VME analog-to-digital conversion mod-

ules and the I/O Manager is demonstrated in Appendix E. This analysis will illustrate the dead time performance benefit that may be had by taking advantage of the dual-port memory and Chain Block Transfer capabilities of the VME modules.

5.7 Concluding Remarks

Due to the fact that the Lucid DAQ was developed to handle the data acquisition requirements of small-to-medium sized experiments (< 1000 channels), at least some of the simplifying assumptions mentioned above cannot be considered valid. However, as will be shown, some components of the I/O Manager may be approximated and analyzed using the state diagram technique described above. The next section will describe the Lucid data acquisition system using some of the ideas presented in this chapter, and discuss the equipment and procedures used to obtain performance measures.

CHAPTER 6

DAQ PERFORMANCE MEASUREMENTS

This chapter describes the techniques and equipment used to obtain performance metrics of the Lucid data acquisition system, including its dead time and those sub-intervals of which it is composed: interrupt and software scheduling delays, and data transfer durations.

From the perspective of an experimenter, a data acquisition system's dead time is perhaps its most important performance metric. A statistical analysis of dead time measurements provides the mean, variance, and minimum/maximum values of its probability distribution. In turn, these values may find use in the design of data collection algorithms and trigger configurations, or in calculations to correct counting-rates obtained during cross-section measurements.

Although methods of measuring the dead time contributions of detectors and their associated electronics modules have been described in Chapter 4, the focus of this chapter is on those systems succeeding the Trigger Logic subsystem in the signal-processing chain. In particular, the interactions between the digitization and I/O manager subsystems will be closely scrutinized, as these interactions are fundamental to to data collection operations. Given that detectors and trigger logic will almost certainly vary per experiment, the dead time contributions of the Trigger Logic subsystem will not be investigated further.

Beginning with a high-level description of the signal-processing chain, the algorithm governing the global behavior of the Lucid DAQ is utilized to develop a simple, two-state queueing model of the system. From this model, an expression is obtained to describe the average portion of time during which the DAQ is unable to accept new events for processing. This equation depends on both the average input rate and the average time required to process an event, thus suggesting the parameters requiring measurement. Following this treatment, those activities occupying the majority of DAQ resources are examined in detail. Finally, the chapter concludes with a description of the methods and resources utilized to measure time-critical system operations.

6.1 DAQ Trigger System

From Figure 6.1, a positive trigger level-1 (TL1) decision drives detector signal digitization. The TL1 *event signature* (see Section 2.1.1) is the logical AND of a detector signal satisfying amplitudinal threshold requirements in coincidence with a beam timing pulse. This TL1 signal will initiate a conversion sequence by the digitization subsystem. The sequence of events, from detector signal generation, to a positive TL1 decision, to the initiation of the digitization sequence, is on the order of tens of nanoseconds in duration. In contrast, the sequence of events following a positive TL1 decision forms the "high-level processing" component of a DAQ, with process durations that are typically orders of magnitude longer than those of first-level trigger processes.

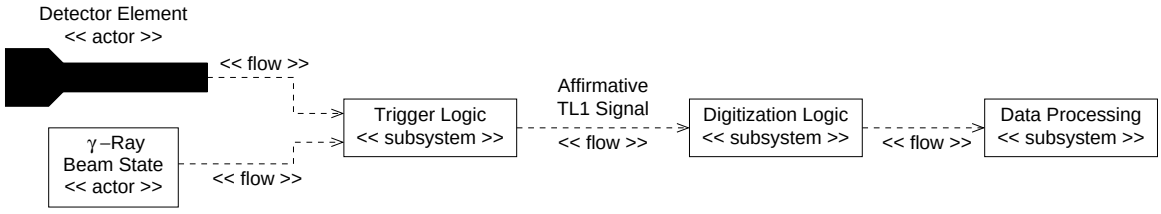

Figure 6.1: Schematic of data flow through a single channel of the Lucid DAQ.

The sequence of events comprising Lucid's reaction to a valid TL1 signal are further illustrated in Figure 6.2. The origin of this second stage of signal processing is marked by the initiation of an analog-to-digital conversion sequence.

The Digitization and Data Transfer periods, as well as the Application Response Latency of Figure 6.2 constitute a group of operations that are fundamental to the Lucid data acquisition system. The primary function of the DAQ may be comprised of various combinations of these key processes and, as such, the system spends the majority of its time executing these processes over and over in succession. Given that the dead time of the Trigger Logic subsystem is on the order of 100 nanoseconds, while the dead time of downstream components is on the order of tens of microseconds, the performance of Lucid is ultimately governed by the duration and variability of these critical operations, each of which are detailed below, in Section 6.2.

6.1.1 The INHIBIT Circuit

Currently, the data collection algorithm used with the Lucid/Blowfish system obtains data on an event-by-event basis. This means that a single event satisfying the TL1 requirements is processed and stored at the I/O manager before any other events may be accepted for

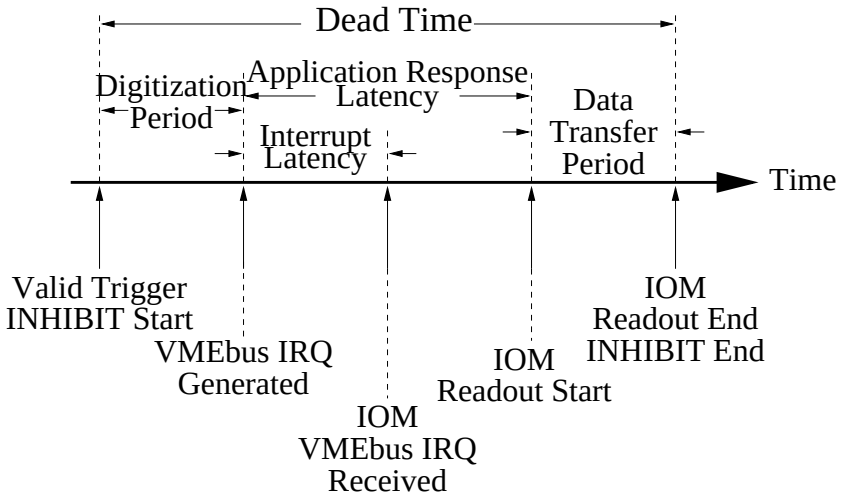

Figure 6.2: Timeline of events generating Lucid's dead time components. This scenario corresponds to the current, "event-by-event" acquisition algorithm in use. Note, periods are not to scale.

processing: buffering of data occurs no earlier in the signal processing chain than once the data arrive at the IOM.

In order to prevent the DAQ from responding to subsequent events that may perturb the processing of a current event, a so-called INHIBIT (INH) circuit, illustrated in Figure 6.3, is used. This circuit inserts a latch mechanism into the signal propagation path such that the circuit elements succeeding the latch are prohibited from responding to successive input signals until the latch is explicitly reset. In the case of the Lucid DAQ, the latch is set upon a valid first-level trigger (TL1) decision, and reset by a software-controlled pulse at the conclusion of event processing, as indicated in Figure 6.2.

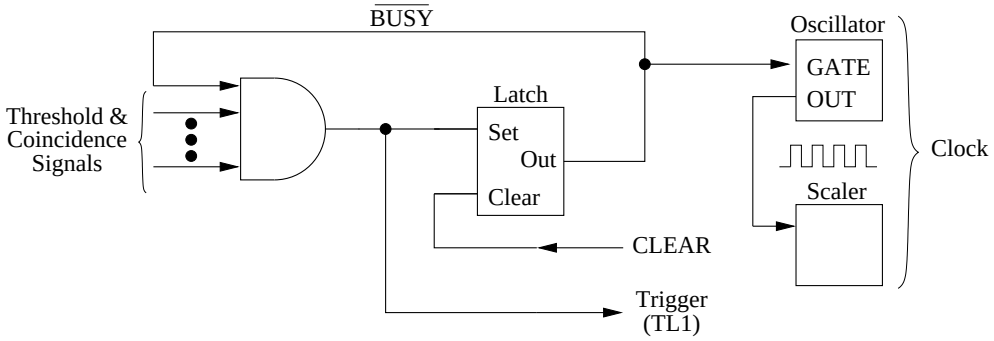

Figure 6.3: Typical first-level trigger logic, incorporating an INHIBIT circuit and dead time measurement facility. Figure adapted from [24].

There are at least two ancillary benefits to utilizing the INHIBIT circuit, as it has been described above:

1. Inclusion of an INHIBIT circuit forces the downstream elements of the DAQ to conform to the behavioral model of a system with non-paralyzable dead time. Thus, although this design imposes a dead time larger than that due to any constituent element, any uncertainty in the *type* of dead time has been eliminated.

2. The INHIBIT circuit (INH) may be used to provide a simple means of measuring dead time during online acquisition sessions [24]. As shown in the schematic of Figure 6.3, the inhibit signal can be utilized to gate a clock signal off and on. The ratio of the gated clock counts to that of ungated (i.e. free-running) clock counts yields the fraction of time during which the DAQ is "dead".

Finally, note the duration of the dead time for a system utilizing such an INHIBIT circuit is governed by characteristics of the system responsible for *resetting* the latch. In the case of Lucid, the duration of event-processing software operations *downstream* of the INH circuit determines the system's dead time, since the latch may be reset only at their conclusion.

6.1.2 Queueing Model

An INHIBIT circuit permits treatment of the DAQ as a simple, two-state system: the acquisition system may only be in either an "Idle", or "Busy" state (see Figure 6.4). Using the ideas of Section 5.3, queueing analysis may be used to model the DAQ as an Erlang Loss system, described using Kendall notation as an *M/G/1/1* queueing center.

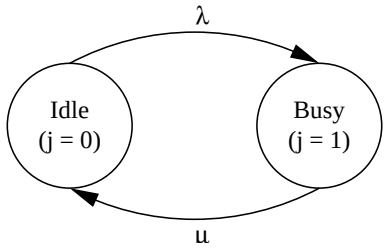

Figure 6.4: State-transition diagram of the *M/G/1/1* queueing model for the Lucid's I/O manager, with two states, $j = 0, 1$.

This model assumes that input events arrive at the system according to a negative exponential distribution, at an average rate of λ (i.e. the input process is Poisson). The average service rate, μ, equal to the reciprocal of the dead time, $\mu = 1/\tau$, is modelled as following a *general* distribution. A general distribution is realistically applicable in this case, as the dead time is effectively bound by the duration required by the IOM to process a single valid TL1 event in its entirety. Hence, the duration of this interval is dominated by software processes, which may not lend themselves to a simple, analytic form. The

service time distribution is dictated by the current use of the "event-by-event" processing algorithm. That is, the service times are an artifact of not buffering event data in devices preceding the IOM in the event-processing chain.

Utilizing results from queueing theory, that portion of time during which an Erlang Loss system is BUSY is expressed by the Erlang B-formula for a single server:

$$B(m=1,\rho) = \frac{\rho}{1+\rho} \tag{6.1}$$

where the traffic intensity, ρ , is given by the product of the average rate of event arrivals (TL1 signals), λ , and the average dead time, τ , such that $\rho = \lambda\tau$. It may be shown that, for the case of an Erlang Loss system, the portion of time that the system is busy is equivalent to that portion of events that are denied service and hence, lost from the system [31].

An unfortunate source of confusion among users of data acquisition systems stems from the usage of the term, "dead time". It is often quoted as a percentage, which is clearly a dimensionless quantity. When expressed as a percentage, "dead time" refers to that portion of time during which a DAQ is incapable of processing further events, and *not* to the period of time required by the DAQ to process a single event. That is, the former quantity is a "duty cycle", corresponding to the BUSY/IDLE cycle, while the latter quantity is an actual time period.

6.2 Dead Time Component Intervals

The following section describes those sub-intervals contributing to Lucid's dead time. These factors come into play only *after* an affirmative level-one trigger signal has been produced, as portrayed in the sequence diagram, Figure 6.2. It may be helpful to retain this definition of latency throughout the following discussion:

> *latency*: "the period of apparent inactivity between the time a stimulus is presented and the moment a response occurs" [32].

6.2.1 Digitization Period

This period is simply the duration required by a VME module to complete an analog-to-digital conversion sequence and store the data in an internal buffer. This information may be provided in the manufacturer's user-manual for a particular module. For example, in

the case of the CAEN v775 family of QDC and TDC modules presently used with the Lucid DAQ, the quoted value is 7 μs for the concurrent digitization of all 32 channels [33]. This duration was verified by observing, on an oscilloscope, the state of modules' BUSY signals, available via their front-panel CONTROL connectors. For all VME modules considered in this thesis, the digitization period is of constant duration.

6.2.2 Interrupt Latency

This is defined here as the period of time originating with the issuance of an interrupt-generating signal from a VME (or CAMAC) module, and terminating when the appropriate Interrupt Service Routine (ISR) is dispatched at the IOM by the RTEMS operating system.

Once a device on the VME bus asserts an interrupt request (IRQ), the VME System Controller module (the sis3100) sends a signal across the fiber-optic link to the I/O manager. This signal causes the CPU, upon completion of the current instruction, to suspend execution of the current thread, save that thread's context, and then invoke the software routine associated with the numeric interrupt *vector* it has received. Note, at this point the identity of the module generating the IRQ is unknown: the interrupt vector contains no additional information. It is the responsibility of software to perform such actions as necessary to obtain the identity of the VME module generating the IRQ. That process constitutes the VME interrupt acknowledge sequence (IACK) discussed in Section 3.6.7. For the timing measurements presented in this thesis, the IACK sequence is not considered a component of the interrupt latency, in order to comply with the conventional definition of the term as given above.

6.2.3 Context Switch Delay

Switching thread context is an operation fundamental to all multi-tasking computer operating systems. The delays incurred during a context switch consist of the period of time required by the operating system to determine and schedule the next thread to allot the processor to, record the state information of the presently executing thread, and restore the state information of the thread scheduled for execution [34]. The duration and distribution of this period is governed by characteristics of the operating system and the speed of the processor on which the OS is running. In particular, real-time operating systems are designed to have deterministic context switch and scheduling delays.

6.2.4 Application Response Latency

The results of schedulability testing performed on real-time operating systems almost invariably quote separate results for interrupt latencies and context switch delays. In reality, the response time of an OS to external stimuli is dependent upon the execution characteristics of both parameters *in succession*. While knowledge of each parameter's distribution and extent in isolation from one another may be valuable, it does not depict a realistic view of a system's real-time response characteristics.

Interrupt latency and context switch delays are linked by virtue of a commonly accepted practice for programming low-level ISR code: *the ISR must not issue any operating system directive (system call) that may block waiting on the availability of a resource.* This restriction exists because an ISR executes without context: state information is not maintained from one invocation to the next. Thus, interrupt service routines are quite limited in the actions they may perform. In fact, it is standard practice for a real-time application ISR to perform only the minimum operations required to service the interrupt and then defer further processing to a thread dedicated to such action. Typically, an ISR will simply disable interrupts at the device issuing the IRQ (Interrupt Request), then invoke a system directive causing the OS kernel to schedule a thread for execution to provide further processing. Thus, the inter-dependence of interrupt latency and context switching delay.

Minimizing the actions, and therefore the duration, of an interrupt service routine serves two purposes, as discussed in Section 3.6.3:

1. The system is able to respond more quickly to other, possibly higher-priority interrupts that may be generated and require service, and

2. The flow of execution is able to enter the context of a thread as quickly as possible, where there are no restrictions on the system directives that may be issued. Thus, the duration of the period when restrictions on system calls exists has been minimized.

One metric, the *RTCore Index,* is defined as "...the worst-case latency experienced by a periodic task on a heavily loaded system, including the interrupt latency, the overhead of the scheduling code, and the cost of a context switch..." [35]. This metric provides a measure of the latency originating with a periodic event and terminating upon entrance to the application thread which handles the event occurrence. Using the RTCore Index as a guideline, a more general metric is proposed here:

> The *Application Response Latency* is that measured on a heavily loaded system for the period originating with the occurrence of a hardware interrupt

request and terminating upon entrance to *application code* dedicated to the handling of that event.

This idea is applied to the I/O manager by designating the originating event to be the generation of an IRQ signal by a VMEbus module and the terminating event to be the entrance to an application routine which handles the generated IRQ. Defined in this way, the interrupt latency, scheduling and context switch delays, and the VME Interrupt Acknowledgment sequence (IACK) are all accounted for in the time-budget of the Application Response Latency.

6.2.5 Data Transfer Period

The direction and volume of data flow in an acquisition system is highly asymmetric, as the system's purpose would dictate. The movement of digitized data, from their place of origin, to a storage or manipulation location, constitutes a time-intensive process. This process becomes more expensive with an increase in the number of information channels, channel occupancy ratio, and digital resolution.

For the DAQ under study, digital information is transfered from the Instrumentation subsystem to the IOM, where it is buffered and assembled into Lucid's proprietary data format. The IOM then forwards the complete events over Ethernet media to the Workstation PC, where the data may then be graphically displayed, analyzed "online", or written to permanent storage for "offline" analyses at the user's convenience. In contrast, the quantity of data written *to* the VME and CAMAC modules is generally small relative to the volume that is transfered to the Workstation for storage.

Data transfer rates, from VME modules to the I/O manager, are dependent upon several factors. One of the most important dependencies, but one that is often overlooked, is the transfer rate capabilities of the VME device itself: modules of one type may simply transfer data faster than modules of another type. Other factors affecting the data transfer rate are the type of VME access mode (e.g. single-cycle or block transfer), the byte-width of the data access, and the volume of data produced, per module, per transfer. Clearly, these factors will influence any benchmark figures produced for VME data transfer rates, and so must be properly accounted for.

6.3 Test Apparatus and Algorithms

Two distinct series of measurements were made to obtain performance figures for the Lucid data acquisition system. The first set of tests were performed at the DFELL facility,

Duke University. The primary purpose of these tests was to measure the dead time characteristics of the acquisition system under realistic, but controlled settings. Secondary attributes of the system under study included the data rates from the VMEbus modules to the IOM and between the IOM and Workstation computers over Ethernet media.

A second set of measurements was performed at the University of Saskatchewan, focusing on the low-level components comprising the VMEbus-IOM interaction. This testing studied the dead time contribution from system attributes that are intrinsic to the timing requirements of VME-PC communication, and exist independent of any Lucid software process. In particular, the timing characteristics between a single VMEbus module and the VME-PCI driver software executing on the RTEMS-controlled IOM were studied. Performance metrics obtained include interrupt and context switch latencies.

The following sections will illustrate the reasoning, algorithms, and equipment used to carry out each of the measurements described above. Before providing a detailed description of both series of measurements, the timing mechanisms used throughout the testing will be discussed.

6.3.1 Timing Mechanisms

All measurements discussed over the next several sections require the determination of time intervals. These intervals encompass the events of interest and must be accurately determined to yield meaningful results. Therefore, accurate methods of measuring the event periods are required. Two mechanisms were used to provide measurements of the event intervals: one timing mechanism used a clock system external to the IOM, while the other made use of a device intrinsic to all Pentium and later-model Intel processors. Each of these devices is described in more detail below.

6.3.1.1 CAMAC Clock

All clocks are composed of two essential elements: a stable oscillator, and a counter to measure the number of pulses from the oscillator [36]. Using this idea, a simple clock was constructed from two CAMAC modules; a SAL clock module (oscillator) and a Kinetic Systems KS3615 24-bit scaler (counter) module.

The SAL clock module features four output channels, adjustable from 5 MHz to 150 Hz, and four output channels at a fixed rate of 1 kHz. In all measurements described below, the SAL clock was configured to emit pulses at 5 MHz, for maximum resolution. Each output channel is individually inhibitable, thereby providing a means to gate the clocks on and off. Feeding an oscillator output into a scaler channel forms a simple clock, which may

be activated without software intervention. This feature is necessary to measure intervals which are initiated by events beyond the control of software, such as the arrival of a GATE signal to the VME digitization modules.

6.3.1.2 Software Clock

In addition to the information provided by the CAMAC clock system, the duration of certain software events were monitored using the the Time Stamp Counter (TSC) register available on all Pentium and later model Intel processors [37]. The TSC is a 64-bit counter that increments at the CPU clock frequency and permits inexpensive access via the single assembly code mnemonic, *rdtsc*. This single instruction will fetch the 64-bit counter value and store the result in a pair of 32-bit, general-purpose, on-CPU registers, *edx* and *eax*, containing the upper and lower 4-bytes of the counter value, respectively.

Encapsulating software events in calls to read the TSC provides timestamps to delineate the start and finish of that event. The difference between adjacent timestamps is then directly proportional to the duration of that event, $\Delta t_n = \alpha_c(t_i - t_{i-1})$, with the constant of proportionality, α_c , being the conversion factor required to transform the number of TSC "ticks" into standard units of time or frequency.

Reading and saving the TSC value was accomplished via the following "C" language macro, utilizing inline-assembly code:

```
#define rdtscll(value)
     asm volatile("rdtsc" : "=A" (value))
```

This simple construct takes a user-supplied, unsigned 64-bit integer and returns the TSC value in it.

6.3.1.3 Clock Conversion Factors

Both of the timing mechanisms described above simply read the number of "ticks" accumulated by a counter. A reference clock is necessary to convert the number of "ticks" produced from one counter, say the TSC, into a time interval. For example, recording the number of TSC ticks after one second has passed will yield the conversion factor for that device, $\alpha_{TSC}\left[\frac{tichs}{second}\right]$. This factor is required to assign physical dimension to the measurement of an interval, since the number of clock "ticks" is the dimensionless quantity that is actually measured. The reference clock used here was the i8254 timer chip, a standard component of the modern PC architecture, typically used by the OS to generate periodic interrupts, thus forming the timing structure for the operating system [38]. These periodic

interrupts determine the time-resolution of the operating system: time-dependent operations may be performed with no finer granularity than that dictated by the periodicity of these interrupts.

The conversion factors for both devices were obtained by similar methods: a periodic thread was invoked on the RTEMS system to record the number of counts each "clock" contained after a period of one second, as determined by the "reference" clock (i8254 timer). This measurement was repeated over several thousand iterations for each clock system.

6.3.2 Dead Time Measurements

This series of measurements were conducted in early August of 2004, at the Duke Free Electron Laser Laboratory (DFELL), situated at Duke University in Durham, North Carolina. The goal was to evaluate the distribution and extent of dead time due to the most frequent operation of the IOM processor: responding to an interrupt request (IRQ) generated by a VME module. This event indicates that valid data may be available for extraction from the digitization modules. Referring to Figure 6.2, this period is denoted with the label, "Dead Time", beginning with the arrival of a GATE signal to the VME ADC modules, and ending with the conclusion of the extraction of the event's data from the modules. As indicated by the event time-line, the period is comprised of several sub-events. Measurement of these sub-intervals will be discussed in their appropriate sections. Next, the methods and the equipment used to measure Lucid's dead time will be discussed.

