LUCID User's Guide

Comments, bug reports and changes to this software or documentation should be mailed to **lucid@skatter.usask.ca**. Bug reports should be accompanied by sample input or description files if appropriate.

Contents

Chapter 1 A Tutorial Introduction

Chapter 2 Using LUCID

Contents

Contents

Contents

Contents

Contents

Appendix D Histogram Save File Format

Chapter

1

A Tutorial Introduction

LUCID is a system which provides a way to acquire and analyze experimental data. It sets up a *data stream* that has three distinct parts: the **READER**, the **LOOKER** and the **WRITER**.

1. The **READER** gets data from some source such as **CAMAC**. It can perform simple software cuts, then make the data available to the rest of the system.
2. The **LOOKER** is optional, and allows analysis of the data as it is being acquired. Although its name implies passively viewing the data, the **LOOKER** can perform elaborate tests and mathematical operations on the data.
3. The **WRITER** is also optional, and can save data on magnetic tape, in disk files, or send the data to other programs as input.

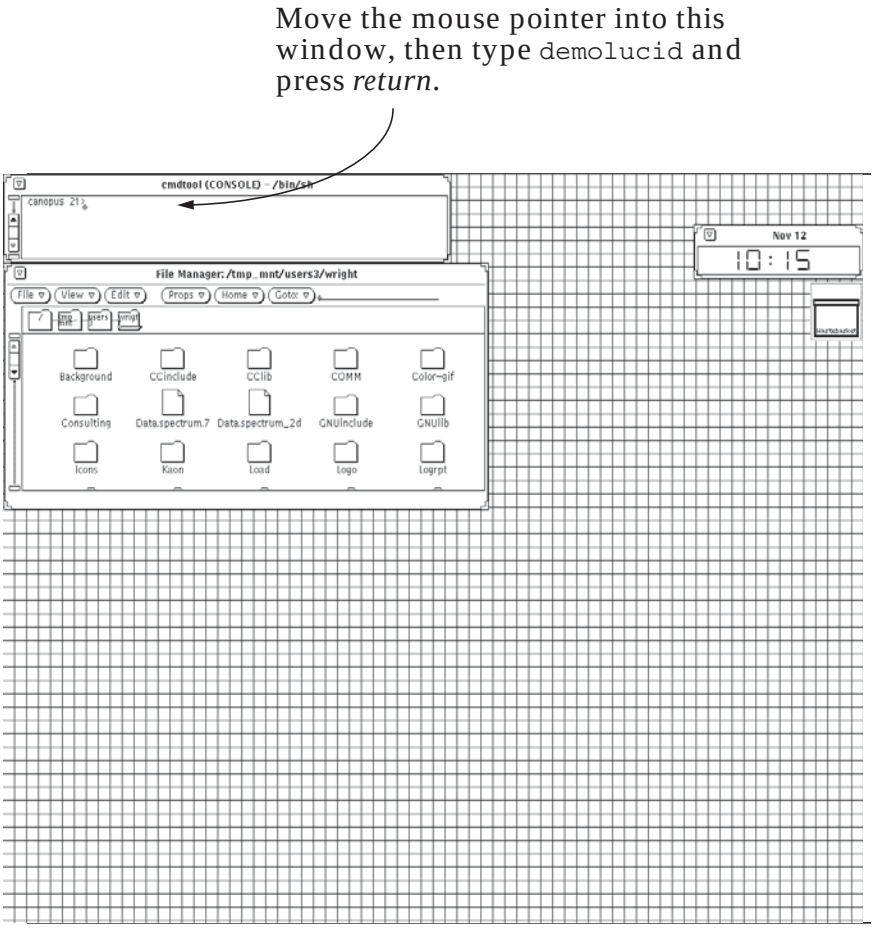
LUCID's knowledge of an experiment comes from the user's description of it; the user is required to make up description files for each of the **READER** and **LOOKER**. The experimenter starts and controls his experiment with the **XLUCID** command, which allows interaction with the **READER**, **LOOKER**, and **WRITER**, and lets the user change important parameters dynamically.