6.3.2.1 Apparatus

The IOM processor in use at the HIGS facility is a 2.4 GHz Pentium 4, with 256 MB of RAM, running the RTEMS operating system. This machine is coupled to the Workstation processor via a point-to-point, 100 Mbps Ethernet connection, and coupled to the VME crate via the fiber-optic link of the sis1100 VME-PCI interface. The Workstation processor is a 866 MHz Pentium III, with 256 MB of RAM. This computer executes the Lucid application under the control of a Linux operating system.

The quantity and type of VME modules utilized for these tests were the same as those that would be used during actual experiments. Module type and physical layout within the crate are shown in Figure 6.5. Three CAMAC modules were also utilized, but are not indicated in this diagram. These consisted of two scaler modules (i.e. counters) and one SAL clock module. As these CAMAC modules are accessed infrequently, their contribution to the system dead time is negligible.

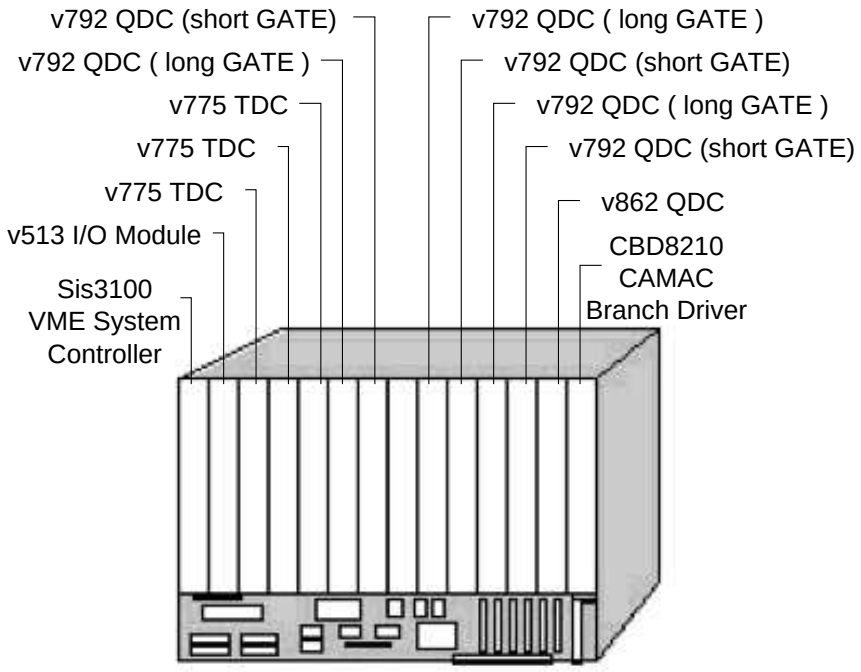

Figure 6.5: Module type and configuration within the VME crate, as used throughout performance testing of the Lucid DAQ, while at the DFELL facility.

The input signals to the system under test were produced by the Blowfish detector array in response to the ambient radiation signals present in the experiment vault at the DFELL. Variation in the average signal rate was controlled by manipulating the threshold voltage at the CFDs (constant fraction discriminators), thus permitting greater or fewer numbers of detector pulses to enter the electronics system. Average input signal rates were generated over the range from approximately 1 kHz to 25 kHz.

Utilizing a radiation source to generate the input signal was important for three reasons:

1. A signal distributed in time according to a negative exponential function contains the least amount of information, or conversely, the maximum entropy, and hence is "the most random" signal possible [30]. Thus, using an input signal of this type to drive the data acquisition system affords exploration of behavioral characteristics at the boundaries of its capabilities.

2. This produces a realistic perspective of system performance, given that this type of input signal is expected from accelerator-induced nuclear reactions, such as those produced by the HIGS facility.

3. Many of the analytic results from queueing theory are formulated based on the as-

sumption that the input process is a Poisson stream. Therefore, exposing the system under study to this form of input signal is crucial when comparing an analytical model of system behavior to experimental measurements. This is of particular importance given that the *M/G/1/1* queueing model developed in Section 6.1.2 requires a Poisson input process.

No experiment has ever successfully disproved the theory that single-channel, nuclear decay is a Poisson process [21]. Thus, there is also some sense of security that the input signal being used to drive the DAQ is of a known, quantifiable nature.

6.3.2.2 Methodology

Interrupts were generated on the falling edge (NIM logic sense) of a CFD output that was fed into the STROBE input of a CAEN v513 Input/Output module. One channel of this I/O module was configured to act as a software-driven output. In conjunction with the STROBE signal (STB), used to gate the SAL Clock ON, the output channel was used to gate the clock signal OFF. Refer to Figures 6.6 and 6.7 for hardware geometry and signal details. This hardware timing mechanism served to measure the sum of the intervals

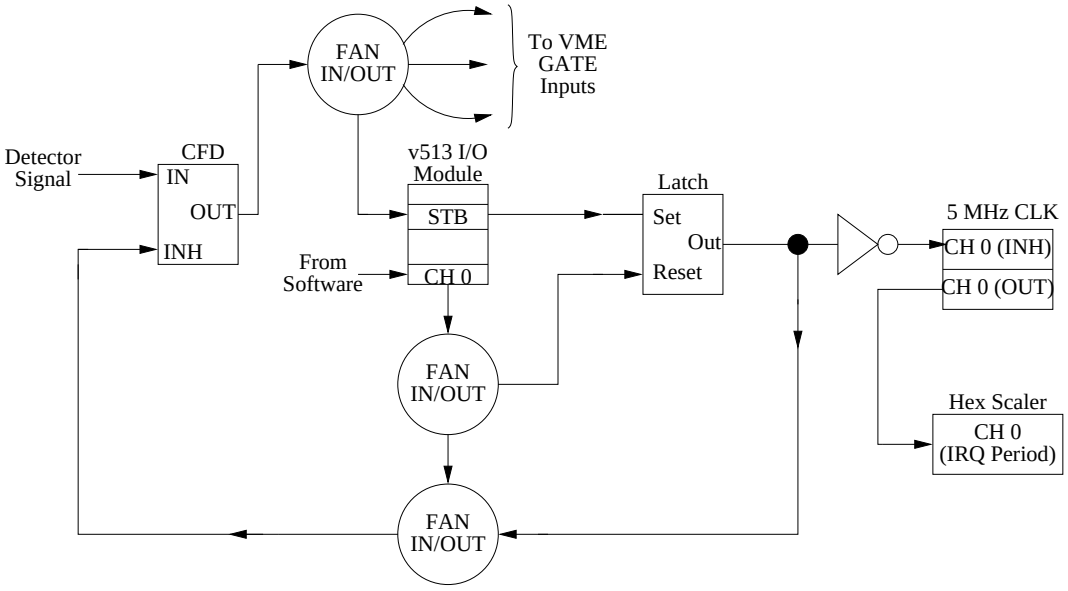

Figure 6.6: Schematic of hardware and control signals used at the DFELL facility to measure the dead time contributed by the "Digitization" and "Application Response Latency" portrayed in Figure 6.2.

denoted in Figure 6.2 as the Digitization Period and the Application Response Latency. The duration of the data transfer from the VME modules to the IOM was recorded using

the TSC mechanism discussed in Section 6.3.1.2. Figure 6.7 illustrates the details of signal timing, corresponding to the schematic of Figure 6.6.

Referring to Figure 6.7, the dead time was computed from the sum of two intervals:

$$\tau = \tau_1 + \tau_2 \tag{6.2}$$

where, τ_1 is the "Digitization plus Application Response Period" , and τ_2 is the "Data Transfer Period". τ_1 was measured using the hardware timer of Section 6.3.1.1, and τ_2 was measured using the IOM's Time Stamp Counter register. The dead time was measured for 100,000 input events, or for a duration of one minute, whichever resulted in the maximum number of recorded events.

In order to minimize the impact of the load accrued by reading out the CAMAC scaler module containing the interrupt-timing information, the scaler was allowed to accumulate counts for 1000 "Data Transfer Period" events before being read. That is, the hardware-timed data was averaged over a period corresponding to 1000 interrupt events. Hence, second order and higher statistical moments were not available for this period. However, this information was obtained in later testing (see Section 6.3.3).

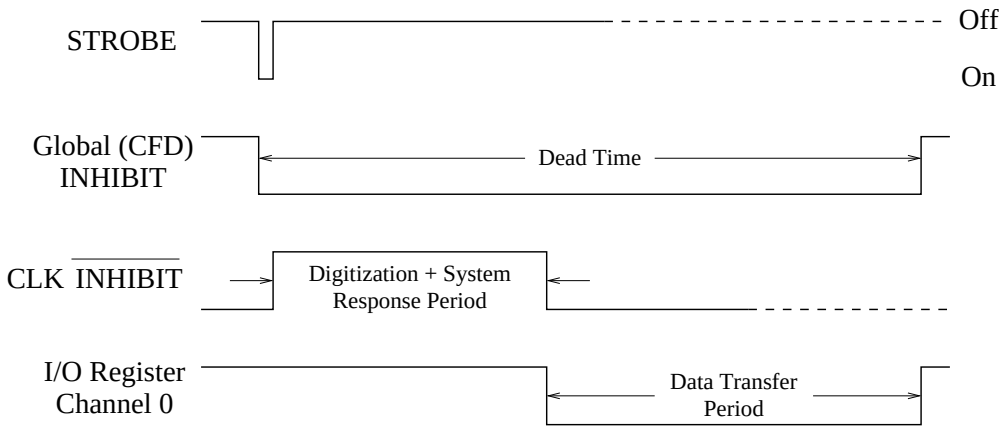

Figure 6.7: Signal timing diagram of events comprising the dead time measurements performed at the DFELL. Periods are not to scale.

Two additional scaler channels were used to measure the total number of CAMAC clock ticks and input trigger pulses. This data was sampled at 2 Hz, again in an effort to minimize instrumentation loading effects on the system under test.

Chain block transfers (A24/D32 CBLT) were utilized to transport data from the VME modules to the IOM. Module data occupancy was set at 3 or 32 four-byte data words per module, and was alternated between runs for a given input signal rate. Three data words per module was chosen as depictive of worst-case module occupancy conditions,

whereas thirty-two words per module corresponds to the maximum possible quantity of data generated per GATE signal. The latter quantity might only be generated during the relatively infrequent event of monitoring the VME digitization modules' DC-bias levels, which are also known as *pedestal* values.

It is advantageous to minimize the quantity of data written to disk during an online data acquisition session; less data written to the hard drive results in a less frequent requirement to transfer data to optical media for permanent storage. Due to the heavy resource requirements of the DVD writer hardware and software, it is not possible to record data to that medium "on the fly": i.e. in real-time, as the data is being acquired. Hence, data acquisition must be suspended while data previously recorded to hard disk is written to DVD. Assuming an online data acquisition rate of 2.5 MBps, enough data to fill 4.7 GB DVD-R may be obtained in approximately a half-hour. Given that the duration to record that quantity of data to DVD is of the same order as the time to acquire it, this process may become a considerable source of system dead time. This problem may be mitigated by the addition of a subsystem dedicated to the task of writing acquired data to optical media.

Past experiments have utilized data compression software (gzip) to achieve compression ratios of approximately 10:1. The perturbative effects due to the usage of compression software on the Lucid host were also investigated during this portion of system measurements. Due to Lucid's dependence upon the TCP/IP stack, any software process used by Lucid, including the Looker and Writer processes, may trigger TCP's rate-control mechanism. This control mechanism has the potential to throttle the flow of data from IOM to Workstation, which in turn may cause the IOM data buffer to saturate, as it is prevented from draining. This situation will cause suspension of data acquisition while the full buffer cannot drain, hence becoming an additional source of dead time.

6.3.3 Dead Time Component Measurements

These measurements were performed in the fall of 2004 at the U of S, with the purpose of determining of the extent and distribution of the periods comprising the "Dead Time" interval of Figure 6.2. Each of these periods will vary from one invocation to the next, with the exception of the digitization period of the VME modules, which is constant for the particular modules used. The source of variation in the periods is due to scheduling "jitter" present in the software/hardware interaction, as the RTEMS kernel allocates execution time on its CPU to the various competing software entities in response to external events.

6.3.3.1 Apparatus

The equipment utilized here consisted of identical Pentium III machines in the roles of IOM and Workstation processors, each featuring 256 MB of RAM and a CPU clock frequency of 450 MHz, running RTEMS and Linux operating systems, respectively. The system under test was the interaction between an interrupt-generating CAEN v513 VME I/O module and the IOM processor, whereas the Linux machine served the role of an I/O traffic-generator. See Figure 6.8.

6.3.3.2 Methodology

The concept and techniques for this aspect of testing were taken from ideas presented in [34]. The underlying concept is to measure the interrupt latency, context switch delay, and the overall Application Response Latency experienced by a high-priority thread on a real-time operating system while that system is subjected to heavy loading in the form of competition for resources incited by lower-priority threads. Worst-case latencies may thus be obtained, providing a measure of the determinism present in the system under test. By definition, a hard real-time system must provide guaranteed bounds on latencies incurred when a high-priority task is scheduled from an ISR, *regardless of the amount of low-priority load*. Although the Lucid data acquisition system does not strictly conform to the description of a hard real-time system, knowledge of worst-case latencies is still valuable, as these define the limitations of the system's capabilities.

Given the practical impossibility of achieving "worst-case" conditions, due to the large number of possible system states in a computer system, a probabilistic approach was taken: by repeatedly measuring latencies on a heavily loaded system, the maximum recorded delay was assumed to correspond to the poorest response of the system under test. It should be noted that the performance of the system under test is determined by the combined effects of a *system* of components and hence, should not be interpreted as indicative of the capabilities any single component in isolation.

Together with the sis1100 driver's interrupt service routine, the main body of the test software consisted of three threads, *isrTask*, *triggerTask*, and *chargenEcho* task, listed in order of decreasing priority. These threads are illustrated in the interaction fragment, *Latency Measurement Scenario*, of Figure 6.9. The RTEMS networking stack and Ethernet driver occupied additional threads, and were identically assigned the lowest priority in the system.

The software initialization procedure consisted of configuring the CAEN v513 I/O module to generate interrupts whenever its input-register was written to within *trigger-*

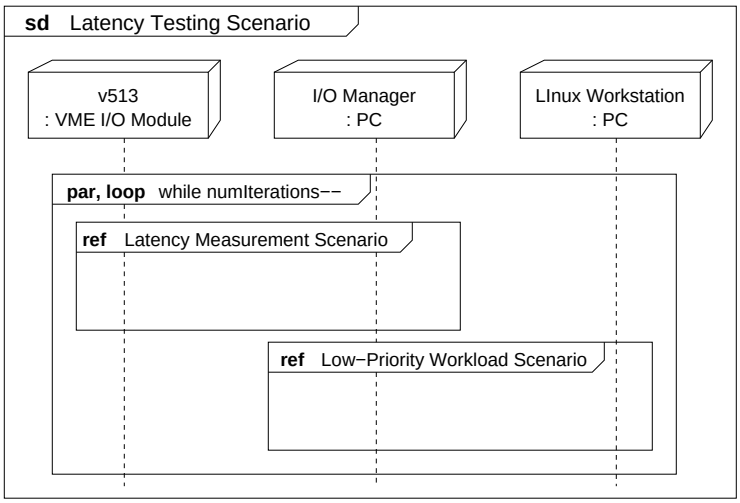

Figure 6.8: Overview of latency measurements.

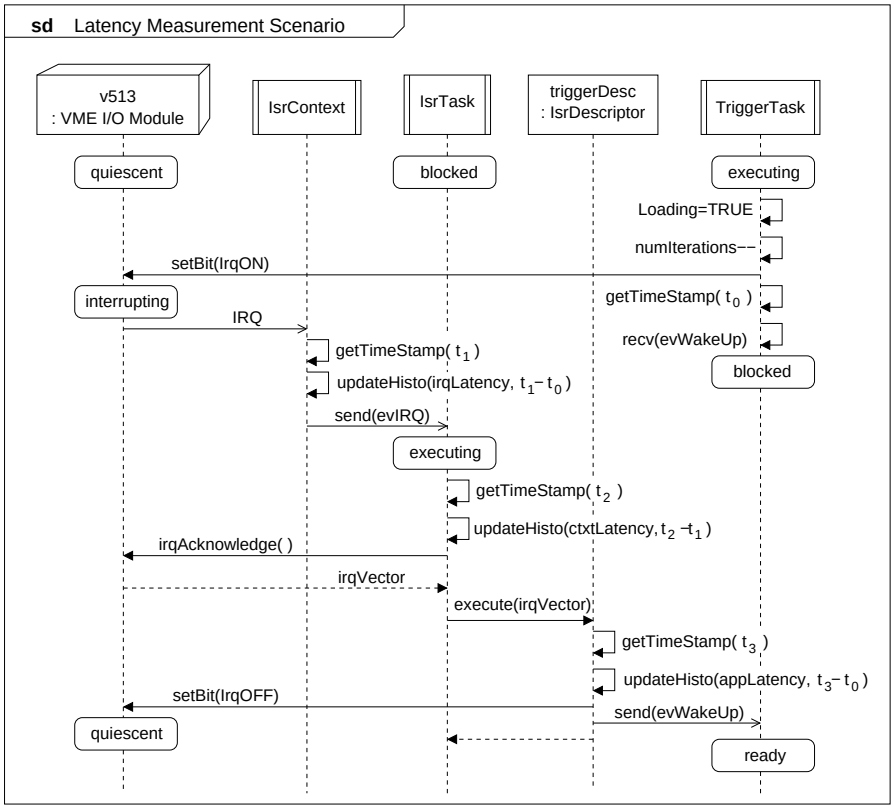

Figure 6.9: Sequence of events during latency measurements.

Task, registering the application's interrupt response routine with *isrTask* (as described in Section 3.6.7), connecting the *chargenEcho* thread to the Linux server, and readying a port for serial communication over the RS-232 medium.

A timing diagram illustrating the above software events and thread interactions is shown in Figure 6.10. Referring to this figure, histograms of latencies were produced for the following timestamp differences:

$$\Delta t_1 = t_1 - t_0 \quad \text{, Interrupt Latency}$$
$$\Delta t_2 = t_2 - t_1 \quad \text{, Context Switch Delay}$$
$$\Delta t_3 = t_3 - t_0 \quad \text{, Application Response Latency}$$

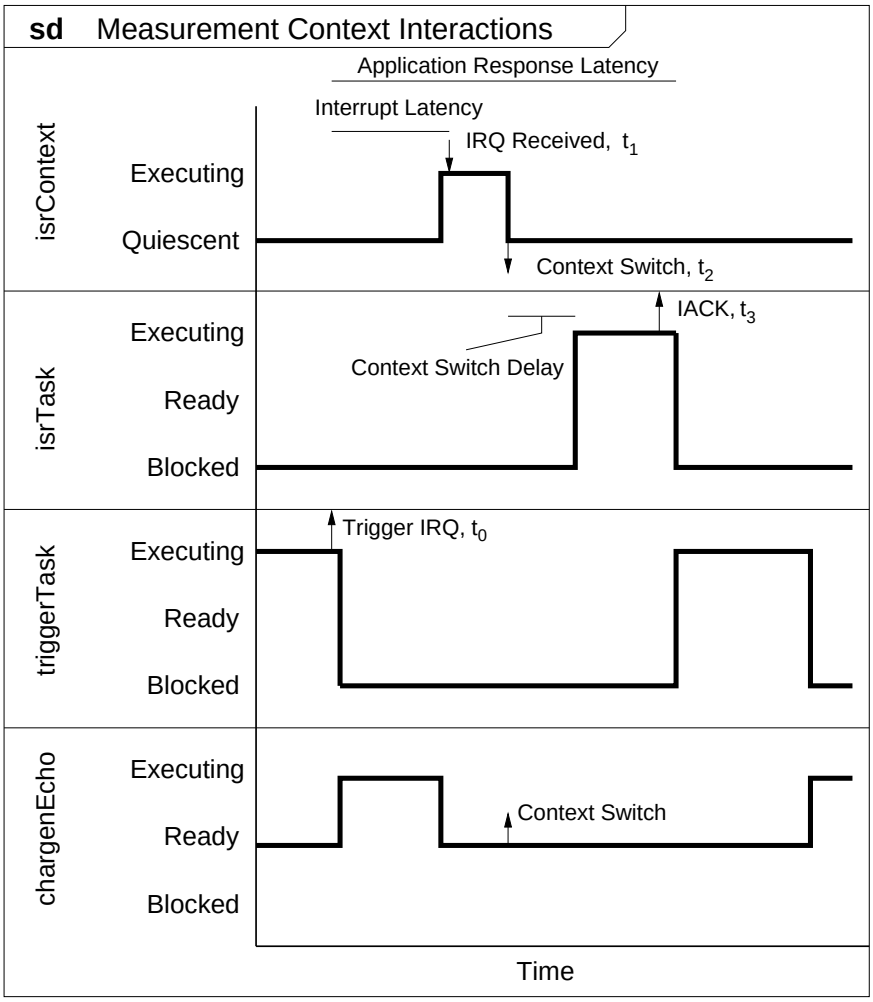

Figure 6.10: Timing diagram of thread interactions and state-transitions during the dead time component measurements. Time stamps were obtained at the points indicated in the figure as, $t_{0\rightarrow 3}$, using the TSC register of the IOM .

Once the *triggerTask* instructs the VME I/O module to generate an IRQ, it obtains timestamp, t_0 , and blocks waiting for a software event from the callback registered with *isrTask*. When the interrupt service routine is invoked by RTEMS, the Time Stamp Register is read again, yielding t_1 . The difference, $\Delta t_1 = t_1 - t_0$, proportional to the interrupt latency, is stored in a histogram structure. The interrupt service routine then unblocks *isrTask*, which, being the highest-priority thread, is scheduled for execution when the ISR returns. Immediately upon entry to this thread, the TSC is read again, giving t_2 and the difference corresponding to the ISR duration plus context switch and scheduling delays, $\Delta t_2 = t_2 - t_1$. The duration of the ISR's execution is assumed negligible and is neglected in the remaining time-budget. The *isrTask* then carries out the VME interrupt acknowledge sequence (IACK) in order to obtain the identity of the interrupting module. Upon successful completion of the IACK sequence, the registered callback obtains the final timestamp of the test sequence, t_3 , providing the Application Response Latency, $\Delta t_3 = t_3 - t_0$. At this point, a software event is sent to *triggerTask*, unblocking it, and the entire test sequence begins again. Note, only when *triggerTask* and *isrTask* are blocked waiting may *chargenEcho* execute.

Loading effects were provided by the low-priority thread, *chargenEcho*, executing I/O operations on behalf of the IOM processor. These I/O operations consisted of reading data produced by a TCP/IP character-generator server running on the Linux host, and echoing that data back to the host over an RS-232 serial communication medium (see Figure 6.11). The character-generator server simply produces a stream of client data in the form a repeating pattern of seventy-two ASCII characters [39]. The client thread (*chargenEcho*) reads a random number of characters from the Ethernet link, between 1 and 72 bytes, and writes the data to the serial interface. Randomizing the volume of input data written and read introduces a degree of variability to the time structure of the I/O loading thread.

In addition to the load imposed by *chargenEcho*, the IOM was also subjected to "ping flooding" by the Linux host. This technique exercises the RTEMS networking stack by eliciting an ICMP ECHO_REPLY packet in response to an ECHO_REQUEST generated via the *ping* utility, a standard program on many Linux distributions [40]. Specifying, *ping -f hostname*, from the Linux command-line will issue ECHO-REQUEST packets as quickly as ECHO_REPLY packets are received, or at 100 Hz, whichever is greater. Statistics of the ICMP request/reply session are reported upon program (i.e. *ping*) termination.

At the conclusion of a specified number of iterations (several million), the test data was transferred to the Linux machine via TFTP for analysis and plotting.

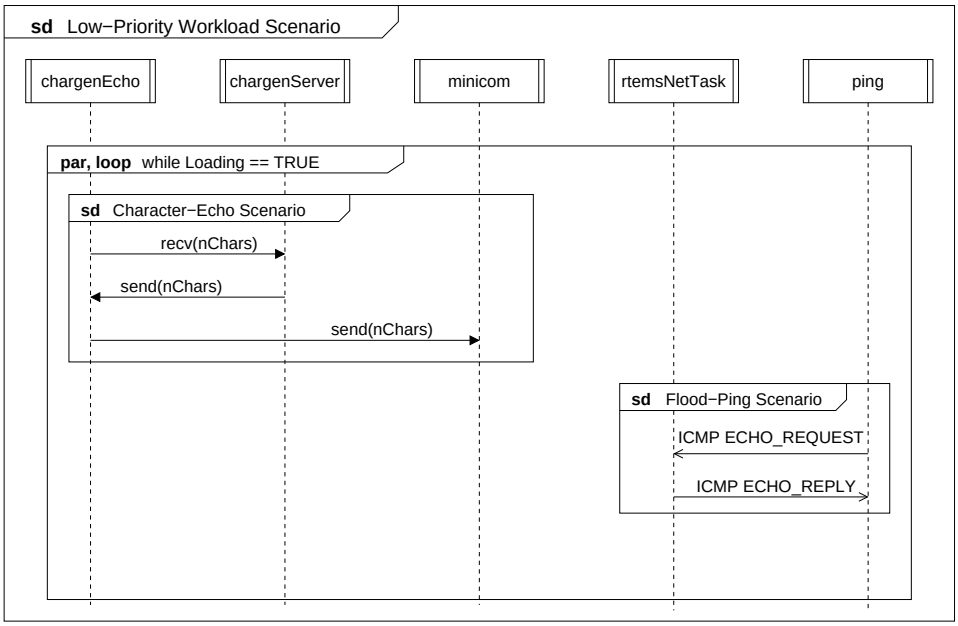

Figure 6.11: Behavior of the character-echo and ping-flood workload.

6.3.4 Data Transfer Rate Measurements

This component of the system performance measurements is perhaps the most conceptually straight forward. As suggested by the section title, the rate of data transfer was measured between the VME modules and the IOM, and between the IOM and Workstation processors. The former test consisted of examining several of the methods possible for data transport across the VME backplane, while the latter testing involved measurement of data rates on the Ethernet link connecting the two processors.

6.3.4.1 Apparatus

The equipment used to determine data rates between the VME bus and the IOM consisted of the same components as described in Section 6.3.2. Measurements of the Ethernet traffic rate between the RTEMS and Linux hosts were recorded simultaneously with the dead time information.

6.3.4.2 Methodology

Several of the available VME data transfer mechanisms were tested, as suggested in [41]. The data transfer methods included memory-mapped read access, as well as 32 and 64-bit block transfers from individual modules. In addition, the chain block transfer functionality of the CAEN digitization modules was also tested for 32-bit data widths. Throughput for

these data access methods was determined by the same technique as that utilized in Section 6.3.1.2. That is, time stamps were obtained from the TSC register to delineate the start and finish of the read routines. The difference of these time stamps provides the duration required to transfer the specified number of bytes from the VME modules to the IOM .

During the testing of Section 6.3.2, data rates were recorded using Lucid's built-in variable, *Lucidbytes*, which maintains a running tally of the number of bytes of data received from the IOM. In this way, it was possible to determine the data rate generated by the acquisition process as a function of input trigger rate.

6.4 Summary

This chapter began with a high-level description of the Lucid DAQ and showed that it may be logically decomposed into three major pieces: the first-level trigger system (TL1), the digitization modules and IOM processor, and the Workstation processor executing the Lucid software application. Although the performance of the TL1 system was not measured, its impact is propagated to the downstream elements of the DAQ through the influence of the INHIBIT circuit. This circuit, in concert with the "event-by-event" global acquisition algorithm, permits application of a simple queueing model to the digitization and IOM components of the Lucid DAQ. In this model, the system may be in only one of two states: BUSY or IDLE. From this description, the input event-loss probability of the DAQ is provided by Erlang's B-formula for the single server queueing system.

Next, the composition of the dead time was examined, revealing that the sequence of events comprising the dead time form fundamental, time-critical operations for the acquisition system. The particular sequence illustrated in Figure 6.2 is dependent upon the algorithm governing the acquisition process, but the same components of the sequence would be present within most data acquisition schemes. Following that, the timing mechanisms utilized in the measurement of Lucid's dead time were described. Finally, a description of the system performance measurements was given, including an account of the methodology and equipment that was used. In the next chapter, the results of the measurements described here will be presented.

CHAPTER 7
DATA ANALYSIS

Data obtained from the measurements described in the previous chapter are analyzed and presented here. These measurements include:

1. CAMAC and TSC clock calibrations,

2. The experiments performed using the photo-nuclear physics equipment at DFELL:

   (a) input signal distributions

   (b) dead time and its major components: Application Response Latency and read-out duration

   (c) Erlang-losses

   (d) Ethernet bandwidth requirements produced by the IOM-to-Workstation data stream

3. A detailed examination of the fundamental processes comprising the I/O Manager's operation. These tests sought to measure the IOM's capabilities in isolation from Lucid application code, and they include measurements of:

   (a) interrupt, context-switch, and Application Response Latency latencies

   (b) VME-to-IOM data transfer duration as a function of software method and data volume.

For the latter items, 3(a) and 3(b), several relevant results from other researchers are shown for relative comparison to the results presented here.

In the following, it is important to note that systematic uncertainties have not been factored into the data analyses: the calculated uncertainties are statistical only.

7.1 Timing Mechanism Calibration

Recall from Section 6.3.1, the devices used to measure the duration of hardware and software events of interest are simple counters, which increment in response to periodic input pulses. In order to convert the dimensionless number of accumulated pulses into a proper time, they must be acquired over a known period of time. This was accomplished by permitting each counter to accumulate pulses for a period of one second, as determined by the hardware/software timing system of the IOM. At the conclusion of each one second interval, the number of counts accumulated by the timing mechanism was recorded. This process was repeated over half-hour and twenty-four hour intervals for the CAMAC and TSC timing systems, respectively. The factors necessary to convert the number of counter "ticks" into a proper time interval were thus obtained.

The number of pulses per period was determined either from a sum of pulses per period, or from the difference between adjacent timestamps, for the SAL clock and the TSC measurements respectively. From this data, key statistical parameters were calculated in the usual manner:

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i \tag{7.1}$$

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_i-\bar{x})^2} \tag{7.2}$$

$$\sigma_{\bar{x}} = \frac{\sigma_x}{\sqrt{N}} \tag{7.3}$$

where $\bar{x}$, σ_x , and $\sigma_{\bar{x}}$ are the average, standard deviation, and standard deviation of the mean (SDOM), respectively, of the number of pulses per sampling period. The results of these calculations are summarized in Table 7.1 for each clock device.

Conversion Factor	$\bar{x}$	$\pm\sigma_{\bar{x}}$
SAL Clock, α_s^{-1}	4989040	20
Pentium III TSC, α_3^{-1}	451029600	1
Pentium IV TSC, α_4^{-1}	2394192114	138

Table 7.1: Clock conversion factors and their uncertainties. Units are *ticks/second.*

For both of the clock systems discussed here, uncertainty due to oscillator drift has

been assumed negligible. That is, the oscillator components of the two clock systems have been treated as being ideal. This permits attributing variation in the counts per sampling period as being due solely to variations in the scheduling of the software mechanism responsible for obtaining the number of pulses. This variability is apparent in the plots of Figure 7.1, each of which is a histogram of the number of oscillator pulses recorded per one second sampling interval, for each clock mechanism.

The inverse of the "tick rates" for each of the clock systems were utilized throughout the remaining analyses to convert the dimensionless number of ticks into properly dimensioned periods describing the events of interest. These conversion factors are summarized in Table 7.1, along with the means, $\overline{x}$, and their uncertainties (SDOM), $\sigma_{\overline{x}}$.

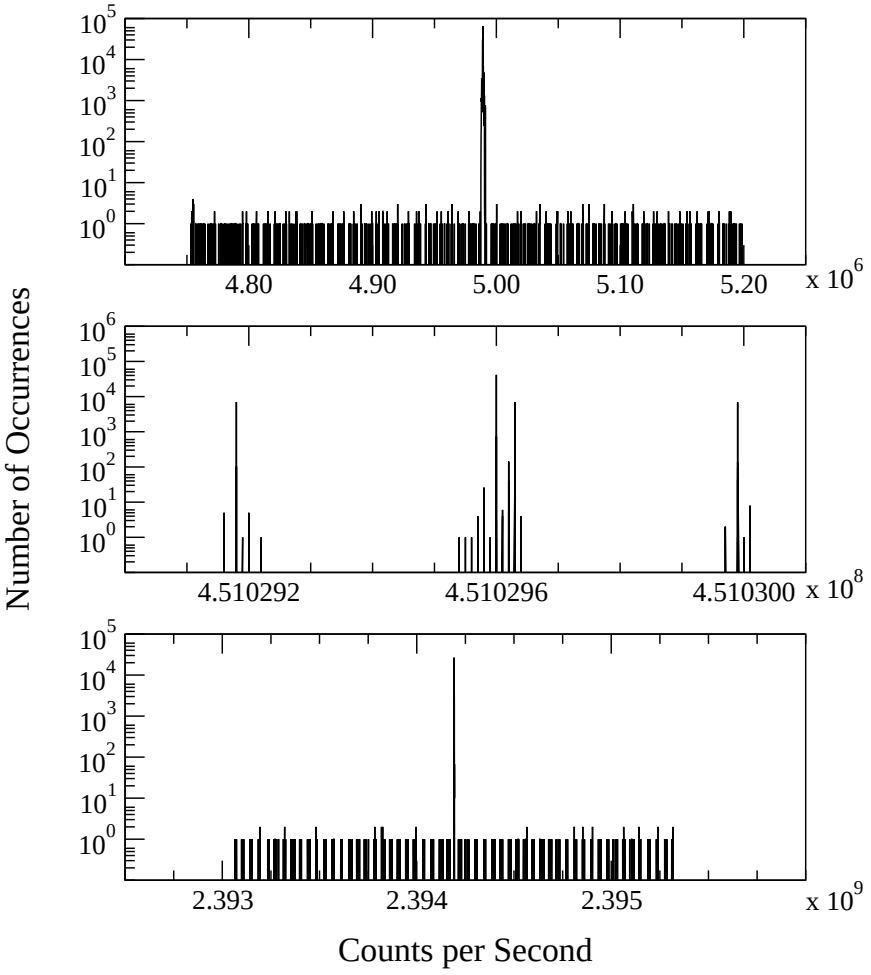

Figure 7.1: Histograms of SAL clock and time-stamp counter (TSC) frequency measurements. Top: CAMAC SAL clock, Middle: 450 MHz Pentium III PC, and Bottom: 2.4 GHz Pentium 4 PC.

7.2 DFELL Results

These measurements were performed to quantify the dead time incurred by the Lucid data acquisition system when it is subjected to realistic loading conditions, as would occur during an actual experiment. Loading was implemented by triggering a logic pulse from constant fraction discriminators in response to above-threshold analog signals generated by the Blowfish PMT array as it detects the decay products of a radioactive source. This hardware-level signal (TL1) may, in turn, cause a VME module to generate an interrupt request, thereby invoking a readout response from the IOM and accumulating dead time. Refer to Figure 6.6 for an illustration of the hardware and control signals used during this phase of testing.

Data-taking sessions were organized sequentially according to their *run number*, with each run further characterized by the presence or absence of several conditions to which the Lucid DAQ is sensitive:

1. The mean frequency of digital output pulses generated by constant fraction discriminators in response to analog input. These are the events that drive the DAQ.

2. The occupancy ratio of the VME digitization modules: they were configured to produce either 3 or 32 channels of input per input event, per module, on a per-run basis (see Section 6.3.2.2).

3. The operation of compression software (i.e. *gzip*) on the Lucid host. This software was expected to negatively affect the acquisition process by triggering the network stack's (TCP) rate-control mechanism, thus throttling the Ethernet data stream produced by the IOM, prior to its being written to permanent storage.

Given their impelling role in the data acquisition process, the time distribution of input events will be examined next.

7.2.1 Input Pulse Distributions

In order to produce input signals to drive this series of measurements, detector elements of the Blowfish array were exposed to the background-radiation signals present in the detector vault at the DFELL. This signal is radiated by surfaces at that location that have become "activated" due to γ-ray bombardment over long periods of time. Although not at physically hazardous levels, the rate of radioactive decays present in the detector vault is substantially greater than the average level of background radiation. The rate at which

Blowfish produces signals was then enhanced and adjusted by altering (lowering) the threshold-voltages of the CFD modules connected to the photomultiplier tubes.

As discussed in Section 6.3.2.1, demonstrating the Poisson-nature of the input signals was important these reasons:

1. Poisson-distributed events are the "most random" type of input signal possible and hence, are an effective stressor of the system under study.

2. Queueing models of DAQ behavior lend themselves well to analytical methods if the input process is distributed in time according to a negative exponential function.

3. Accelerator-induced nuclear reactions are stochastic processes known to approximate a Poisson distribution. Thus, Poisson input processes form a realistic impetus to the system under test.

One method of obtaining a quantitative measure of the agreement between a measured distribution and an analytical distribution is to perform a *chi-squared* (χ^2) test. The χ^2 test provides the means to evaluate how similar a measured quantity is to an assumed form.

In this instance, the number of pulses generated by the CFDs and counted by a CAMAC scaler module (LeCroy 2551) were assumed to obey Poisson statistics. Using a form of χ^2-test known as a *reduced* chi-squared, or χ^2_r , the "goodness of fit" of this assumption may be analyzed using the equation:

$$\chi^2_r = \frac{1}{N-2} \sum_{k=1}^{N} \left(\frac{M_k - E}{\sigma_k} \right)^2 \tag{7.4}$$

where $M_k = \frac{m_k}{T_k}$ is the number of CFD pulses (hence, the number of decays) recorded in the k^{th} sampling interval of duration T, $E = \frac{1}{N} \sum_{k=1}^{N} M_k$ is the number of CFD pulses averaged over the entire run of N samples, and $\sigma_k = \frac{\sqrt{M_k}}{T_k}$ is the uncertainty in the number of CFD pulses per sampling interval, assuming the counts follow a Poisson distribution.

The output of this test is a single scaler value, indicating how well the distribution of measured data agrees with the assumption of a Poisson distribution: results near unity indicate that the errors between the empirical data and the model are randomly distributed. Results differing significantly from unity indicate problems fitting the model to the data, problems with the analysis, or both. The results of performing a reduced-χ^2 test for the data of each run are plotted in Figure 7.2.

While the majority of data points are clustered around $\chi^2 \approx 1$, validating the assumption of a Poisson input signal for those data points, approximately one dozen of the 63

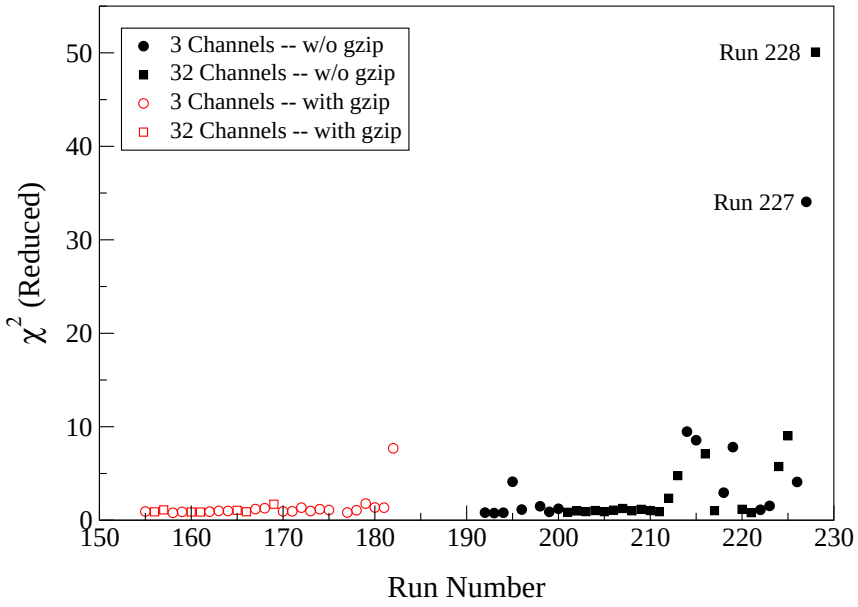

Figure 7.2: Reduced -χ^2 of input pulses. The run numbers are indicative of the sequence of data recording.

data points (most prevalent at the higher run numbers) possess reduced chi-squared values that differ considerably from the ideal value of unity.

Recall from the measurement description provided in Section 6.3.1, that the CFD and SAL-clock pulses were both accumulated in the same CAMAC scaler module and that both values were recorded every $T_k = 0.5$ seconds over each run's duration. Examples of the raw scaler data thus obtained are illustrated in Figure 7.3, for runs 227 and 228, which possess chi-squared values differing from unity by the greatest margin.

Figure 7.3 indicates abrupt deviations from the mean on both channels of the CAMAC scaler. Note the lack of correlation between irregularities present in each channel: if these features were the result of the IOM software being delayed in extracting the values from the scaler, it could be expected that both channels would feature correlated deviations from the norm in proportion to the period that readout was delayed. As this feature is not present, it may be tentatively concluded that the output processes, data extraction by the IOM, was not responsible for the data sets' irregularities. Hence, either the input processes are not well-behaved, or the scaler is miscounting/mis-storing the data.

It is interesting to note that phenomena similar to the effect seen on the scaler channel was also documented during earlier measurements carried out by another member of the same research group as this author [42]. Although unconfirmed, it is suspected that the input signal level "threshold" setting on the CF8000 discriminator may have been adjusted to an unstable setting and/or temperature and fatigue effects may have caused the count-

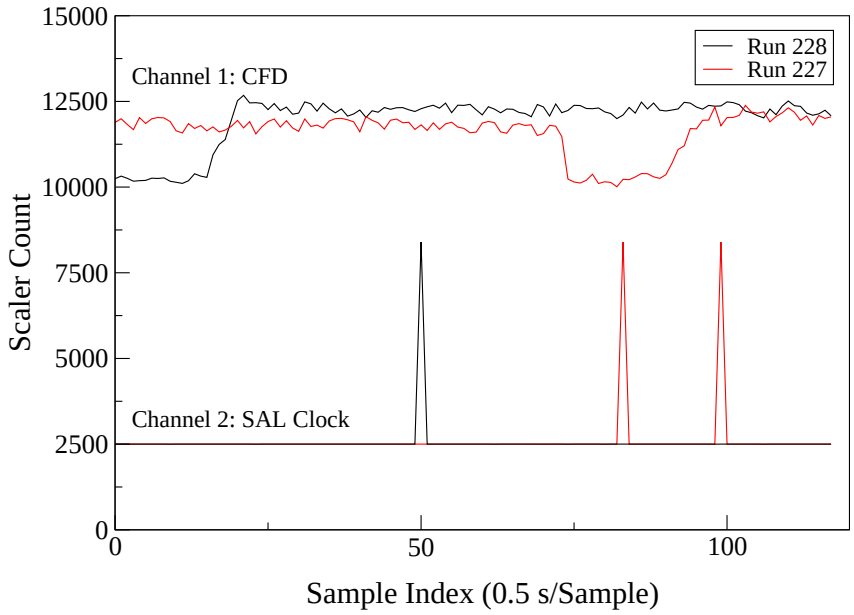

Figure 7.3: Raw scaler counts of input pulses for runs 227 and 228. Note, the SAL clock values have been scaled down by a factor of 10^3 to fit on the same plot.

rate shift seen in the CFD channel. This effect would be orthogonal to any miscounting artifacts produced by the CAMAC scaler device.

7.2.2 Dead Time Measurements

Recall from Section 6.2, the dead time incurred via the event-by-event data readout algorithm may be considered as being composed of the sum of two subintervals:

1. *Application Response Latency* - the period originating with the arrival of a successful TL1 signal and terminating upon the activation of the software application intended to service that event.

2. *Readout Duration* - simply that period required to move digitized event data from the buffers of the VME modules to a buffer on the IOM.

The average dead time of $N \times M$ such events, $\overline{\tau}$, may be calculated using Equation 7.1 and the fact that the average of a sum is equivalent to the sum of the averages:

$$\overline{\tau} = \overline{\tau_1 + \tau_2} = \overline{\tau_1} + \overline{\tau_2} \tag{7.5}$$

$$\overline{\tau}_1 = \frac{1}{NM}\left(\alpha_s^{-1}\sum_{i=1}^{N}\sum_{j=1}^{M}\tau_{ij}\right) \tag{7.6}$$

$$\bar{\tau}_2 = \frac{1}{NM}\left(\alpha_4^{-1}\sum_{j=1}^{NM}\tau_j\right) \tag{7.7}$$

$$\sigma_{\bar{\tau}}^2 = \sigma_{\bar{\tau}_1}^2 + \sigma_{\bar{\tau}_2}^2 \tag{7.8}$$

where $\overline{\tau_1}$ and $\overline{\tau_2}$ are the averages of the Application Response Latency and the readout duration, respectively, and the clock conversion factors (the α's) are as denoted in Table 7.1. Application of these calculations to the data gathered at DFELL is shown in Figure 7.4, where the solid lines indicate the result of applying weighted-average calculations to data sets delineated by the number of channels of data read out per module (i.e. either 3 or 32 channels). The weighted average of a set of data and their uncertainties, $\bar{x_i} \pm \sigma_i$, is given by [23]:

$$\bar{x_w} = \frac{\sum_{i=1}^{N} w_i \bar{x_i}}{\sum_{i=1}^{N} w_i} \tag{7.9}$$

where the "weights", w_i , are simply the inverse squares of the uncertainties associated with the measurement:

$$w_i = \frac{1}{\sigma_i^2} \tag{7.10}$$

and the uncertainty in the weighted-average itself, σ_w , is

$$\sigma_w = \left(\sum_{i=1}^{N} w_i\right)^{-\frac{1}{2}} \tag{7.11}$$

The final result of applying this statistical analyses is a dead time of 50.9 μs for the 3-channel data, and 92.9 μs for the 32-channel data, with null uncertainties within experimental precision.