> lu•cid (loo´cid) **1.** bright; shining. **2.** transparent. **3.** designating an interval of sanity in a mental disorder. **4.** clear to the mind; readily understood [*lucid* instructions] **5**. clearheaded; rational [a *lucid* thinker] - **lu•cid´i•ty, lu´cid•ness** *n.* **lu´cid•ly** *adv.*

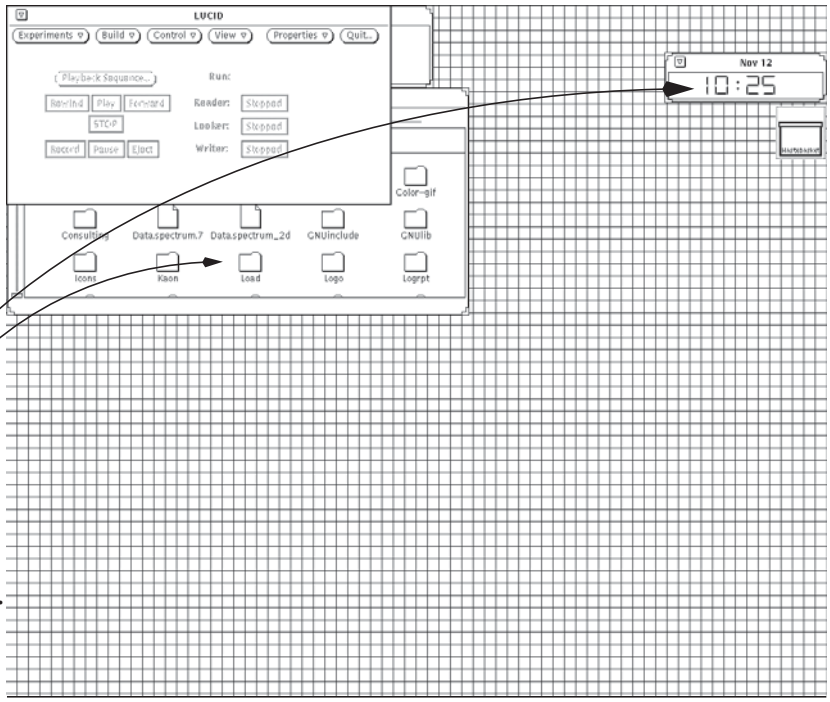
A Short Tutorial Session

The best way to learn how to use **LUCID** is to sit down and try it! You need a few things to get started. First, make sure that you have an account on the computer system. You'll be given a **log-in name** and a **password** which will let you access most computers in the Laboratory network. Second, find an available workstation and type in your login name and password.

To use **LUCID**, you'll need to start a "window" system called *OpenWindows*. The system administrator might have set up your account so that windows come up automatically. If not, the computer will ask what type of windows you want, and you should select OpenWindows. In any case, your screen will look something like this:



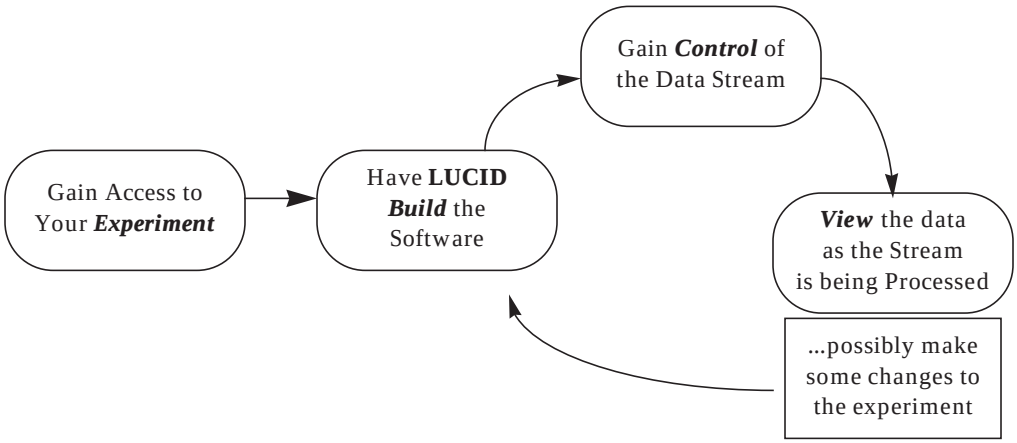
After typing `demolucid`, you'll have to answer "y" to "do you want to create a new experiment (y):", and then wait for a few seconds until the **LUCID** window appears. Your screen will look like this:



Move the mouse pointer into each of these other windows, and press the button labelled *Open*, located on the left side of the keyboard. The windows will close into *icons*.

You would get the same result if you just typed `xlucid`, but `demolucid` installs some small files which are required for this demonstration. We'll find out more about these files later.