A note must be made here regarding the uncertainty calculation, σ_1, for the average Application Response Latency, $\bar{\tau_1}$. Recall from Section 6.3.2.2, in order for the measuring process to be reasonably unobtrusive to the workload itself, the clock counts for $N = 1000$ of these events were allowed to accumulate on a scaler channel before readout. Thus, measurements of individual event durations were unavailable for use in calculating the standard deviation of the distribution, as in Equation 7.2 above: only the distribution of the averages was available. However, it may be shown that an estimate of the distribution's standard deviation may still be obtained from the distribution of the averages [43]. This

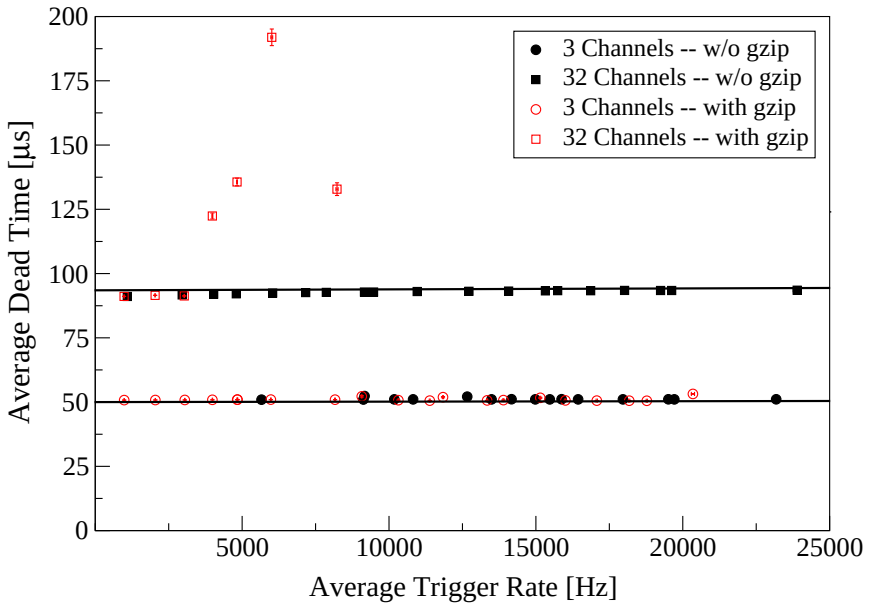

Figure 7.4: Average dead time, $\bar{\tau}$, as a function of average trigger rate, λ. The solid lines indicate the results of the weighted-average calculations.

results in the uncertainty being expressible as:

$$\sigma_1 = \sqrt{N}\sigma_e \tag{7.12}$$

where σ_e is the *standard deviation of the distribution of averages*. That is, σ_e is a product of applying the following statistical considerations to the Application Response Latency measurements, based on the available information and the perturbation constraint:

$$\bar{\tau}_j = \frac{1}{N}\sum_{i=1}^{N}\tau_i \tag{7.13}$$

$$\bar{\tau}_x = \frac{1}{M}\sum_{j=1}^{M}\bar{\tau}_j = \frac{1}{MN}\sum_{i=1}^{MN}\tau_i \tag{7.14}$$

$$\sigma_e^2 = \frac{1}{M}\sum_{j=1}^{M}\left(\bar{\tau}_j - \bar{\tau}_x\right)^2 \tag{7.15}$$

where $\bar{\tau_j}$ is the data read out from the CAMAC scaler averaged over $N = 1000$ events, $\bar{\tau_x}$ is simply the mean of the sum of previous means, $\bar{\tau_j}$, and σ_e is the standard deviation of the distribution of means, $\bar{\tau_j}$.

It must also be noted that 32-channel data-set in which compression software (gzip) was utilized was truncated after approximately 7 kHz input rate due to the adverse impact on system performance (see Section 7.2.3).

7.2.2.1 Application Response Latency and Readout Duration

Of the various means by which the Lucid DAQ may incur dead time, responding to hardware-generated requests for service is the most frequent and arguably the most time-critical. This section presents an analysis of the individual contributions to dead time due to Application Response Latency and data readout duration.

Plotted in Figure 7.5 are the results of applying Equations 7.6 and 7.7 to the dead time measurement data accumulated at DFELL. Again, the solid lines represent the weighted-averages of the data sets: 20.0 μs for the Application Response Latency, and 72.1 μs and 31.1 μs for the 32-channel and 3-channel readout rate cases, respectively. The uncertainties associated with these values is zero within experimental precision.

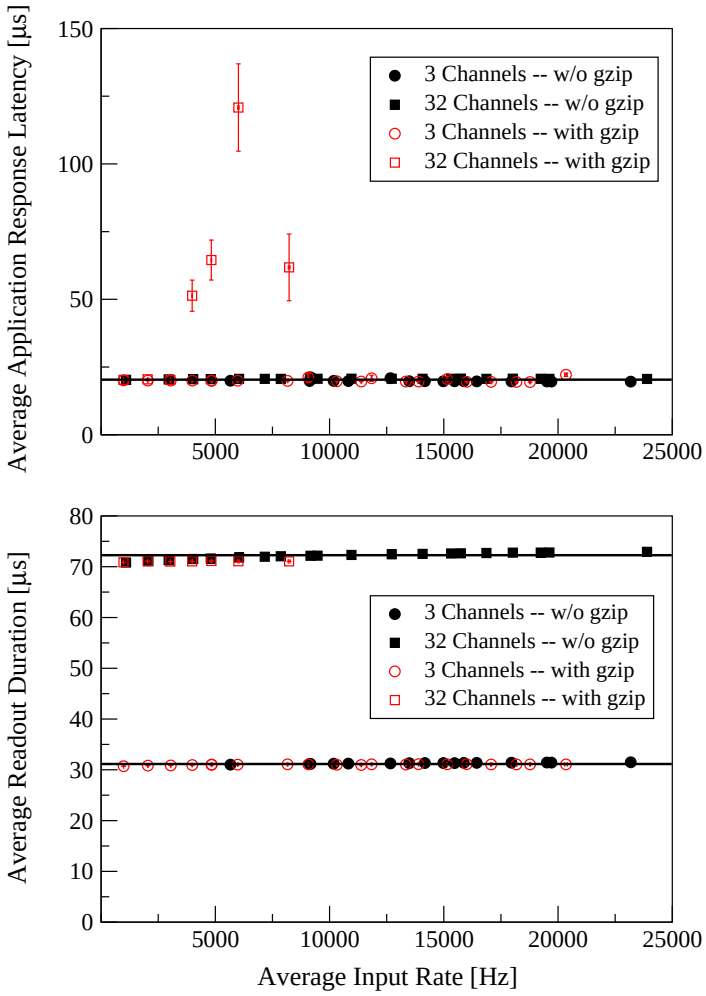

Figure 7.5: Application Response Latency (top) and Readout duration (bottom), as functions of the average input trigger rate. The solid lines indicate the weighted average for each data set.

Note the correlation between the outlying data points in the dead time plot, Figure 7.4,

and the outliers in the Application Response Latency plot of Figure 7.5. Also, note the lack of such outliers in the Readout duration plot of Figure 7.5, thus indicating that some process is prolonging the IOM's average response time.

It is be hypothesized that these outliers are due to an unfavorable combination of factors. First, note that these points correspond only to those data sets where the VME modules were configured to produce 32 channels of data per module, per event, *and* the Lucid host was executing compression software (gzip), in real-time, on the incoming network data stream prior to committing it to backing store. Next, recall the fact that the Application Response Latency period is initiated by a hardware signal, and terminated by a software-initiated signal. Thus, if the software process was occasionally delayed in providing the signal to delineate the end of that period, the average length of the period would be extended in proportion to the frequency and length of those delays.

If the gzip program on the Lucid host cannot process the incoming data stream from Lucid, this will cause Linux kernel's *tcp receive window* to become full, in turn triggering TCP's data-flow control mechanism. In this case, the Lucid host will transparently signal the networking stack of the IOM, indicating that the socket to which *VmeReader* is attached can temporarily accept no more data. This will prevent the IOM from making progress in draining its *DataBuffer*. If this buffer should reach its "high-water mark", the *Acquisition thread* will suspend its operations, becoming delayed in providing the end-of-Application-Response-signal.

There is evidence to support these ideas in the form of the results seen for the average data rate as observed by the *VmeReader* process on the Lucid host. The results of those measurement are presented below, in Section 7.2.3.

7.2.2.2 Erlang Losses

Applying the average dead time and input event rates found in Sections 7.2.1 and 7.2.2 to the Erlang-B equation reveals the event-loss characteristics as a function of input event arrival rate. The Erlang-B equation and its associated uncertainty for the $m = 1$ case are re-expressed here, where the over-score notation has been dropped from those terms denoting mean values:

$$B(\rho = \lambda\tau,\ m = 1) = \frac{\rho}{\rho + 1} \tag{7.16}$$

$$\sigma_B^2 = \left(\frac{\tau}{(\rho+1)^2} \sigma_\lambda \right)^2 + \left(\frac{\lambda}{(\rho+1)^2} \sigma_\tau \right)^2 \tag{7.17}$$

Equations 7.16 and 7.17 were applied to the data of each run, yielding the data points of Figure 7.6. The solid curves in the figure were obtained by using the weighted-average

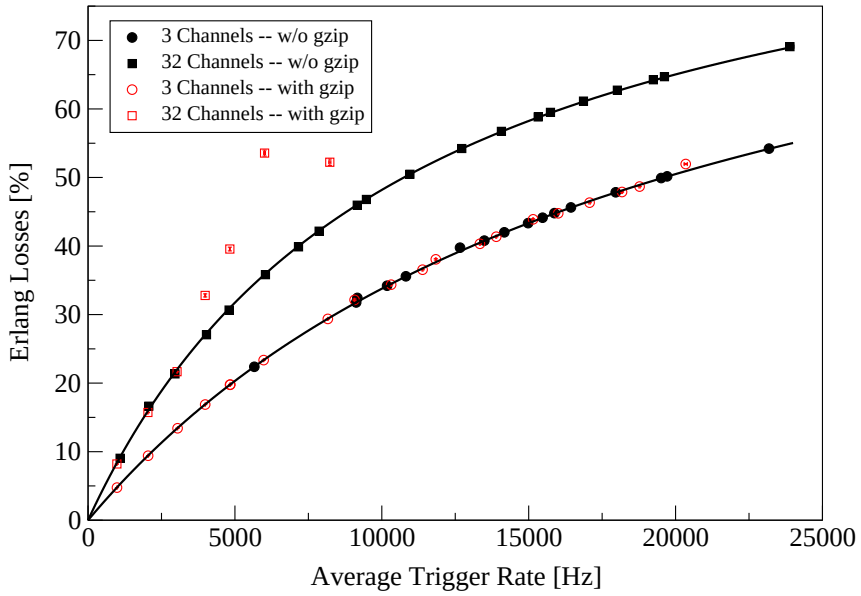

Figure 7.6: Erlang loss as a function of input trigger rate. The solid curves represent the results of using the dead time weighted averages obtained in Section 7.2.2 in the Erlang-B equation, $B(\rho = \lambda\tau, m = 1)$.

values for dead time found in Section 7.2.2.

In light of these results, the DAQ might be further characterized as a type of "filter", with respect to the effect it has on an input information flow. In this sense, one can define its cut-off frequency, or "-3 dB point", as being the average input-event rate for which the DAQ loses 1/2 of all events. Referring to Figure 7.6, these cut-off frequencies occur at approximately 10.5 kHz and 20 kHz for the 32-channel and 3-channel test cases, respectively.

Comparing Figure 7.6 to the analytical model of Lucid, developed in Section 6.1.2 for the event-by-event data acquisition mode, it may be deduced that the model is a sound representation of the DAQ's behavior in this mode of use.

7.2.3 VmeReader Data Rate

These measurements examined the Ethernet data rate produced by the IOM. Rather than instrument the IOM to directly measure this property and risk contributing an artificial workload, two intrinsic features of the *Looker* process on the Lucid workstation were used to monitor this information.

The *Looker* makes several of its internal variables available for programmer use. Two of these were used to monitor the rate of the IOM-produced data stream:

1. *LUCIDbytes* - a running-sum of the bytes accumulated from the data stream, per

run.

2. *LUCIDtime* - a time-stamp, as returned by the *time()* system call. This is applied to the *lh_time* field of a *LucidRecord* structure by the *VmeReader* when it adds a new record to the shared-memory pool, accessible by the *Looker* and *Writer* consumer processes.

Using these variables in an offline-analysis session, the average data rate per run was calculated as:

$$R = \frac{\text{Total bytes per run}}{\text{endrun time} - \text{startrun time}} = \frac{N}{t_1 - t_0} = \frac{N}{\Delta t} \tag{7.18}$$

$$\sigma_R^2 = \left(\frac{\partial R}{\partial N}\sigma_N\right)^2 + \left(\frac{\partial R}{\partial (\Delta t)}\sigma_{\Delta t}\right)^2 = \left(N\frac{(\sigma_{t_1} + \sigma_{t_0})}{(\Delta t)^2}\right)^2 \tag{7.19}$$

where $\sigma_N = 0$ and, due to the one second granularity of the *time*() system call, the uncertainty of the run duration is simply $\sigma_{\Delta t} = 2$ [s]. Application of these two equations to the data gathered at DFELL results in the plot shown as Figure 7.7.

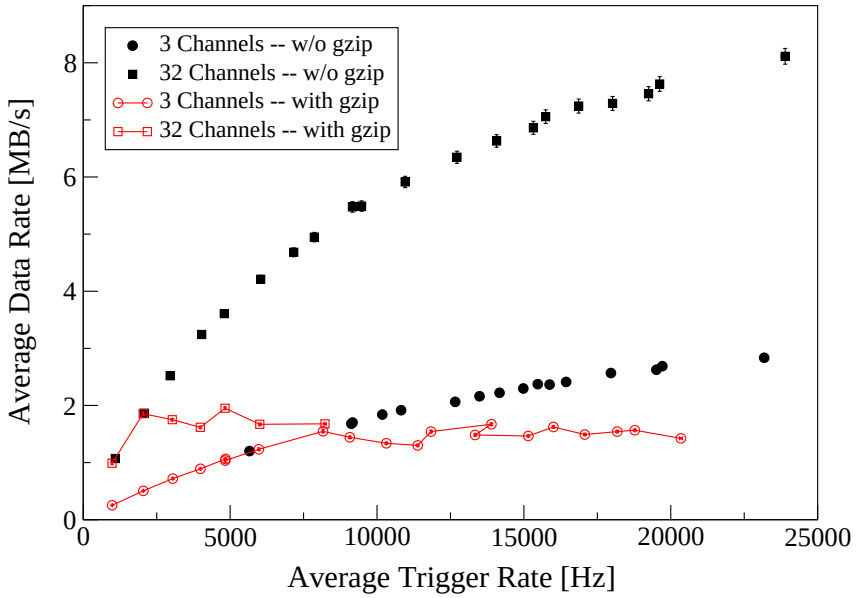

Figure 7.7: Average Ethernet data rate seen by the *Reader* process. Note the apparent "plateau effect" due to compression software affecting the Ethernet data stream.

Ideally, the duration of each test run should be *nearly* the same for the Workstation as it is for the IOM. However, the effect of gzip on the data stream prior to its commitment to permanent storage is readily apparent in Figure 7.7. The effected data points in this plot are illustrated by the solid, red connecting line. The rate at which this "plateau effect"

occurs is dependent upon several factors, such as Workstation CPU and front-side bus rates, the *type* of compression software (i.e. gzip, bzip, compress, etc.) and the tuning options utilized, and even the operating system itself.

7.3 Dead Time Component Results

These measurements are a more detailed study of the processes contributing to the DAQ's dead time. In these tests, the focus is on the IOM processes *in isolation*. That is, in the absence of any IOMReader executable. Although a Linux workstation was used to store test data produced by the IOM and also to provide an artificial workload, Lucid played no role in these tests.

The following timing characteristics comprising the dead time of the IOM were measured:

1. *IRQ latency* - the period originating with the issuance of an interrupt request (IRQ) signal from a CAEN v513 digital I/O module and terminating when the corresponding interrupt service routine (ISR) is dispatched by the IOM.

2. *Context Switch delay* - the period of time required to schedule the next thread to run, save the context of the currently executing thread, load the context of the new thread, and begin execution of the new thread.

3. *Application Response Latency* - formally defined in Section 6.2.4 as the period originating with occurrence of the hardware issued IRQ and ending upon entrance to user application code dedicated to servicing that event. This period encompasses the two previous periods, plus the duration of the VME IACK sequence carried out by the sis1100 driver code.

4. *Data Transfer rates* - the memory-mapped (mmap), block transfer (BLT), and chain block transfer (CBLT) readout rates from CAEN VME modules to the IOM, via the sis100 VME-PCI interface.

While these are all fundamental operations of the IOM, the first three may constitute fundamental operations for *any* I/O-driven computing system.

Section 7.3.1 presents the results of measuring 1, 2, and 3, as previously detailed in Section 6.3.3, as well as reporting results for comparison from the paper that inspired the methodology of those measurements. Section 7.3.2 presents the VME-to-IOM readout rate data, in addition to results for comparison from research examining similar data using several VME-PCI interfaces on a Linux host.

7.3.1 IOM Latencies

The probability distributions of the interrupt, context switch, and Application Response Latency measurements, described in Section 6.3.3, are shown as semi-log plots in Figures 7.8 and 7.9. Table 7.2 summarizes the statistical parameters for each distribution.

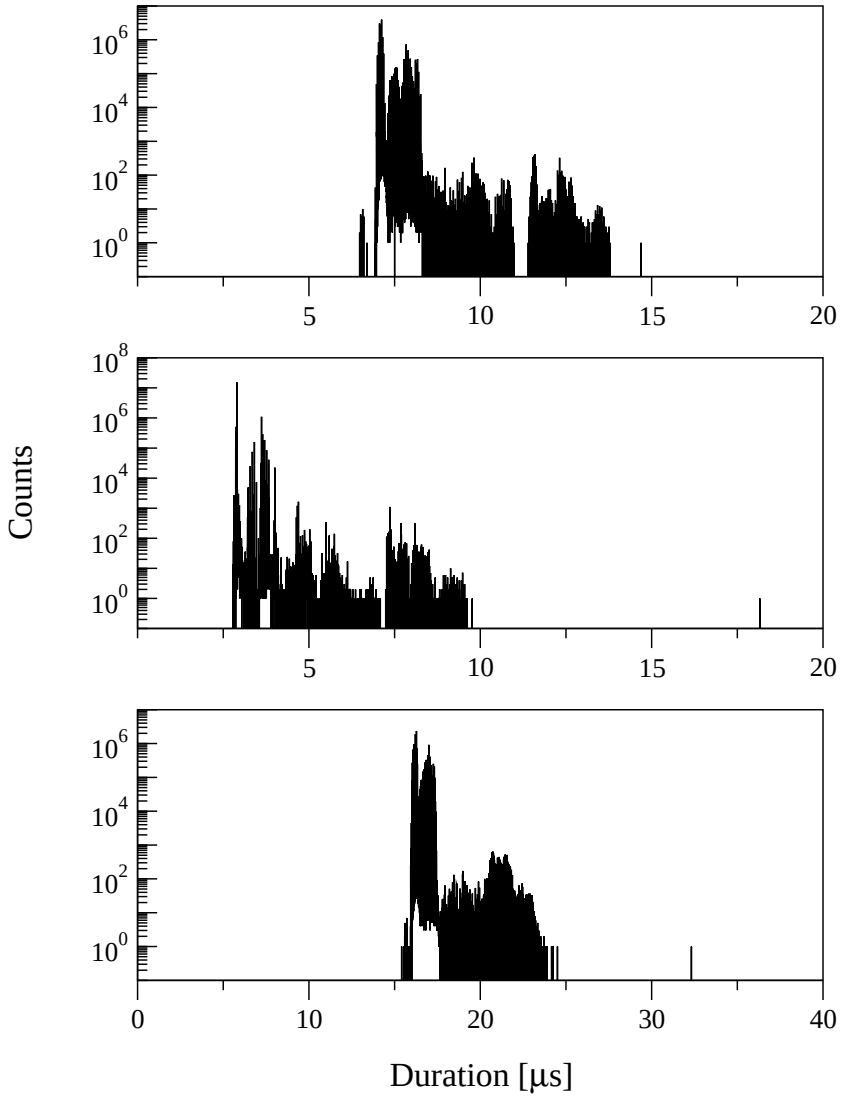

Figure 7.8: Latency distributions for an otherwise idle IOM system. From top to bottom: interrupt, context switch, Application Response latencies.

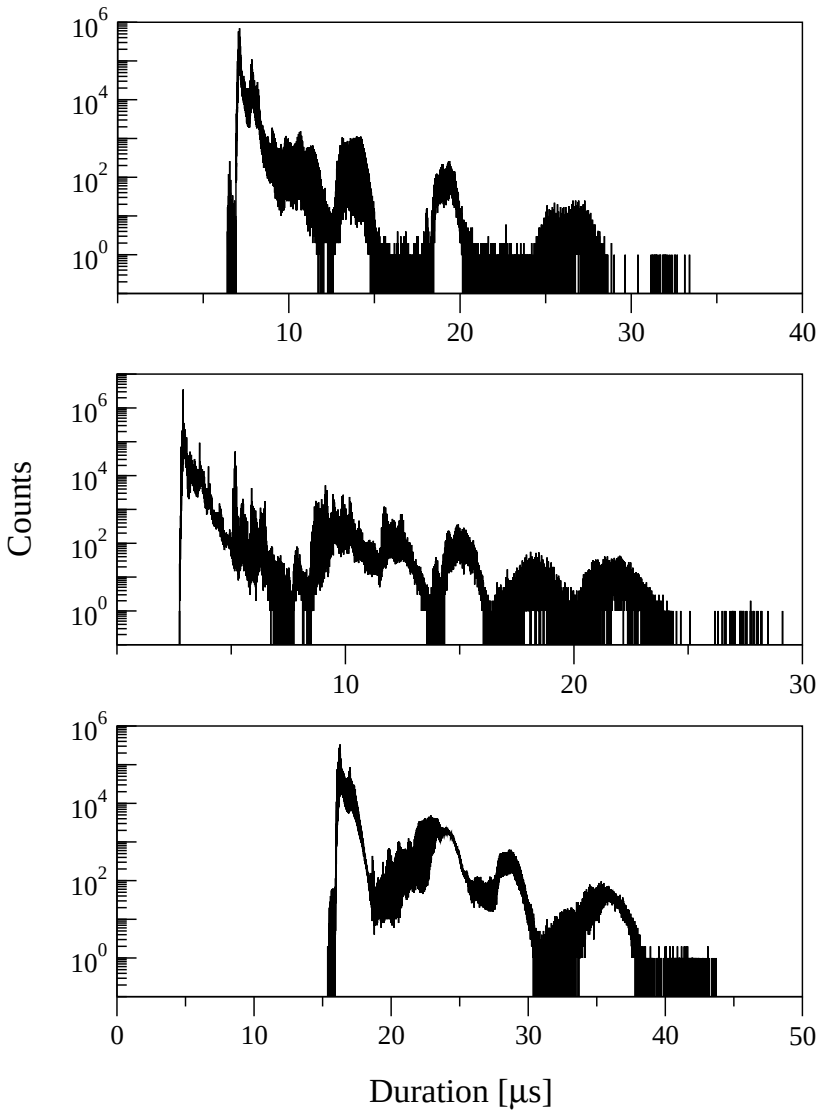

Figure 7.9: Latency distributions for the IOM under heavy loading by low-priority I/O tasks. From top to bottom: interrupt, context switch, and Application Response latencies.

Parameter	Idle System	Loaded System
	max (avg $\pm\sigma$)	max (avg $\pm\sigma$)
Interrupt Latency	14.7 (7.3±0.4)	33.4 (7.5±1.0)
Context Switch Delay	18.2 (3.0±0.2)	29.1 (3.3±1.3)
Application Response Latency	32.3 (16.5±0.4)	43.7 (17.5±2.6)

Table 7.2: RTEMS IOM latency timing results. All times are in units of μs.

Although the general shape of the distributions is consistent between the idle and loaded systems, the effect of loading is readily apparent in Figure 7.9, as the distributions appear to be "smeared" towards greater time values. This is logical, as one might intuitively expect the loaded system to take more time, on average, to accomplish the same set of tasks that it performed when idle. There are simply more tasks competing for processor time and therefore, the probability is proportionally greater that the code path to schedule and execute a thread will be prolonged. Similar reasoning accounts for the scaling of the parameters' maximum values, which are approximately an order of magnitude greater than their mean. The Application Response Latencies are an exception to this, as their maximum values are approximately twice the mean value.

By way of comparison, results obtained for a similar RTEMS system and measurement algorithm are summarized in Table 7.3 [34].

Parameter	Idle System	Loaded System
	max (avg $\pm\sigma$)	max (avg $\pm\sigma$)
Interrupt Latency	15.1 (1.3±0.1)	20.5 (2.9±1.8)
Context Switch Delay	16.4 (2.2±0.1)	51.3 (3.7±2.0)

Table 7.3: Latency measurements for a PowerPC-based RTEMS system [34]. All times are in units of μs.

In that paper, the system under test was a MVME2306 PowerPC 604, with a CPU clock frequency of 300 MHz running the RTEMS operating system with its native API. The results obtained in this thesis for the 450 MHz PC fall within a similar numeric range.

It should be noted that the IRQ source in "Open Source Real Time Operating Systems Overview" was an on-board timer, whereas the source used in the tests of this thesis was an external device: a CAEN v513 module connected to the IOM via the sis1100 VME-PCI bridge and gigabit optical fiber. After all, the objective of this test was not to test the latency of RTEMS on the pc386 BSP, but to test the latency of the *system* in a global sense, using the same equipment as would be utilized in a nuclear physics experiment.

As a final note for comparison, measurements of the scheduler latency of Linux reported in one research paper (conceptually equivalent to the *context switch latency*, de-

fined here) indicate that while the average result was within one order of magnitude to that found for RTEMS in this thesis, the worst-case latencies differ from the average by approximately three orders of magnitude [44]. Similar results were found in other studies of Linux's scheduling capabilities [8] [45]. These findings suggest that the quality of service (QoS) achievable for a data acquisition system application under Linux may be considerably less than what is possible by using a real-time executive for a similar application.

7.3.2 VME to IOM Data Transfer Rates

These tests are a measure of data transfer duration as a function of transfer size and technique. Studying the duration of the software methods controlling the VME-to-PC transfer of data reveals two pieces of information:

1. the minimum duration of the method. This may be thought of as the minimum "cost" of invoking a particular read method.

2. the rate of data transfer from the VME module(s) to the PC, as determined by the VME access type (CBLT, etc.) and byte-width.

The former data indicates the minimum execution time of a particular "read" method, and hence, it's applicability in scenarios where time-constraints exist, while the latter is simply a measure of the bandwidth used to transfer data between the CAEN VME modules and the IOM. Both are important parameters to know when designing software for an experiment.

Figure 7.10 portrays A24/D32 memory-mapped accesses (simple pointer de-referencing operations in the C programming language). Figure 7.11 illustrates A24/D32 and A24/D64 block transfers from a CAEN v862 QDC module, as well as A32/D32 chain block transfers from the chain of modules described in Section 6.3.2.1. All data points in these plots are statistical averages, calculated in the sense of Equation 7.1, and the solid lines are the result of applying linear regression analysis (i.e. the method of least squares) to those data points. The equations resulting from the linear regression analysis are also shown in each plot accompanying their appropriate data set. The information determined from these analyses are summarized in Table 7.4.

Based on the results of this analysis, it is now possible to determine the *DMA threshold* for the IOM system. This may be defined as the minimum amount of data that must be present for read out such that it becomes more efficient to use BLT/DMA operations, rather than single-cycle VMEbus accesses. The existence of this threshold is due to the fact that memory-mapped, single-cycle VME accesses are both atomic and deterministic,

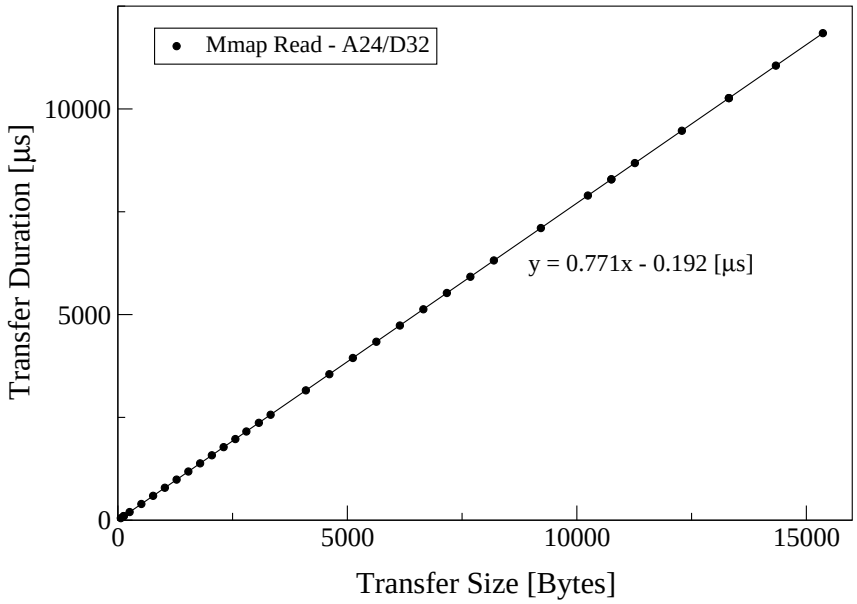

Figure 7.10: Duration of memory-mapped (mmap) read versus transfer size. Readout was performed from a single VME module, a v862 QDC.

Method	Rate [*MB/s*]	Min. Setup Time [μs]
Mmap – D32	1.3	-0.2±0.3
BLT – D32	18.2	18.2±0.4
BLT – D64	25.5	18.8±0.1
CBLT – D32	17.9	24.1±0.0

Table 7.4: VME-to-IOM data transfer rates.

while block transfers possess neither of those qualities. In their present implementation, the block transfer software methods are synchronous, and hence will block while waiting for data.

The DMA threshold may be calculated by equating the linear regression D32 BLT and CBLT equations to the D32 mmap equation and taking the mean of the resulting values. This operation produces a value of 30 bytes, or approximately 8, 4-byte data words. Note, this result holds only for the combination of the RTEMS version of the sis1100 VME-PCI device driver and the particular VME modules utilized in this portion of testing.

It is interesting to note that version 2.02 of the Linux/NetBSD driver distributed for use with the sis1100 interface defines a similar threshold, with the exception being that software sets a default threshold value of 24 words, or 192 bytes. This discrepancy may be attributable to the hardware used during that testing, or to the greater overhead incurred as a result of operating within the Linux and NetBSD kernels, or to a combination of the

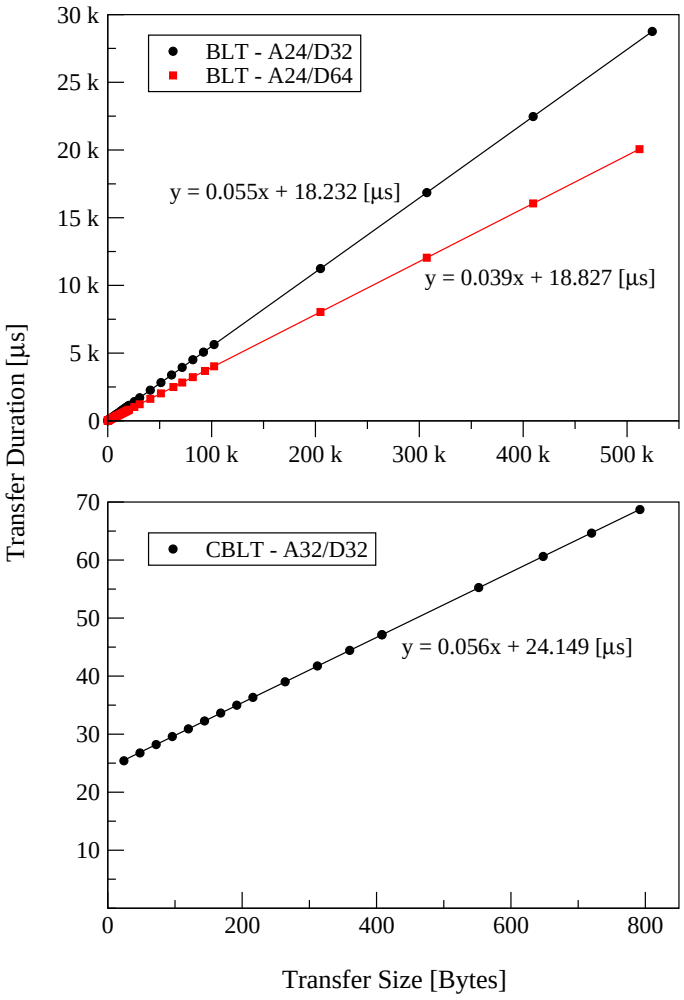

Figure 7.11: Block transfer (BLT) duration as a function of transfer size. Top: BLT duration from a single CAEN v862 module. Bottom: Chain Block Transfer (CBLT) duration for a VME Chain consisting of 7 QDC and 3 TDC modules**.**

two factors.

The results obtained for this section of testing may be compared to those found in one unpublished report, whose subject was performance measurements of several different VME-PCI bridge devices operating under Linux 2.2.12 on a dual-CPU, 400 MHz Pentium II platform [41]. Such a comparison serves as a measure of the performance of the sis1100 VME-PCI interface under RTEMS, relative to other systems with similar functionality.

Although the report's author did not test the interrupt subsystems of the various interfaces, and a precise description of the timing mechanism was also missing, performance figures for readout rates from a VME RAM module were available, and it is these results which have been compiled in Table 7.5. Three interfaces were tested in that report, but the results of only two are shown here, as tests of the omitted device shared no points of

comparison in common with the measurements of this section of the thesis.

Method	National Instruments VME-PCI 8026 kit	SBS Bit 3 Model 617 Adapter
Mmap - D32	2.1 MBps	1.5 MBps
BLT - D32	11.5 MBps	22.5 MBps
Setup Time	50 μs	150 μs
BLT - D64	13.6 MBps	N/A
Setup Time	50 μs	

Table 7.5: Performance figures for several VME-PCI bridge devices on a Red Hat Linux system [41].

7.4 Summary

This chapter presented an analysis of the data obtained by the methods described in Chapter 6, providing several performance metrics of the Lucid data acquisition system, including dead time, event-loss and data transfer rates, and IOM latencies.

An investigation of the timing mechanisms used to measure the duration of events of interest produced several conversion factors necessary to transform unitless pulse counts into properly dimensioned periods. These conversion factors were utilized in each of the remaining experiments.

A study of the data gathered using the experimental apparatus at the Duke Free-Electron Laser laboratory generated values for several important DAQ parameters. Although it was discovered from the analysis of data gathered at Duke Universiy that portions of input event inter-arrival time data could not be definitively catagorized as originating from Poisson processes, the behavior of the DAQ was found adhere to a simple, analytical model of an Erlang-loss system with queueing parameters M/G/1/1.

It was also found that the independent conditions of real-time compression software use and the IOM's production of large data volumes may seriously compromise the quality of service of data acquisition.

Data analysis of IOM latencies and VME-to-IOM data transfer rates indicate that the performance of the IOM in these areas is comparable to figures produced elsewhere using similar equipment and measurement strategies.

CHAPTER 8
SUMMARY AND CONCLUDING REMARKS

8.1 The Upgrade Project

Contemporary particle-physics measurements rely on automated data acquisition systems (DAQ) to process information generated by experiment-specific detectors. Given the probabilistic nature of the data involved, numerous measurements must be obtained in order to achieve a satisfactory level of statistical uncertainty. If experimental events of interest occur relatively infrequently, the duration required to perform an experiment may become prohibitively lengthy. The same prohibitive duration might also arise if the DAQ is unable to acquire data at a rate sufficient to produce the necessary level of statistical uncertainty.

The data acquisition system known as *Lucid*, developed and employed by U of S researchers, was the subject of a comprehensive upgrade project described herein. The intent of these upgrades was to capitalize on increased input event rates, such as those expected from increases in γ-ray flux produced by enhancements to the High Intensity Gamma-Source facility where the U of S researchers travel to conduct experiments.

Responsibility for the digitization of analog detector signals was largely displaced from the domain of CAMAC to VME modules. This change reduced data conversion times from 60 μs to less than 10 μs, while providing approximately twice the channels-per-module density available with the CAMAC systems at hand. This hardware change also introduced an increase in theoretical maximum data transfer rates across the backplanes of the CAMAC and VME crates from 3 MB/s to 40 MB/s, respectively.

Zero/overflow suppression is an operational characteristic of the VME modules employed in this phase of the Lucid upgrade project. This feature optionally excludes converted data values falling outside an adjustable range, thereby eliminating the processing time requred to achieve the same effect. This also reduces the volume of data transferred from the modules to downstream, DAQ processors.

Another beneficial feature gained by the migration from CAMAC to VME is support for configuring a group of contiguous modules to respond to accesses as though the group were a *single, virtual* module. In this configuration, the virtual module supports both

chained block read-outs (CBLT) and *multicast writes* (MCST). A CBLT read access to the virtual module causes all members of the group to respond in sequence with their available data, while a single MCST write will be propageted sequentially to all modules in the group. This capability provides an economical means of VME bus access, as the bus master need conduct only a single transaction to access all members of "the chain."

From its origin in the VME digitization modules, data is transferred to a processing node denoted here as the *I/O Manager*. In this role, the previous incarnation of Lucid featured a 40 MHz MVME-167 single-board computer embedded in the VME back-plane. This system accessed modules on the back-plane via an ASIC, the VMEChip2, which was unable to perform block transfers from VME slave modules to the host system's memory.

The post-upgrade system now utilizes the common desktop PC platform, the *x86* architecture, to host the I/O Manager software. The production system has a 2.4 GHz Pentium 4 CPU, and is coupled to the VME hardware by a VME-PCI interface (sis1100) produced by Struck Innovative Systems. The sis1100 interface consists of two modules communicating over a gigabit fiber optic link: one module resides on the PCI bus of the I/O Manager, while the other sits on the VME bus. The sis1100 is capable of block transfer as well as single-cycle VME bus transactions, and it is able to perform them at rates of 18 MB/s and 1.3 MB/s, respectively.

The advent of new hardware and its new capabilities demanded the addition of software to support them. As a first step in the modification of Lucid's software components, the real-time operating system of the I/O Manager was shifted from a commercial product, pSOS+, to the open-source freely available Real-Time Executive for Multi-processor Systems (RTEMS). The choice of RTEMS was motivated by several reasons: 1) substantial "in-house" experience was available, 2) prior negative experiences with the vendor of pSOS+, 3) zero purchase price and complete access to *all* source code, 4) a responsive user community was available for consultation, and 5) the presence of features unavailable with pSOS+, such as runtime (dynamic) loading and linking.

The effort to convert pSOS-dependent code to RTEMS was greatly eased due to the programming interface they shared in common, the Real-Time Executive Interface Definition (RTEID). Although the function names differed between the two real-time operating systems, equivalent functions were available across both. Thus, porting code from pSOS+ to RTEMS was often as simple as directly substituting a pSOS+ function with the RTEMS equivalent.

Device driver source code supplied with the sis1100 bridge device supported only the Linux operating system and therefore had to be extensively modified to work from an RTEMS platform. This work was of considerably more complexity than that involved in

porting the I/O Manager software from pSOS+ to RTEMS, requiring driver modifications in the areas of operating system integration and driver initialization, PCI bus access methods, interrupt handling, execution context, and inter-process communication. However, the highest level API from the original Linux driver was retained and even expanded upon in this phase of the Lucid upgrade project.

System descriptions provided in Chapters 1-3 proceeded from the highest levels of architectural abstraction, Lucid's use cases and sub-systems, through to the mid-level, or *mechanistic* design, where the collaborative roles of software objects are described, and finally down to the detailed design level where individual software components are layed out.

This thesis contributes to the *vocabulary* of the Lucid data acquisition system by providing documentation, design descriptions, and terminology for sub-systems where previously there had been none. In the past, components such as the I/O Manager and `IOMReader` application lacked any concrete identity, or clear statement of the functionality they provide. As mentioned in the discussion of design patterns (see Appendix C), the importance of providing a common vocabulary is critical for discussion of the system under study.

8.2 DAQ Modeling and Performance Measurements

The perspective and techniques afforded by Queueing theory were applied to mathematically model Lucid's "event-by-event" data acquisition scheme as an *M/G/1/1* queueing system. Results of this model were demonstrated to be reconcilable and equivalent with the traditional concept of *dead time*, long familiar to experimental physicists.

The dead time of a data acquisition system was defined to be (see Chapter 4),

> "...a time interval following a registered (or detected) event, during which the counting system is insensitive to other input events".

Dead time causes a loss of input events, thereby extending experiment duration, as well as distorting the statistical distribution of input event inter-arrival times. Thus, dead time is a key property of data acquisition systems that should be both minimized and well characterized.

The dead time, its constituent sub-intervals, and the data transfer rate properties of Lucid were measured in a series of tests conducted under realistic conditions and workloads. Lucid's dead time was found to be comprised of three main sub-intervals: 1) the VME

module digitization period, 2) the Application Response Latency, and 3) the VME-to-I/O Manager data transfer duration.

The Application Response Latency is a performance metric contributed by this work and defined as (see Section 6.2.4),

> "...that measured on a heavily loaded system for the period originating with the occurrence of a hardware interrupt request and terminating upon entrance to application code dedicated to the handling of that event".

In this way, the definition encompasses interrupt latency, context switch delays, and the cost of performing a VME interrupt-acknowledge sequence.

Measurement results for the average Application Response Latency, data transfer duration, and dead time are listed in Table 8.1 for data volumes representative of worst-case experimental conditions.

Parameter	$\frac{3\frac{channels}{module \cdot event}}{\left(\sim 0.15\frac{kB}{event}\right)}$	$\frac{32\frac{channels}{module \cdot event}}{\left(\sim 1.0\frac{kB}{event}\right)}$
Application Response Latency	20.0 μs	20.0 μs
Readout Duration	31.2 μs	72.1 μs
Dead Time	50.9 μs	92.9 μs
-3 dB input-frequency	20 kHz	10.5 kHz
Avg. Output Bandwidth @ -3 dB input-frequency	2.7 MBps @ 20 kHz	5.9 MBps @ 10.5 kHz

Table 8.1: Summary of IOM performance data measured at DFELL.

Also listed in Table 8.1 are figures for a "-3 dB" input rate and the average Ethernet bandwidth usage at those frequencies. In analogy with a filter "corner" frequency, the "-3 dB" point corresponds to that input event frequency at which Lucid will lose one-half of all events.

Given the detrimental effect observed in this testing phase due to the use of compression software on the real-time data stream, its use on raw, Ethernet data rates above 1.5 MB/s cannot be recommended, or minimally, it should be used with a cautious eye towards the rate of information flow from the IOM to the user's Workstation.

The "event-by-event" data collection policy presently employed with the Lucid DAQ places the responsibility for clearing the TL1 INHIBIT signal with the I/O Manager. Thus, the dead time incurred for events triggered at the TL1 level is largely governed by the duration of software processes. The negative impact of this policy may be mitigated by displacing to hardware the responsibility for removing the INHIBIT condition (see Appendix E).

In order to study in finer detail the sub-intervals of Lucid's dead time, another series of measurements was conducted targeting those periods comprising the Application Response Latency. The interrupt latency, context switch delay, and Application Response Latency were quantified independant from any Lucid-specific code. The results of these tests are summarized in Table 7.2, reproduced here in Table 8.2 for ease of viewing. These

Parameter	Idle System		Loaded System	
	max	(avg $\pm\sigma$)	max	(avg $\pm\sigma$)
Interrupt Latency	14.7	(7.3±0.4)	33.4	(7.5±1.0)
Context Switch Delay	18.2	(3.0±0.2)	29.1	(3.3±1.3)
Application Response Latency	32.3	(16.5±0.4)	43.7	(17.5±2.6)

Table 8.2: RTEMS IOM latency timing results. All times are in units of μs.

results were found to be numerically comparable to published figures obtained using similar hardware.

The VME-PC data transfer rates were measured for a variety of read-methods and data widths, the results of which are listed in Table 7.4, also reproduced here in Table 8.3. These tests revealed not only the maximum rate of transfer obtainable, but also the minimum time required to employ each read-out technique. The values obtained here compared favorably to those found in an unpublished work exploring similar systems.

The data listed in this table also permits calculation of a "DMA-threshold": the volume of data required such that it is more economical to perform a DMA-based block transfer than single-cycle read accesses. This "threshold" was found to be approximately 8, 4-byte words.

Method	Rate [*MB/s*]	Min. Setup Time [μs]
Mmap – D32	1.3	-0.2±0.3
BLT – D32	18.2	18.2±0.4
BLT – D64	25.5	18.8±0.1
CBLT – D32	17.9	24.1±0.0

Table 8.3: VME-to-IOM data transfer rates.

Based on the results of these tests, it may be concluded that the combination of common, desktop hardware and open-source RTOS can be used to realize an acceptable quality of service for the Lucid data acquisition system.

8.3 Ideas for Future Investigation

8.3.1 IOM API Changes

Code which utilizes an API based upon the ORKID/RTEID specs may not be made to execute on general purpose systems such as Linux, thus precluding the use of those systems as both development *and* testing platforms. What is needed is a kernel API common to both the development and deployment platforms. The POSIX API satisfies this requirement.

The RTEMS implementation of the POSIX API consists of "wrappers" around the Classic API, thus providing an inexpensive means (in terms of execution time) to develop software capable of executing on both Linux workstations and RTEMS targets. That is, software may be developed, tested, and debugged on the host workstation *prior* to deployment and final testing on the intended RTEMS target platform.

8.3.2 Modifications to Sis1100 Device Driver

The ability to attach user-defined callbacks to special events, such as link-loss and link-reacquisition, would be a useful feature, and provide additional robustness to the instrumentation interface.

Providing a VME API as described in a CERN technical report for the ATLAS detector may aid portability across different platforms and applications [46]. As it now exists, *libVME* is largely compliant with the services outlined in the CERN document, and would require mainly cosmetic refactoring to achieve full compliance.

8.3.3 Acquisition-DataWriter Interaction

Instead of the current semaphore-based synchronization scheme, the interaction between the
`Acquisition` and `DataWriter` threads may be better suited to a message-queue-based system. This configuration would simply consist of having `Acquisition` send messages to the queue, on which the `DataWriter` would block-waiting for activity. The messages would be drawn from a fixed-sized, resource pool and need only consist of a pointer to the next `DataRecord` to be transferred to the Lucid host. If the pool of messages becomes exhausted, this would force `Acquisition` to block, while `DataWriter` consumes and frees resources necessary for `Acquisition` to resume execution.

REFERENCES

[1] J. Park and S. MacKay, *Practical Data Acquisition for Instrumentation and Control Systems*. Elsevier, first ed., 2003.

[2] T. A. Collaboration, "Atlas detector and physics performance," tech. rep., ATLAS Collaboration, 1999.

[3] H. R. Weller, "The higs facility: A free-electron laser generated gamma-ray beam for research in nuclear physics," *Modern Physics Letters A*, vol. 18, pp. 1569–1590, 2003.

[4] P. C. Johns and *et al*, "Medical x-ray imaging with scattered photons," in *Opto-Canada: SPIE Regional Meeting on Optoelectronics, Photonics, and Imaging*, The International Society for Optical Engineering, 2002.

[5] R. K. Bock and *et al*, *Data Analysis Techniques for High-Energy Physics*. Cambridge University Press, second ed., 2000.

[6] D. Murray, "Data acquisition and analysis systems." SAL Internal Report: Computers and Control System, 1988.

[7] D. Murray, "Lucid: A unix-based data acquisition and analysis system for nuclear physics," in *IEEE Seventh Conference Real-Time '91 on Computer Applications in Nuclear, Particle, and Plasma Physics*, IEEE Nuclear and Plasma Physics Society, 1991.

[8] L. Abeni, "A measurement-based analysis of the real-time performance of linux," in *IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, (San Jose, California), September 2002.

[9] W. E. Norum, "An improved data acquisition system at the saskatchewan accelerator laboratory," *IEEE Transactions on Nuclear Science*, vol. 41, pp. 52–54, 1994.

[10] W. D. Peterson, *The VMEbus Handbook*. VMEbus International Trade Association, fourth ed., 1997.

[11] D. H. R. Larsen, "Camac: A modular standard," *U.S. NIM Committee: CAMAC Tutorial Articles*, pp. 9–14, 1976.

[12] A. Rubini and J. Corbet, *Linux Device Drivers*. O'Reilly & Associates Inc, second ed., 2001.

[13] The PCI Special Interest Group, *PCI Local Bus Specification*, 1995. Revision 2.1.

[14] R. J. O'Conner, "Interfacing vmebus to pcibus," *VMEbus Systems Magazine*, pp. 1–4, 1996.

[15] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley, third ed., 2004.

[16] OAR Corp, *RTEMS Applications C User's Guide*, 2005.

[17] V. Yodaiken, "Against priority inheritance," tech. rep., FSMLabs, 2002.

[18] The PCI Special Interest Group, *PCI BIOS Specification*, 1994. Revision 2.1.

[19] J. Thorpe, "A machine-independent dma framework for netbsd," in *USENIX Conference*, USENIX, 1998.

[20] J. W. Muller, "Particle counting in radioactivity measurements," tech. rep., International Commission of Radiation Units and Measurements, 1995.

[21] G. F. Knoll, *Radiation Detection and Measurement*. Wiley Inc, third ed., 2000.

[22] J. W. Muller, "Dead-time problems," *Nuclear Instruments and Methods*, vol. 112, pp. 47–57, 1973.

[23] J. R. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, second ed., 1997.

[24] W. R. Leo, *Techniques for Nuclear and Particle Physics Experiments*. Springer-Verlag, second ed., 1994.

[25] R. B. Cooper, *Introduction to Queueing Theory*. North Holland Inc, second ed., 1981.

[26] L. Kleinrock, *Queueing Systems Volume II: Computer Applications*. Wiley Inc, first ed., 1976.

[27] R. C. Larson and A. R. Odoni, *Urban Operations Research*. Prentice-Hall, first ed., 1981.

[28] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling*. Wiley Inc, first ed., 1991.

[29] D. M. Gingrich and *et al*, "A pipeline controller for the atlas calorimeter," in *Electronics for Particle Physics Conference*, LeCroy Corporation, 1997.

[30] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory*. Wiley Inc, third ed., 1998.

[31] L. Kleinrock, *Queueing Systems Volume I: Theory*. Wiley Inc, first ed., 1975.

[32] Merriam-Webster Inc., *Merriam-Webster's Dictionary*, 2002.

[33] CAEN Nuclear Instrumentation, *Technical Information Manual: Mod. V792 series 32 Channel QDCs*, 2003. Revision 9.

[34] T. Straumann, "Open source real time operating systems overview," in *8th International Conference on Accelerator and Large Experimental Physics Control Systems*, (San Jose, California), 2001.

[35] FSM Labs Inc, *The RTCore Index*, 2005. Online document: available at *http://www.fsmlabs.com/the-rtcore-index.html*.

[36] Hewlett Packard, *Application Note 1289: The Science of Timekeeping*, 1997.

[37] Intel Corporation, *The IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2002.

[38] Intel Corporation, *82C54 CHMOS Programmable Interval Timer*, 1994.

[39] J. Postel, *RFC 864: Character Generator Protocol*. Network Working Group, 1983.

[40] J. Postel, *RFC 792: Internet Control Message Protocol (ICMP)*. Network Working Group, 1981.

[41] K. Schossmaier, "Assessment of vme-pci interfaces with linux drivers." CERN Internal Note: ALICE DAQ, July 2000.

[42] J. Ives, "Simulation and measurement of the response of the blowfish detector to low-energy neutrons," Master's thesis, University of Saskatchewan, 2003.

[43] R. Igarashi, *Private communication*, August 2005.

[44] C. Williams, *Linux Scheduler Latency*. Red Hat Inc, 2002. Online document: available at *http://www.linuxdevices.com/files/article027/rhrtpaper.pdf*.

[45] A. Goel, "Supporting time-sensitive applications on general-purpose operating systems," in *Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Boston, MA), December 2002.

[46] R. Spiwoks and *et al*, *VMEbus Application Program Interface*, 2001.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Inc, first ed., 1996.

[48] T. W. Pratt and M. V. Zelkowitz, *Programming Languages: Design and Implementation*. Prentice-Hall Inc, third ed., 1996.

[49] D. Watson, *High-Level Languages and Their Compilers*. Addison-Wesley, first ed., 1989.

[50] M. E. Lesk, "Lex - a lexical analyzer generator," tech. rep., Computing Science Technical Report 39, Bell Laboratories, 1975.

[51] S. C. Johnson, "Yacc: Yet another compiler compiler," tech. rep., Computing Science Technical Report 32, Bell Laboratories, 1975.

[52] A. D. Frari, *Lucid and Vme Modules*. U of S Subatomic Physics Institute, first ed., 2003. SPIR-122.

[53] R. Igarashi, A. D. Frari, D. Chabot, and T. Regier, *Lucid: User's Guide*. Subatomic Physics Institute, University of Saskatchewan, third ed., 2004. D. Murray, W.E. Norum, T. Wilson, G. Wright (First & Second Ed).

[54] D. C. Nygren, "Effective vmebus memory mapping," *Communication System Design Magazine*, pp. 36–42, 1999.

APPENDIX A
SOFTWARE ACCESSIBILITY

Source code and/or binaries for the software developed as part of the work contained herein may be obtained by directing requests to:

daron.chabot@usask.ca

or,

daron@nucleus.usask.ca

APPENDIX B
EDL-GENERATED SOFTWARE: MYEXPERIMENT.R

See EDL source code in Section 1.3.2.

```
/* myExperiment.frontend.h */
/*
* Event 'ReadADC' (0) is triggered:
* every 0.1 seconds
*/
#define NReadADC 0
struct EReadADC {
  unsigned long size;
  unsigned long VmyADC[34];
  char pad0[2];
  unsigned short eventnumber;
};
#ifdef FRONTEND_PROCESSOR
/*
* VME Variable Declarations
*/
struct runtime_data VMEmyADC_rt = { 5, 0xffffffff, -1, -1, 0, 0, 1, NULL};
char* VMEmyADC_lamclear_params = NULL;
char* VMEmyADC_lamdisable_params = NULL;
char* VMEmyADC_lamenable_params = NULL;
char* VMEmyADC_read_sort_params = NULL;
char* VMEmyADC_clear_params = NULL;
char* VMEmyADC_init_params = NULL;
char* VMEmyADC_thresholds_params = NULL;
char* VMEmyADC_compressionfilter_params = NULL;
char* VMEmyADC_blockwrite_params = NULL;
char* VMEmyADC_blockread_params = NULL;
/*
* Event Function Prototypes
*/
static void EReadADC(void);
/*
* Maximum event size
*/
unsigned int MaxEventSize = 144;
#endif /* FRONTEND_PROCESSOR */
```

```
/* myExperiment.frontend.c */
#include <stdlib.h> /* for malloc & free */
#include <string.h>
#include <linac_rtems.h>
#include <sis1100_api.h>
#include <lucidformat.h>
#include <vmereader.h>
#include <vmecomm.h>
#include <vme.h>
#include <functions.h>
#define FRONTEND_PROCESSOR
#include "myExperiment.frontend.h"
/*
* Data buffer management
*/
extern char *Dataptr;
extern char *DataRecordLimit;
/* File descriptor used to access sis1100 */
extern int devfd;
extern rtems_id AcquisitionTaskId;
extern unsigned int VmeBaseAddr_A24;
extern unsigned int VmeBaseAddr_CSR;
extern unsigned int VmeBaseAddr_A32;
/*
* This is code for
* the 'ReadADC' (0) event.
*/
void
EReadADC (void)
{
  struct EReadADC *dp = (struct EReadADC *)Dataptr;
  dp->size = sizeof *dp - sizeof dp->size;
  caen_blockread(dp->VmyADC,sizeof(dp->VmyADC),NULL);
  dp->eventnumber = NReadADC;
  Dataptr += sizeof *dp;
  if (Dataptr > DataRecordLimit)
    SendDataRecord();
}
/*
* LUCID timed event information
*/
static struct Timer TimerTable[] = {
  { EReadADC, 100000UL, NULL },
};
struct Timer *TimerTableBase = TimerTable;
```

```
/*
* No CAMAC LAM's used in this experiment.
*/
/*
* No interrupts are used in experiment.
*/
unsigned int vme_irq_mask = 0x0;
/*
* LUCID Initialization
*/
void
InitializeLucid (void)
{
  /* Cause a VME hardware reset.  */
  if (vmesysreset(devfd) < 0)
    LogFatal("SIS1100:  VME system reset failed.\n");
  caen_v792_init(NULL,0, &VMEmyADC_rt,VMEmyADC_init_params);
}
/*
* Routine to cause a
* Begin Run trigger.
*/
void
StartRun (void)
{
}
/*
* Routine to cause a
* End Run trigger.
*/
void
StopRun (void)
{
}
/*
* Routine to cause a
* Resume Run trigger.
*/
void
ResumeRun (void)
{
}
/*
* Routine to cause a
* Suspend Run trigger.
```

```
*/
void
SuspendRun (void)
{
}
/*
* Events which are to be performed upon user command.
*/
void
UserCommand (int code)
{
/*
* No user defined commands.
*/
}
```

APPENDIX C
DESIGN PATTERNS

A *design pattern* is a description of communicating objects and classes that are customized to solve a general design problem in a particular context [47]. Design patterns are described by a cooperative group of software objects, their structure and behavior, and their communication protocol. The roles and interaction of these objects form a *collaboration*, realizing a solution in a specific problem domain. To contrast the notion of a design pattern with that of an algorithm, note that an algorithm is a solution to a *computational* problem, whereas a design pattern is a solution to a *software engineering*, or design problem [15].

A design pattern is comprised of four important aspects [47]:

1. *Pattern name* - The importance of creating and applying descriptive, meaningful names to software features cannot be overstated. Creating a common vocabulary permits the succinct exchange of abstract concepts amongst designers, developers, and users.

2. *Problem description* - This is also known as the *problem-space*, and it represents the domain of the design pattern's applicability. The problem description may include context and pre-conditions required in order to implement a pattern.

3. *Solution* - This describes the components of the design, their communication, responsibilities, and collaboration roles. The pattern itself is not a solution, but a template from which a solution may be generated: a pattern provides a description of a design problem and how an abstract collaboration of software elements solves it.

4. *Consequences* - the pros & cons of the design pattern's application to the problem-space. As optimization and design always involves some form of compromise, the suitability of a particular design pattern must necessarily be weighted by consideration of the advantages and disadvantages afforded by its use.

Utilizing these points, the following discussion will illustrate the details of two design patterns which play important role's within the Lucid data acquisition system.

Note that in the following discussion, either, or both of two paradigms may be at work to distribute data from source to destination:

1. *The "push" model* - data is forced upon, or *pushed*, from the source to destination, upon state change. This model combines control and data flow: the source of the data determines when and to whom it is delivered.

2. *The "pull" model* - in this paradigm, clients are responsible for extracting, or *pulling* data from the source for their own use. This model decouples data flow from the flow of control, as the data source bears no responsibility for its distribution.

C.1 Observer Pattern

Also known as the *publish-subscribe* pattern, the Observer design pattern targets the problem of disseminating information from a single source, or server, to multiple clients. In other words, this pattern describes a form of *demultiplexer*, providing a subscription mechanism whereby clients register their intent to be notified upon the occurrence of some event of interest.

The structure and behavior of the collaborating roles involved in the realization of the Observer pattern are shown in Figure C.1.

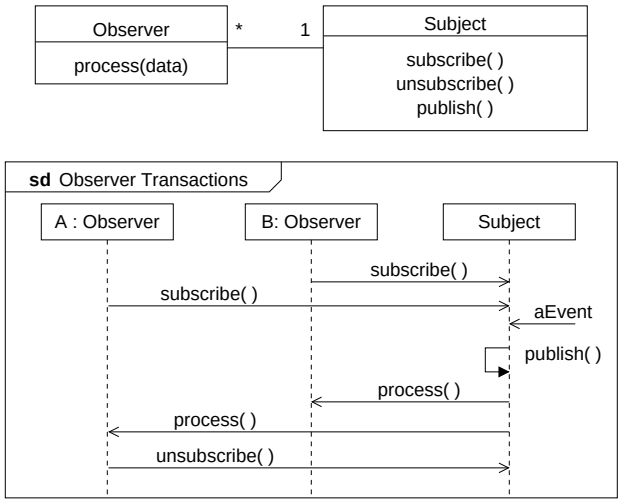

Figure C.1: Structure and behavior of objects participating in an Observer pattern: (top) collaborating classes, (bottom) sequence diagram of collaborator transactions.

The key aspects enumerated in the previous section are provided next, outlining the problems addressed, the solution proposed, and the consequences associated with an implementation of the Observer pattern.

C.1.1 Problem Description

An unknown number of clients require timely notification upon the occurrence of some event. These events of interest might typically occur according to some well-defined policy, for example periodically, asynchronously, or upon a change in the value of some data. Furthermore, clients will generally execute some actions in response to the event.

C.1.2 Solution Description

Clients (observers) are dynamically coupled to the event publisher (subject) via a *subscription* mechanism. This mechanism is implementation specific, but may be as simple having the subject add its subscribers to a list.

When the event of interest occurs, the subject notifies the observers via a *publication* mechanism. Again, this is implementation specific, but if the subject maintains a simple

list of observers, as mentioned above, the publication operation may be as simple as walking the list and invoking each observer's `process( )` method. The `process( )` method is more commonly known as a *callback*.

C.1.3 Consequences

The Observer pattern simplifies the administrative task of distributing event notification from a single source to multiple interested parties. The dynamic subscription policy permits runtime flexibility, as the subject need not concern itself with the identity of its clients. Another feature of this design is that information flow is uni-directional, from subject to client only. Also, centralization of event publication from a single entity avoids replicating the event delivery machinery in each client.

However, as the subject has no control over the content of client callback methods, any client may cause excessive delay of subsequent event publication if its callback requires a lengthy processing duration. This concern becomes of critical importance when the Observer pattern is utilized in a real-time/reactive system, where excessive delay may lead to system failure.

C.2 Proxy Pattern

The Proxy design pattern is very closely related to the Observer pattern, with the major difference being that the former decouples the subject from its clientele by means of a surrogate object, the Proxy. This agent hides the implementation details of the subject from its clients, and vice versa. The subject publishes data to a proxy, where it is then distributed to the interested parties.

Figure C.2 shows the objects and their roles within the collaboration of elements forming a Proxy design pattern. The `ServerSideProxy` and `ClientSideProxy` objects both inherit from the abstract base class, `BaseProxy`. The UML notation for indicating inheritance is the closed-arrowhead, which points to the "parent" class.

C.2.1 Problem Description

One or more clients or consumers require timely notification of events of interest to them. For example, these events may be the availability of new data for processing, or some periodic occurrence. Note, this is the same problem addressed by the Observer pattern. However, the Proxy pattern explicitly addresses the situation where clients may be expected to subscribe to both local and remote information publishers. In this case, the use of a mediating agent, or proxy, permits transparent separation of client and server, shrouding the implementation details of one from the other.

C.2.2 Solution Description

By routing observer-subject messages through a third-party agent, the communication end-points need not know the implementation details of one another. This allows

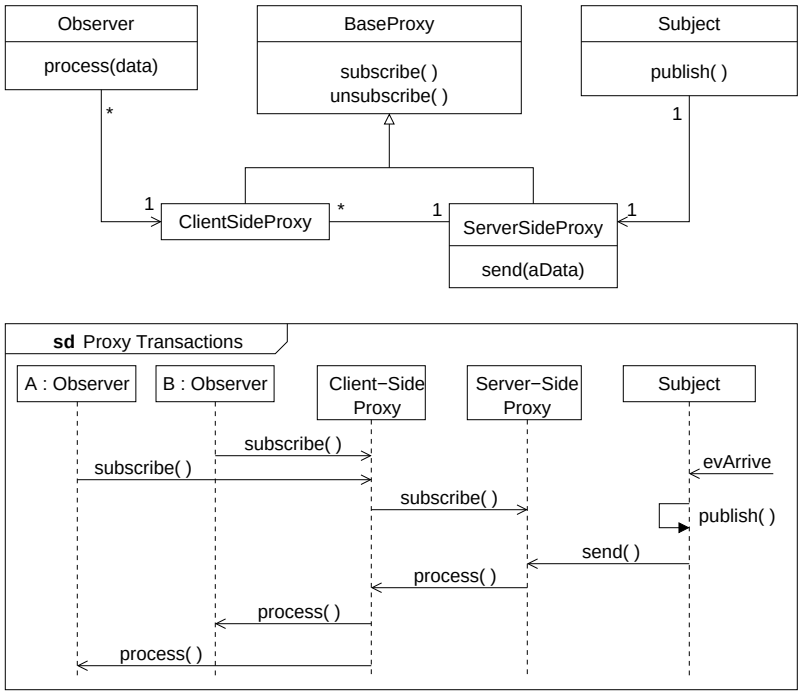

Figure C.2: Proxy design pattern.

`ClientSideProxy` and `ServerSideProxy` to occupy different address spaces, or even be physically remote from one another: communication details are encapsulated in the proxy objects, as implicit in the structural component of Figure C.2.

The largely one-way information flow from subject to observer is reflected in terms of the instance multiplicity evident in this figure: the Proxy pattern requires one `ClientSideProxy` instance per published event, per address space, and one instance each of the `ServerSideProxy` and `Subject`.

C.2.3 Consequences

Isolating the Client-Server communication details in broker objects simplifies client construction, as the same source code may be used whether the server is local or remote; only changes to the communication mechanisms of the client-side and server-side proxies are required.

Inter-address space information flow is minimized to a large extent through employment of this pattern, as traffic flows only on event publication, and then only from `ServerSideProxy` to `ClientSideProxy`; all other communication channels are local to that particular address space (i.e. server-side, or client-side).

APPENDIX D
LUCID'S CODE GENERATION SUBSYSTEM

The data acquisition behavior of the I/O Manager is determined by user-provided instructions specified in a high-level programming language known as the Experiment Description Language (EDL). Once users have provided the instructions they wish the I/O Manager to perform in the form of an EDL text-file, Lucid's code generation machinery is then invoked using the *build-dialog* of the *gxluxid* user interface (see Figure 1.6). The product of this process is an executable image, which is loaded onto and run by the IOM.

However, the generation of an executable image is only the final step in the sequence of operations required to produce a machine-executable program. This series of operations is known as *compiling*, and the software component responsible for these operations is known as a *compiler*.

Compilers are designed to construct executable images for a specific environment, as defined by a particular operating system, hardware architecture pair. They also constrained in the sense that compilers target specific programming languages, such as C or C++. Thus, a compiler must dissect its input to determine the grammatical validity, according to the formal specifications of a particular language.

Beginning with an examination of two categories of grammar frequently encountered in compiler technology, the following discussion presents an overview of a generic compilation process, concluding with a study of tools commonly used in compiler construction.

D.1 Backus-Naur Form Grammar

Written languages, programming languages included, are expressed using a finite set of symbols, or alphabet. The categories of permissible symbol arrangements (words) define the *syntax* of a language, while the set of allowable syntactic sequences (sentences) are governed by the *grammar* of that language.

A Backus-Naur form (BNF) grammar is composed of a set of *productions*, or BNF-rules, defining a programming language. These provide mathematical rules defining the set of character strings permissible within a target language. Perhaps most importantly, BNF-rules introduce a notation for production specification. As an example of this notation, consider the production:

$$< letter > \rightarrow A \mid B \mid C \tag{D.1}$$

This rule defines a language composed of the single characters, 'A', 'B', or 'C', and is read as, "A *letter* is either an *'A*', *'B*', or a *'C'*. In this context, the term *letter* is known as a *syntactic category*, or *non-terminal*, and constitutes a label for the simple language defined by the above rule; non-terminals may be decomposed into more fundamental structures. The characters to the right of the $\rightarrow$ symbol are known as *terminal* symbols, and may not be decomposed further: they are the fundamental syntactic elements of a language. Note, the $\mid$ symbol denotes alternatives, as in a logical OR operation, and in some instances may

be indicated with a $\vee$ symbol, the alternation operator.

A non-terminal defined by a BNF-rule may itself be recursively used in that rule, thus implying repetition. For example:

$$< word > \rightarrow < letter >< letter > | < word > \tag{D.2}$$

Using the preceding concepts, a programming language may be formally specified as the set of syntactic categories (non-terminals) and the terminal symbols of which they're composed [48]. A complete BNF grammar is a set of productions defining a hierarchy of sub-languages, leading to a root-level non-terminal. In the case of a programming language, that root-level syntactic category is generally <*program*>, which is the entire set of characters comprising a source-code file written in the target language.

The hierarchical decomposition of a source-code file, from the root non-terminal, down through the fundamental terminal symbols, implies that a syntactic analysis of it will produce a *tree structure* because of the structure imposed by the BNF-productions. Within this *parse-tree*, the root-node is the non-terminal designating the entire language (e.g. <program>), each branch is a non-terminal designating the content of the sub-tree below it, and each leaf is a terminal lexical item.

The utility of BNF grammars lie in their simplicity. However, this simplicity requires that common syntactic constructs such as optional, alternative, and repeated elements are specified using an inelegant representation. Extensions to BNF provide notation to describe these common elements, including: 1) optional elements are indicated by enclosing them in square brackets, $[\cdots]$, 2) alternatives are indicated with the '|' symbol as previously discussed, and 3) arbitrary sequences of element instances may be indicated by enclosing the element in braces, followed by an asterisk, $\{\cdots\}*$.

Backus-Naur form grammars are not without their shortcomings. Contextual dependencies are one aspect of syntax that cannot be addressed by a BNF production. For example, the restriction common to many programming languages that "the same variable identifier may be declared only once in the same block", cannot be expressed using a BNF production.

D.2 Regular Grammars and Regular Expressions

Regular grammars are a special case of a BNF-grammar, and have productions of the form:

$$< non-terminal > \rightarrow < terminal >< non-terminal > | < terminal > \tag{D.3}$$

Also equivalent to regular grammars are *regular expressions*. These are defined recursively as follows [48]:

1. Individual terminal symbols are regular expressions.
2. If α and β are regular expressions, then so are $\alpha \vee \beta$, $\alpha\beta$, (α), and $\alpha*$.
3. Nothing else is a regular expression.

where $\alpha \vee \beta$ represents the alternatives, α or β, $\alpha\beta$ is the concatenation of the regular expressions α and β, and $\alpha*$ signifies zero or more repetitions of α, also known as the Kleene closure of α .

D.3 Compiler Processing Overview

The process of translating source code into an executable program may be logically decomposed into two activities: 1) *analysis* of the input source-file, and 2) *synthesis* of an executable, or object program. The UML *activity diagram* of Figure D.1 depicts the source-code translation process of a typical compiler.

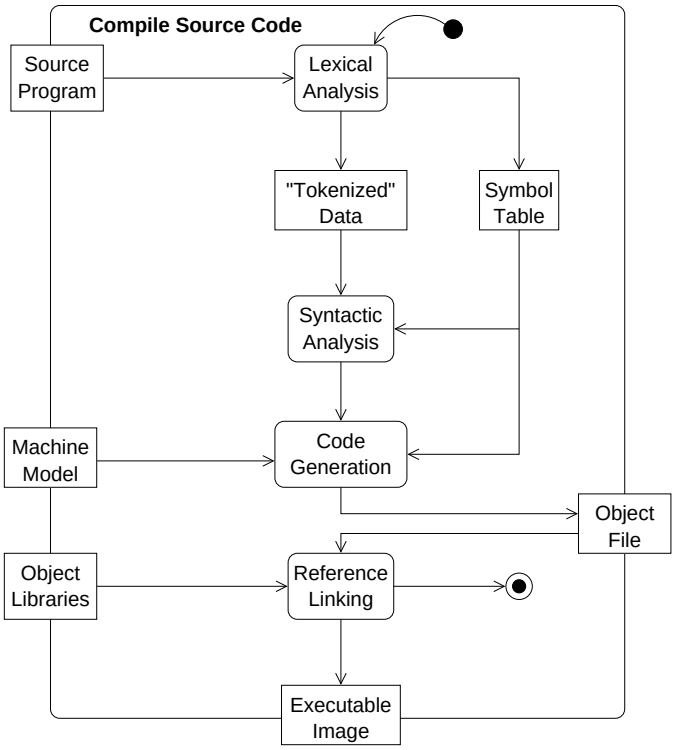

Figure D.1: Activity diagram of a generalized source-code compiler.

Activity diagrams may be thought of as the UML solution to the ubiquitous flowchart, and are typically employed to model the progression of actions performed by software objects: i.e. they express algorithmic activities. Boxes are used to denote *objects* (i.e. the typical UML class/object figure), while *activities*, or actions are indicated by round-cornered boxes. Thus, in Figure D.1, the `Source Program` node serves as an *input parameter* to the `Lexical Analysis` activity, while the `Executable Image` and/or `Object File` serve as output nodes. The small, solid dot at the top of the diagram is a *pseudo-state*, denoting the point of initiation for the activity flow. Similarly, the encircled dot near the figure's bottom is a *terminal* pseudo-state, indicating the activity's conclusion.

D.3.1 Source-Code Analysis

Although programmers typically structure their source-code documents according to the logical grouping suggested by the data structures and operations acting on those structures, compilers know nothing of this artificial structure and must therefore decompose and examine the source-file, symbol-by-symbol, in order to process it. Source-code analysis typically involves these phases [48]:

1. *Lexical Analysis* - the first step in translation is to examine the sequence of symbols comprising the source-file, and identify those symbols which form the elementary pieces of the programming language. For example, these elementary symbols may include keywords, numbers, mathematical operators, expression delimiters, or comments. This process is performed by a *lexical analyzer*, or *scanner*, which generates *tokens*, or *lexical items*, as it discovers them (see Figure D.1). It is the duty of the scanner to identify each lexical item it encounters, and then to attach a type-identity tag to those tokens. This tag is used in subsequent phases of the translation process.

2. *Syntactic Analysis* - utilizing the lexical items and their tags produced in the previous stage by the scanner, the function of a *syntax analyzer*, or *parser*, is to identify the larger-scale features of a program's structure: statements, expressions, declarations, etc. Syntactic analysis usually proceeds in concert with *semantic analysis*: only after the syntactic analyzer identifies a sequence of lexical items forming a syntactic unit (e.g. an expression, or function-call) may a semantic analyzer process the result and determine if the unit is correct in the current context. Communication between syntactic and semantic analyzers is commonly via a *stack* data structure: the syntactic analyzer *pushes* lexical items onto the stack as it encounters and processes them, while the semantic analyzer *pops* them off of the stack as they are processed and found to be correct.

3. *Semantic Analysis* - this procedural element forms the link between the analysis and synthesis stages of source-code translation. The processing performed at this stage is dependent upon language and application specifics, but may include activities such as:

   (a) *Symbol-Table Maintenance* - a symbol-table is a crucial data structure in all compiler software [49]. Typically, it contains an entry for each unique identifier obtained from the lexical and syntactic analysis of the source program. Beyond just the identifier itself, the symbol-table may retain information on identifier type (variable, array name, etc.), variable type (integer, double, string, etc.), identifier value, and other data used in subsequent phases of source-code translation.

   (b) *Insertion of Implicit Information* - information in the source-file that may be unspecified by a programmer must be made explicit in the low-level object program. For example, some C-language compilers permit the return-type of a function to be left unspecified, implying a return-type of `int` as the compiler-supplied default.

(c) *Error Detection* - syntactic and semantic analyzers must be made to respond correctly to improperly formed programs, in addition to properly-structured source files. For example, a lexical item that is incorrect for the context in which it appears must be identified as an error, and the compiler's user notified with a meaningful message containing the location and symptom of the error.

(d) *Compile-Time Operations* - although not a feature of all programming languages, compile-time operations, such as macro expansion and conditional-compilation, are usually processed during the semantic analysis phase of program translation.

D.3.2 Executable Object Synthesis

The second major phase of source-code compilation is the synthesis of an executable program, or image. Utilizing the output produced by the semantic analyzer, this phase includes the operations of code generation and, optionally, code optimization. Also, depending on whether or not sub-programs are translated separately, or if references to previously compiled libraries of executable code are present, a final linking and loading stage may be required to complete the program's transformation. Each of these phases are examined in the following:

1. *Optimization* - the output of a semantic analyzer typically a representation of the executable image in some *intermediate* format. Prior to supplying this intermediate form to the code-generation phase of the translator, optimization may be performed to enhance either execution speed or memory requirements, or both. However, even if optimization capabilities are present within a compiler, this phase of translation is almost always optional, being activated only at the user's discretion.

2. *Code Generation* - following semantic analysis and the optional optimization phase, the next step in the translation process is code synthesis, or code generation. As the name suggests, it is at this stage that properly formatted code is output, based on the information produced from all previous operations of the compilation process. The code generated may be assembly-language statements, machine-code, or some other format suitable for execution on a CPU, or for further processing. The product of this stage is known as an *object file*.

3. *Linking and Loading* - if sub-programs were separately compiled, or there are unresolved references to code contained in external libraries, these issues are resolved at this stage of the translation process. Individual sub-programs are coalesced and, in the case of *statically linked* images, externally-referenced executable code is located and bound into the final, executable product. Another form external-reference linking may be used to produce a *dynamically linked* image, where references are resolved at the program's run-time. However, this type of image will not be covered here.

During these phases the compiler will consult its *machine model*: an internally maintained software representation of the target hardware architecture it is generating instructions for.

This model may be thought of as a virtual image of the target CPU, permitting the compiler to track the occupation of registers and other resource usage. In this way, the compiler is able to monitor and mitigate resource conflicts, as well as optimize the generated object code for the target architecture.

D.4 Compiler Tooling

Compiler designers recognized that several stages in the translation process could benefit from the development of automated tools to handle aspects that were likely to be found in any compiler suite. Lex, a lexical analyzer generator, and Yacc, a parser generator, are two popular tools developed for compiler construction.

Originally developed for use on a commercial operating system, the utility and popularity of these software tools drove software developers to create open-source versions, known as Flex and Bison, as alternatives to Lex and Yacc, respectively.

D.4.1 Flex

The role of a lexical analyzer is to examine source-code, producing lexical items (tokens) based on syntactic structures recognized in the input stream. Input to the Flex program consists of a user-specified configuration file containing *regular expressions* which define the basic syntactic items of a language, as well as *actions* to be taken upon recognition of those syntactic items in the input content [50].

Upon activation, Flex will utilize the configuration file as an instruction set to produce a lexical analyzer, or scanner. Configuration files are composed of three sections, delimited by %% symbols:

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Of these sections, only the `rules` section is not optional. Thus, a simple example of a lexical analyzer for identifying words and numbers has the configuration file of the form:

```
%{ #include <stdio.h> %}
%%
[0-9]+                  printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]*    printf("WORD\n");
%%
```

Fed with this configuration input, Flex will produce a lexical analyzer, which will emit the string, "NUMBER", whenever it encounters one or more of the characters, "0" through "9", and "WORD" whenever it encounters a letter followed by zero or more letters *or* digits.

In practice, the output actions specified above would typically be configured to emit a *token* upon encountering a regular expression match, instead of merely printing a string. Tokens identify syntactic items, and must be produced when Flex is utilized with its companion syntax analyzer generator, Bison.

D.4.2 Bison

From Figure D.1, the phase of translation immediately succeeding lexical analysis is that of parsing, or syntactic analysis. Bison is an open-source syntax analyzer generator, and is designed to be used in concert with Flex such that together, the two components form the source-code analysis engine for a compiler [51].

Provided with input in the form of a configuration file defining a set of grammar rules, Bison will produce a parser program, suitable for inclusion as a component in a compiler application. The Bison input configuration file is of a similar structure to that used by Flex, consisting of *definitions, rules,* and *subroutine* sections. In this case, the *rules* section defines BNF grammar rules and the actions to be performed when source input matches those rules. For example, the BNF grammar rule

$$< sum > \rightarrow < number > + < number > \tag{D.4}$$

may be expressed in Bison as:

```
sum: NUMBER PLUS NUMBER
    { printf("The sum is %d\n", $1+$3); };
```

This parser would require a Flex scanner configured to emit tokens upon recognizing the NUMBER and PLUS terminal syntactic elements:

```
%%
[0-9]+              yylval=atoi(yytext); return NUMBER;
\+                  return PLUS;
%%
```

In the Bison configuration fragment above, the $1 and $3 notation is used to extract the numeric values of the first and third arguments, respectively (i.e. *NUMBER* and *NUMBER*). Given an input string of the form, "$2+2$", the parser's output would be, "The sum is 4".

Section D.5 will examine the roles of Flex and Bison within Lucid's code generation software component.

D.5 Lucid's Code Generation Component

This section will examine Lucid's code generation system. In particular, it will present the work performed to augment that system with additional functionality including support for generic VME-based interrupts, block transfers, CBLT and MCST operations, as

well as automating instrumentation resource management. This work is built on the core infrastructure described in [52].

Lucid's code generation component is a type of *compiler*: using the source-code analysis tools described in Section D.4, this compiler parses Experiment Description files (EDF), generating source-code in the C programming language. In turn, this C-code is translated to machine executable format using the open-source GNU C-compiler suite, *gcc*.

Similarly, Module Description files (MDF) are also syntactically analyzed as part of this process. These files are persistent-storage devices, containing operational attributes of the various CAMAC and VME modules used in concert with the Lucid DAQ. MDF entries comprise the library of instrumentation modules that Lucid knows how to communicate with.

The software artifact responsible for generating C-code based on the information contained in Module and Experiment Description files is known as the `VmeBuildReader` (see deployment diagram, Figure 3.2). `VmeBuildReader` is responsible for generating two, cooperative executables:

1. *Server-Side DAQ Application* - this image is formed when code generated by compiling a user's EDF is linked with the `IOMReader` library, *libIOMReader.a*. The object module thus produced, *aExperiment.pc386* in Figure D.2, is then downloaded to and executed by the I/O Manager. This module forms the `Subject` of a *Proxy* design pattern, publishing event-types specified by `Trigger` definitions and executing client callbacks given by `Event` object definitions, as specified within the EDF. The `IOMReader` is detailed in Section 3.7, where the proxy collaboration members are depicted in Figure 3.7.

2. *Client-Side DAQ Application* - this executable forms the client-side proxy object of a Proxy design pattern. In that role, this component maintains direct communication with the IOMReader, and is responsible for placing data acquired by the I/O Manager into a shared-memory buffer, then signaling data consumers when that data is ready for their retrieval. Additional details of this application will not be provided here, but they may be found in [53].

Figure D.2 is an activity diagram illustrating `VmeBuildReader`'s production of a server-side data acquisition application, *aExperiment.pc386*, from the user-provided Experiment Description file, *aExperiment.r*, and the instrumentation equipment Module Description files, *vme.modules* and *camac.modules*. As the complement to this diagrammatic representation of software *behavior*, Figure D.3 reveals the aggregate *structure* of `VmeBuildReader`, exposing important internal objects and their relationships.

In addition to the UML activity diagram features already discussed (see Section D), Figure D.2 introduces two new elements, *forks* and *joins*, indicated by the thick bars. A fork represents a division of activity flow, whereas a join indicates a coalescence of activities. In both cases, the actions on the outgoing-side may not commence until *all* input activities have arrived at the fork or join.

From Figure D.2, note that the final products produced by `VmeBuildReader` are the C-language header and source files, *aExperiment.frontend.h* and *aExperiment.frontend.c*.

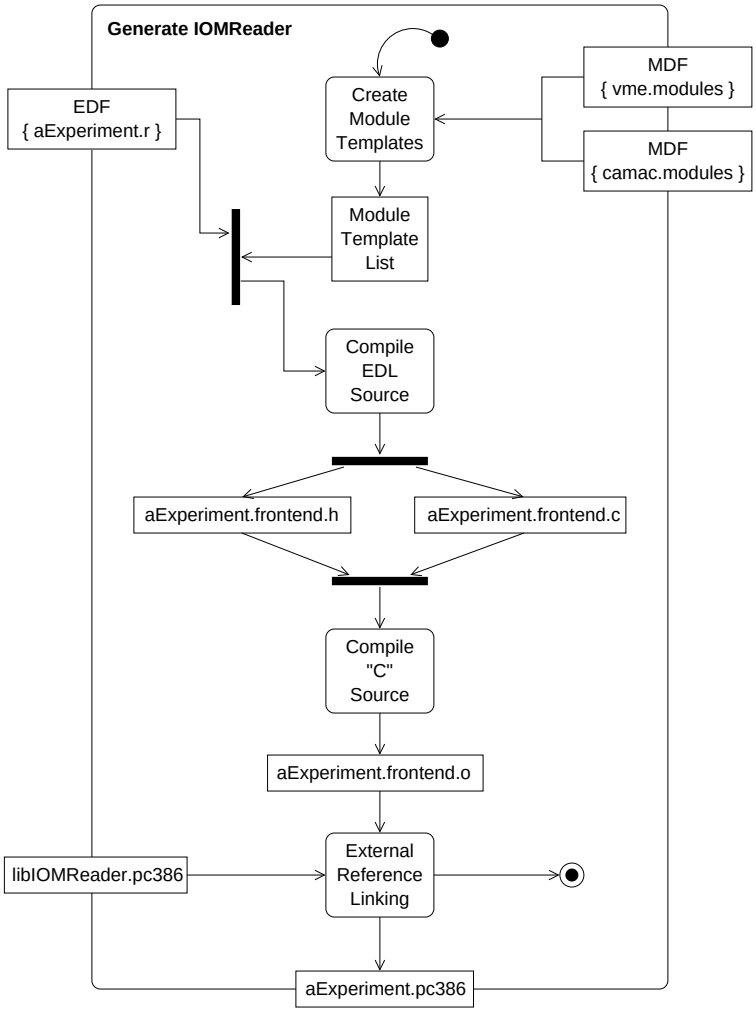

Figure D.2: Activity diagram depicting the creation of an `IOMReader` application from a user's Experiment Description file, *aExperiment.r*.

The ".frontend" suffix implies these files are destined for the I/O Manager. Conversely, a lack of such suffix denotes files that will form an executable for use on the host workstation. After those files are created, standard compiler tools (gcc *et al*) are used to form the image, *aExperiment.pc386*.

D.5.1 An Example EDF: *aExperiment.r*

Consider the contents of a sample Experiment Description file, *aExperiment.r* :

```
#Module definitions
define Adc1 "792" S(5) lam(96) level(7)
define Adc2 "792" S(6)
define Adc3 "792" S(7)
define theChain chain "Adc1,Adc2,Adc3"
```

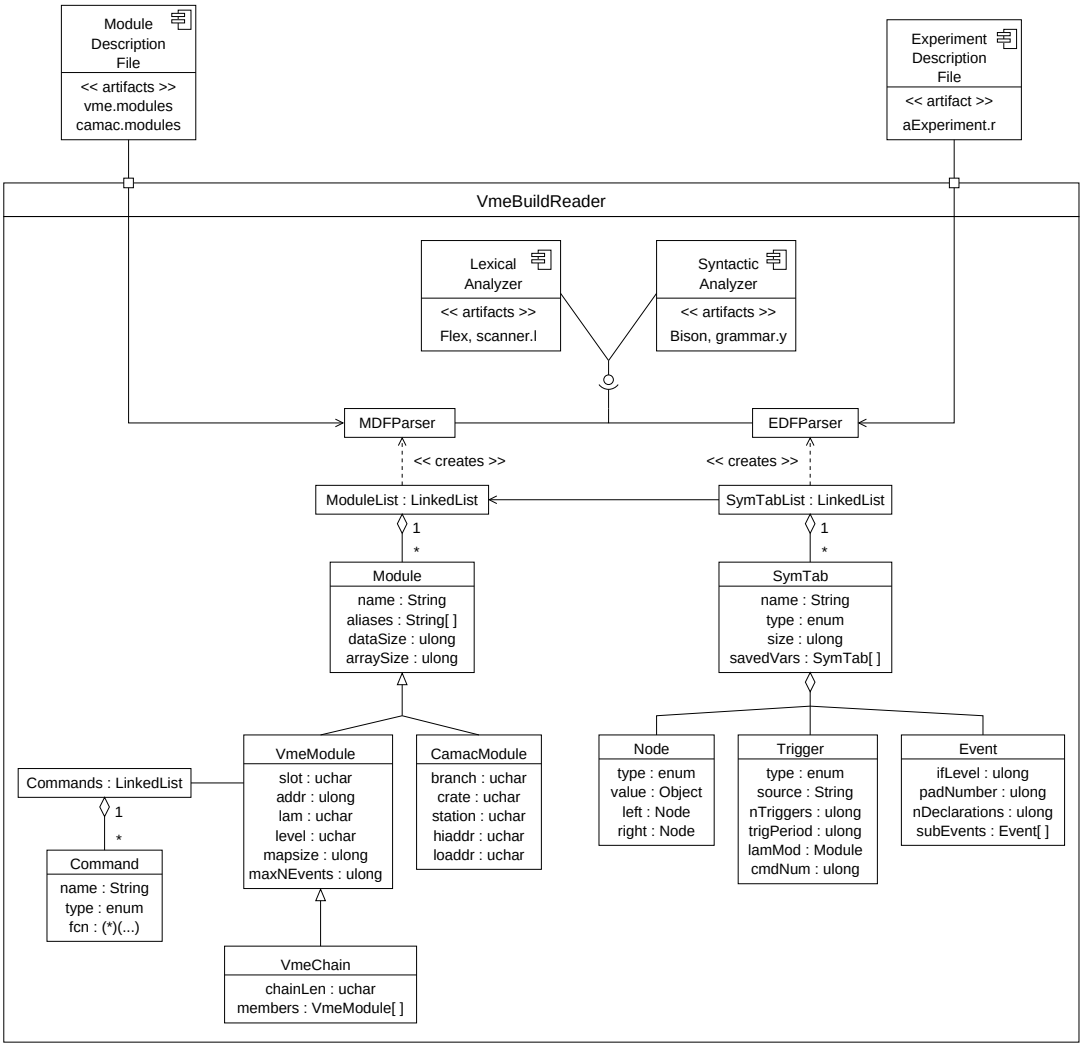

Figure D.3: Structured class diagram of the `VmeBuildReader` artifact.

```
#Trigger definitions
trigger readChain every Adc1 lam

#Event definitions
event readChain:
      blockread and save theChain
```

The `Module`, `Trigger`, and `Event` definitions of this example will be used in the remaining discussion to illustrate the processes by which the `VmeBuildReader` program generates the server-side data acquisition application of Figure D.2, *aExperiment.pc386*.

Three VME modules of type "792" are defined above, corresponding to CAEN v792 charge-integrating, analog to digital converters (ADC). As implied in Figure D.2, the attributes of this type of instrumentation module are provided in the MDF file, "*vme.modules*". The geographic, or "slot" position of each module within the VME crate is given using the *S(x)* notation, while module *Adc1* is to be configured to produce an IRQ vector of 96 at the

interrupt-level of 7 using the *lam(96)* and *level(7)* syntax. Referring to an interrupt as a LAM (Look-at-Me) is a legacy notation retained from the period when CAMAC modules were the sole source of instrumentation-based interrupts within the Lucid DAQ.

The final definition, above, uses the `chain` syntactic element, introduced as part of the work of this thesis. This terminal lexical item permits experimenters to take advantage of VME chained-block transfers (CBLT) and multi-cast writes (MCST), as discussed in the previous chapter (see Section 2.2.4.6). Here, `theChain` is a `VmeChain` object, consisting of all three of the ADC modules previously defined. From the inheritance hierarchy portrayed in Figure D.3, note that `VmeChain` objects are a type of `VmeModule`, which in turn are a type of `Module`.

The lone `Trigger` statement specifies that the actions defined within the `Event`, `readChain`, will occur with each interrupt raised by the `Adc1` module. The `blockread` terminal syntactic item, also introduced via the work of this thesis, will cause code to be generated which will perform a block transfer of data collected by members of `theChain`. This data will then be saved and transmitted from the IOM to the experimenter's workstation, according to the policy discussed in Section 3.7.

D.5.2 The MDF Compiler

As illustrated in Figure D.2, the first activity performed by `VmeBuildReader` is a syntactic analysis of the information contained in those files comprising Lucid's module description database, *vme.modules* and *camac.modules*. The result of this analysis is the creation of the `ModuleList` object, depicted in Figure D.3. This object is a *linked-list* of all the `Module` types described by the module definition databases, and serves as a module-template cache, consulted by `VmeBuildReader` as it encounters module definition statements in the EDF, *aExperiment.r*.

Module Description files are simply plain-text documents describing the operational attributes of instrumentation modules in a machine-parsable format. That is, the language used to describe modules was designed to permit lexical and syntactic analyses using Lex and Yacc.

Maintaining module information centrally, and expressing it in a machine-parsable format, relieves Lucid's users from having to know the operational details of the various instrumentation devices used in experiments. At the same time, this technique affords the ability to allocate instrumentation resources, such as module address ranges and interrupt vectors, with a minimum of user intervention. The logic utilized by VmeBuildReader to provide this autonomy is further discussed in Section D.5.3.

As an example of the content and structure within a module description file, consider the following excerpt from the VME module description file, *vme.modules*, typically created and maintained by Lucid's administrator:

```
module "CAEN 792 32-channel charge ADC"
       "792"
       "caen792"
    readsize is 34 by 32 by 1
    mapsize is 65536
```

```
geoaddr is 0x1002 A24 D16 rw geo
statusregister is 0x100e A24 D16 ro geo
databuffer is 0x0000 A24 D32 ro
blockread is function caen_v792_blockread
lamenable is function caen_v792_lam_enable
```

M*odule*, *mapsize*, and *function* are examples of terminal syntactic elements, or *keywords*, defined using regular-expressions within the Flex input specification file, *vmescanner.l*. The *module* keyword denotes the start of a module definition, followed immediately by quoted strings representing formal names and aliases, by which a module of this type may be referenced in user's experiment description file. Following the alias section are the *readsize* and *mapsize* keywords, denoting data storage and VME bus address range requirements, respectively.

Geoaddr, *statusregister*, and *databuffer* are descriptive labels for module registers housing the functionality suggested by their names. From left to right, the register's offset from the module's base-address (0x1002), address space (A24), data-width (D16), and access permissions (read/write (rw)) are all provided to fully describe where and how to access the register's content.

The final two entries in the above excerpt indicate that the *blockread* and *lamenable* keywords are associated with the *caen_v792_blockread* and *caen_v792_lam_enable* procedures (functions), respectively. When these keywords are encountered in a user's Reader description file, Lucid's compiler subsystem will insert the appropriate routines in the generated code. Using this method, arbitrary software routines may be constructed for insertion into the code-stream to handle critical module operations requiring more sophistication than may be achieved using single-cycle VME access methods.

An excerpt of the production-definition section of the MDF parser, *vmegrammar.y*, is shown here:

```
modulefile: moduledefinitions;
moduledefinitions: /* empty rule */
          | moduledefinitions moduledefinition;
moduledefinition: module name aliases statements commands
          {  installmodule(mod, true);  } ;
module: MODULE
      { yyerrok; } ;
name: QSTRING
    { mod = newmodule($1); } ;
aliases: /* empty rule */
       | aliases QSTRING
       { mod->addAlias($2); } ;
statements: statement
          | statements statement;
statement: READSIZE IS NUMBER BY NUMBER BY NUMBER
         { mod->arraysize = $3*$7;
           mod->datasize = $5;
```

```
            mod->maxNEvents = $7;
          } ;
          | MAPSIZE IS NUMBER
          { mod->mapsize = $3; };
commands: command
        | commands command;
command: STRING IS FUNCTION STRING
       { makecommand(mod,3,$1,$4,0,0); } ;
```

Note the root, non-terminal element of the module-definition grammar, *modulefile*. The remaining productions serve to systematically and hierarchically identify the grammatical structure of the lexical items identified by the scanner.

As the tokenized input is matched against each *regular grammar* production, a `VmeModule` object is created and populated according to the actions taken when a match to the rule is encountered (see Figure D.3). A description of the key steps enciphered in the above grammatical productions is list here:

1. The non-terminal, *moduledefinition*, allocates a new `VmeModule` object and set its canonical name to the initial quoted string specified in the MDF. For example, "CAEN 792 32-channel charge ADC".

2. Upon encountering other quoted strings following the first, each is added to the `VmeModule`'s list of aliases. These are simply easy-to-use names with which an experimenter may use when assigning module-types to variables used in a Reader-Description file.

3. The *statements* non-terminal is decomposed and physical attributes of that module type are added to the `VmeModule` object, including the maximum amount of storage required when reading a module's data and the amount and type of VME address space consumed (i.e. its *mapsize*).

4. Finally, each *command* non-terminal element is resolved into its constituents, resulting in a `Command` object being created according to the variety found. Each `Command` object created is stored in a list associated with the parent `VmeModule` object, and specifies one of three types of commands: single-cycle VME read/write operations, bit set/clear operations, or procedural functions.

The final result of the lexical and syntactic analyses of the module description file is a list of `VmeModule` objects defining the identities, attributes, and commands available for use in Reader-description files (see Figure D.2). This list is consulted by a second source-code analyzer used within Lucid to produce executables for use on both workstations and the I/O Manager

D.5.3 The EDL Compiler

Following the creation of the `ModuleList` object, the second phase of EDF compiling proceeds with the production of C-code. The configuration for the lexical analyzer of

this phase is generated by Flex using the file, *scanner.l*. Upon examining the content of *aExperiment.r*, this scanner's primary job is to identify and generate *tokens* for the various syntactic elements of its input, including the approximately 60 keywords present within the Experiment Description language. Examples of these keywords include the non-terminal items *define, chain, blockread, trigger, event* and *lam*.

Upon recognizing these elements, the scanner will replace their occurrence in its output with *tokens*: unique representations of the syntactic elements. EDF variables are treated specially in that a symbol-table object, or `Stab` object is allocated for each variable defined by the scanner. See Figure D.3 for the UML depiction of an `Stab` object. These objects are critical elements within Lucid's compiler technology, constituting a record of variable identity, type, and storage requirements.

The EDL lexical analyzer is configured to associate each syntactic element it recognizes with one of four, elemental object types, three of which are indicated in Figure D.3:

1. *Stab* - a symbol-table object is allocated for every variable or module defined in a Reader file.

2. *Node* - represents character strings and expression trees.

3. *Trigger* - used to represent information related to trigger definitions. For example, the action (event) to be executed and the cause of the trigger (periodic, interrupt, etc.).

4. *Integer* - used to represent terminal lexical items such as *blockread, and, save, define,* etc.

As alluded to in the previous section, instrumentation software resources are allocated according to a *machine model*, internally maintained by `VmeBuildReader`. When the EDL compiler encounters a module definition such as, `define Adc2 "792" S(6)`, it records that module's base address. Equipped with this information and the range of addresses consumed by modules of that species (retrieved from the `ModuleList` template), VmeBuildReader is able to detect and alert users of address-space conflicts at compile-time, rather than at run-time. Using a similar methodology, interrupt level and vector conflicts are also averted.

Figure D.4 is a representation of the perspective of VME address space maintained and referenced by `VmeBuildReader`. Modules are allocated address ranges according to the strategy discussed here, an adaptation of ideas found in the literature [54].

The `A24` space of Figure D.4 contains the address range reserved for CAMAC modules. This range extends from 0x00800000 to 0x00BFFFFF, consuming approximately 4 MB of the 16 MB available in the `A24` plane. The remaining regions of this plane may be utilized by VME modules.

Wherever possible, use is made of the geographic addressing capability afforded by the VME64x specifications. The relevant address space is the 24-bit addressable `CSR/CR` space in Figure D.4. Geographic addressing simplifies the assignment of module base-addresses, as modules supporting this feature are able to autonomously discover and assign themselves an address based on their slot-position within the VME crate.

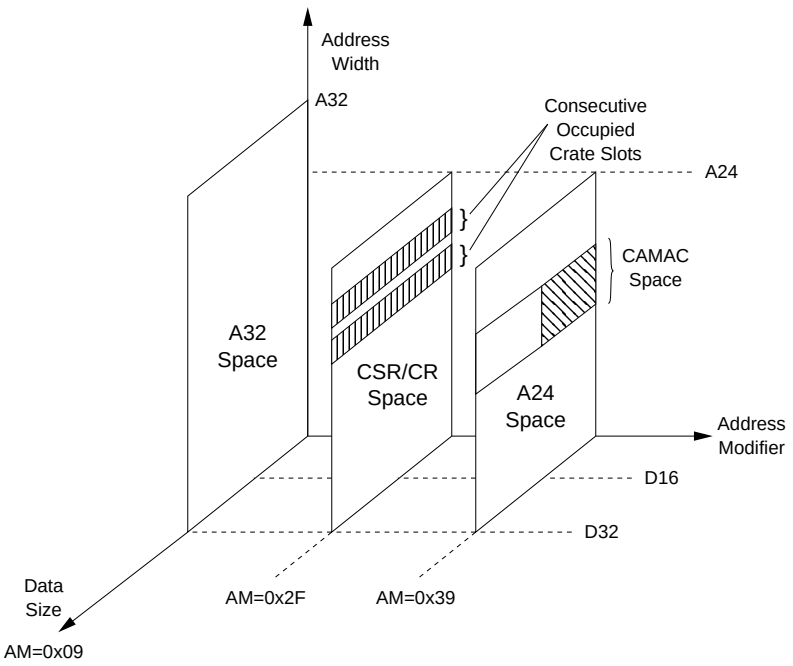

Figure D.4: The model of VME address spaces used by Lucid's code generation component.

In Lucid's present configuration, the `A32` space is occupied only by VME modules participating in CBLT and/or MCST operations. According to the VME64x standard, those types of transactions *must* occur in A32 space. From Section 2.2.4.6, a group of contiguously positioned VME modules may be configured to respond, *as a single unit*, to read/write requests targeted at a virtual base-address in `A32` space. A "chain-object" so configured behaves as though it were logically a single module.

`VmeBuildReader` will automatically generate a `VmeChain` object, its virtual address, and the initialization instructions required for the configuration of each `chain`-definition encountered in an EDF. Because `VmeChain` objects are a type of `Module` (see Figure D.3), actions such as reading and writing a `Module` also make contextual sense when applied to the former. Thus, no special syntax is required in `VmeBuildReader`'s treatment of activities targeting `VmeChain` objects.

APPENDIX E
MULTI-HIT MODE DAQ OPERATION

Recall that the data collection policy employed with the Lucid DAQ is "event-by-event": above-threshold detector pulses are digitized then extracted by the I/O Manager, readying the system for a new event. This algorithm advances sequentially; at no point are multiple events allowed to accumulate in the digitization hardware. The following discussion examines a parallelized data collection algorithm referred to here as *multi-hit mode*. Multi-hit mode is first explored from a queueing model approach, followed by a comparison of model predictions with experimental data.

E.1 Queueing Model

Consider the two-station, tandem queueing system portrayed in Figure E.1. The rate of input events is governed according to a negative exponential (Poisson) distribution, arriving at Station 1 with the average rate of λ. This station has a capacity of one, servicing each event at the average rate of μ_1, with the service times also following the Poisson distribution. Events arriving at Station 1 while it is occupied are lost (Erlang system).

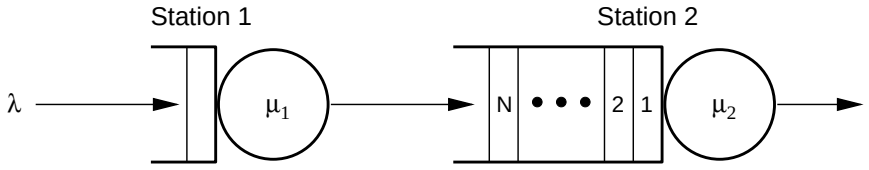

Figure E.1: Illustration of the 2-station, tandem queueing system.

After being serviced at the first station, events must proceed to Station 2 where they are serviced at an average rate of μ_2, also governed by a Poisson distribution. Station 2 has a finite queue capacity, N, where events may reside prior to their treatment at this station. If an event departing Station 1 encounters $N-1$ enqueued events at the second station, the tandem system will turn away further input until the situation is relieved. The particular policy invoked in this case is to block input from the system until the *entire* set of events has been serviced and left the system. The average service rate of this policy is μ_3, again following a negative exponential time distribution. The state diagram of this system is shown in Figure E.2.

Following the example presented in Section 5.6, one may obtain the steady-state *conservation of flow* equations for each state portrayed in Figure E.2:

$$\mathbf{0} = \boldsymbol{\pi}\mathbf{P} \tag{E.1}$$

where $\boldsymbol{\pi}$ is the vector of state probabilities and $\mathbf{P}$ is the matrix of state-transition probabilities, p_{ij}. This tandem system is also subject to the constraint that the state probabilities must sum to unity:

$$\mathbf{1} = \boldsymbol{\pi}\mathbf{e} \tag{E.2}$$

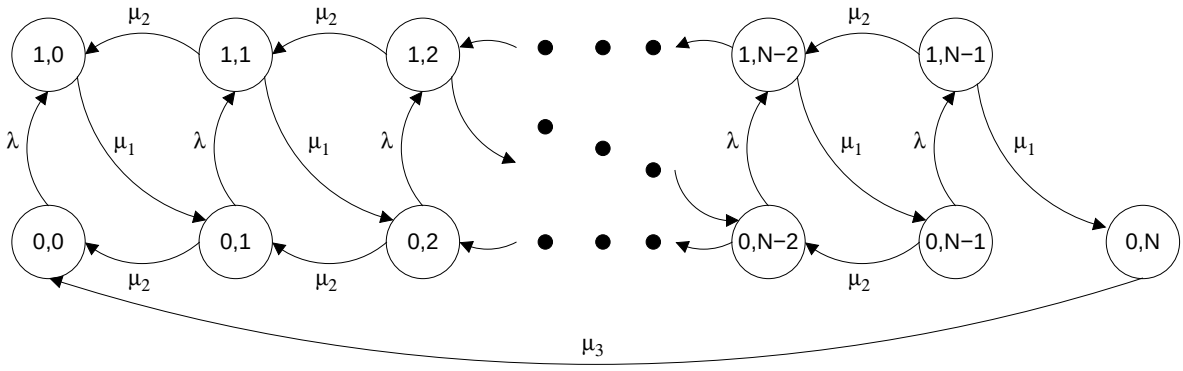

Figure E.2: State diagram of the tandem queueing system described here.

where **e** is the unit vector.

This system of linear equations lends itself to a solution suggested by a technique from the queueing theory literature [30]. From this solution one may extract the probability of input event loss, which is simply the sum of the state probabilities where events are prohibited from entering the system:

$$\sum_{j=0}^{N-1} \pi_{1,j} + \pi_{0,N} \tag{E.3}$$

E.2 Physical Model

The queueing model previously described was physically realized using standard components of the Lucid DAQ. Measurements made with this system provide a means of testing tandem model predictions against a physical implementation of that system.

Input events were provided in the form of photo-mulitiplier tube pulses generated by plastic-scintillator response to radiation products emanating from a point source (ruthenium-106). The pulse generation rate was controlled by varying the position of the radioactive source and the gain-voltage applied to the PMT. In this fashion, input event rates of up to approximately 45 kHz were generated. These signals correspond to the model parameter, λ .

The first station of the model corresponds to the digitization stage of CAEN VME modules (v792), arranged in a *chain* configuration. The duration of digitization is represented by the model parameter, μ_1^{-1} . The multi-event, dual-port FIFO buffers of the VME modules provide the model queue of *Station 2*, while the station's service was provided in the form of data read-out by the I/O Manager.

In the measuements described here, the "virtual module" of the VME chain object was comprised of three, v792 modules configured to digitize three channels of data per input trigger. The chain object was additionally configured to generate an interrupt upon the data FIFO becoming full (i.e. after 32 events). This condition is reflected by the $0,N$ state of the queueing model.

The data extraction algorithm differs from that typically employed with Lucid: instead

of reacting to interrupts generated when a *single* event has been digitized and stored by the VME modules, the IOM was configured to continuously poll the virtual chain module to learn when one or more digitized events were ready for read-out from the FIFO. If an event was ready for extraction, the I/O Manager performed a single chain block transfer (CBLT). The duration required to perform this CBLT corresponds to the model parameter, μ_2^{-1} .

Upon receiving the "FIFO-full" interrupt, the IOM altered its read-out policy such that *all* events were extracted from each module of the chain using three, *sequential* block transfer operations. This mechanism is much more economical than performing N chained block transfers. The duration of this sequential read-out operation corresponds to model parameter, μ_3^{-1} .

The front panel signals available from the v792 modules were used to gate a 5 MHz oscillator signal fed into one channel of a CAMAC scaler module. A second channel of the scaler was used to record the *uninhibited* number of oscillator pulses. The ratio of gated to ungated pulses provides a measure of the duration during which the system was prevented from responding to input event signals. Additionally, a third scaler channel was utilized to register the number of pulses emitted from the PMT-scintillator detector. The contents of all three scaler channels were recorded once per second for each test duration of 100 seconds over a range of average input event rates up to 45 kHz.

E.3 Analyses and Results

Solving the linear equation system of the queueing model requires that the average service and input event rates (durations) be provided. The average input event rate was allowed to vary over the range 0-100 kHz, while the values used for the average service durations were as follows:

1. $\mu_1^{-1} = 7.25\mu s$, the digitization period of the CAEN v792 modules as measured by oscilloscope.

2. $\mu_2^{-1} = 27.5\mu s$, the readout period required to perform a CBLT of three digitized events from the three-module VME chain.

3. $\mu_3^{-1} = 162.2\mu s$, the readout period required to perform three, sequential block transfers of $N = 32$ digitized events from each of the three v792 modules used.

Values provided for μ_2^{-1} and μ_3^{-1} were calculated using data obtained from Chapter 7.

In Figure E.3 are several curves demonstrating the fraction of lost input events (Erlang losses) as a function of average input event rate (λ). Computation of Equation E.3 produced the plot's red line for the tandem queue model with the average service time parameters described above, while data collected from the experimental system is represented by the individual points spanning (approximately) the first 43 kHz of input rate.

For comparison, curves are also present showing the Erlang losses for each station of the theoretical tandem system *taken in isolation*. That is, the green and blue curves represent the losses experienced by *M/M/1/1* and *M/M/1/32* queueing stations, respectively.

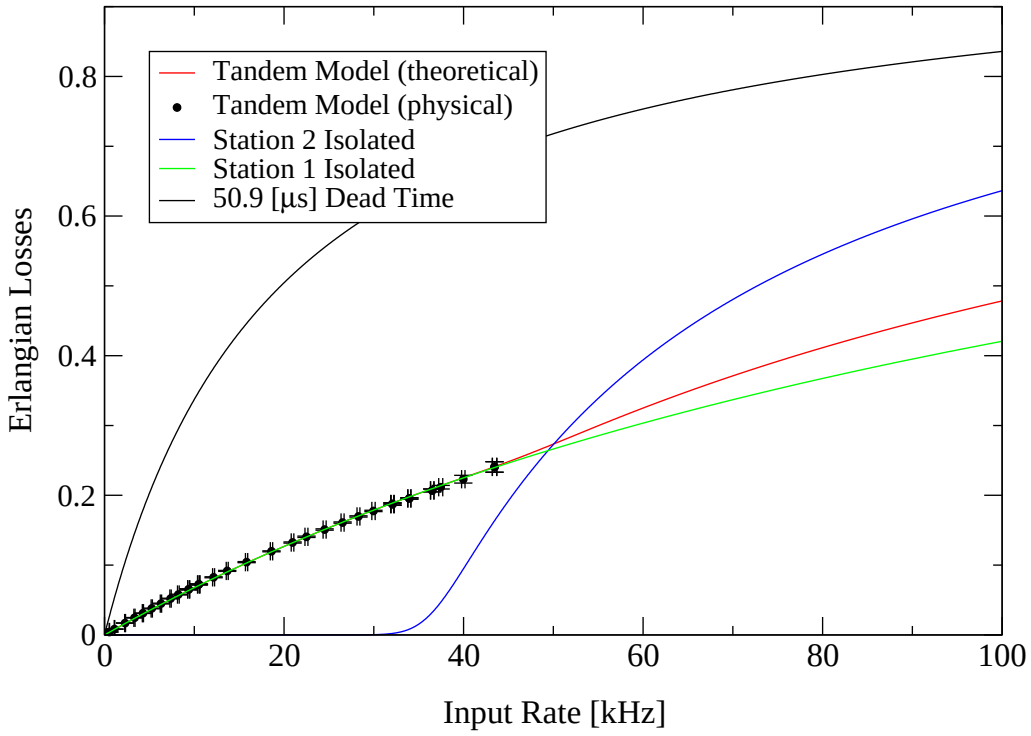

Figure E.3: Plot of model calculations and experimental results.

Finally, the Erlang losses are also plotted (black curve) for an *M/M/1/1* system with an average service time of $50.9\mu s$. This curve was reproduced using data obtained in Chapter 7, and represents the typical behavior seen with the Lucid DAQ as influenced by its "event-by-event" data acquisition algorithm.

Figure E.3 indicates that, for average input event rates less than 50 kHz, the dead time of this tandem system is largely governed by the service time of the first queueing station. This observation stands in contrast to the findings of Chapter 7, where it was found that the dead time of the DAQ was dominated by the duration of processes executed by the I/O Manager. However, it should be reiterated here that the greater dead time found in previous measurements was due to a *combination* of the "event-by-event" data extraction policy *and* the duration of IOM activities. The advantage gained by introducing a level of concurrency to the data acquisition algorithm is readily apparent in the form of significantly decreased dead time at the cost of increased complexity.