We see in the window that **LUCID** has several *menu buttons*, located along the top of the window. The first four correspond to the most common steps that you'll take during the lifetime of an experiment. In very general terms, the process could look like this:



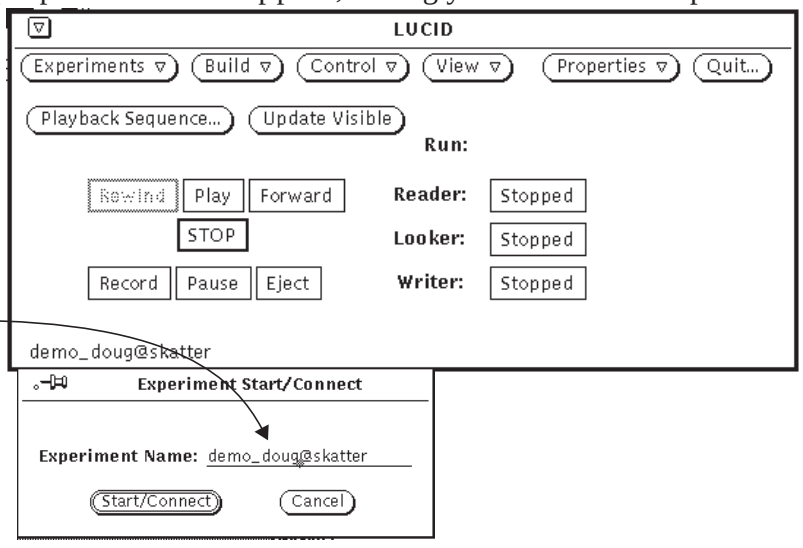
Buttons are drawn on the screen to allow choices to be made. To make a selection, move the mouse arrow over top of the button, and click it; that is, press then release the leftmost button on the mouse; the left mouse button is called the *select* button.

A *menu button* is slightly different; it has a small triangle displayed to the right of the label. This means that more options are available, which you can see by clicking the rightmost mouse button.

A *menu* is displayed, containing all of the options for that button. For instance, the **Experiments** menu button contains four options. The fourth one has been made inactive in this example, and the first one has been highlighted with an oval. In this case, the first choice is the default one; that's the one that will be used when you *select* the button without getting the menu. When you select a button containing an ellipsis (...), a sub-window will be displayed. To make a selection from the menu, just click the rightmost mouse button over top of the choice you want.

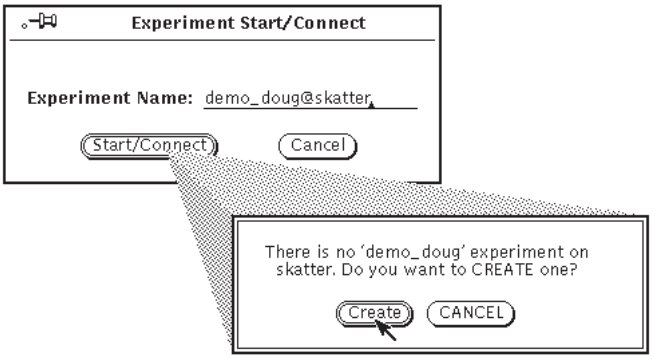
Starting an Experiment

To start an experiment, *select* the **Experiments** menu button. A pop-up window will appear, asking you to name an experiment:



Type "demo_*yourname*@skatter" for this tutorial

You're going to create a new "experiment" using your own name as a base. Make sure that the pointer is pointing into the **Experiment Start/Connect** sub-window, then (using your login name instead of *name*), type `demo_name@skatter` and press *return*[1]. First, you'll be asked to confirm your choice, then to confirm that the experiment should be created:

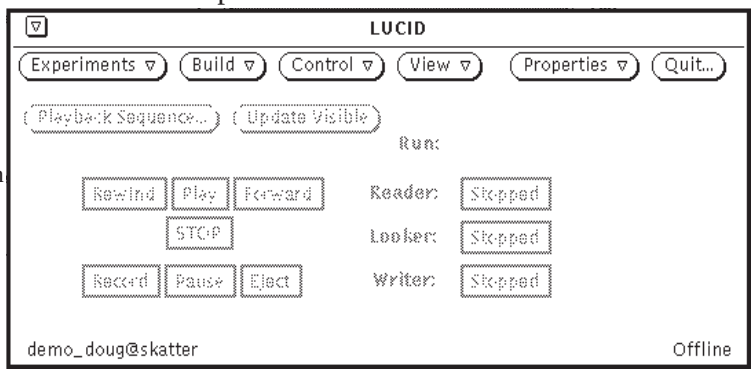


Use your own name, but make sure that it starts with the word "`demo_`...", so it won't be mistaken for a permanent experiment.

Select the **Create** button if **LUCID** is displaying the correct name. You won't be able to do anything else until you've assured **LUCID** that you've seen it's message. You can also just press *return* to confirm your request.

After a moment, the **Experiment Start/Connect** sub-window will disappear and a label at the bottom left of the **LUCID** window will name the current experiment.

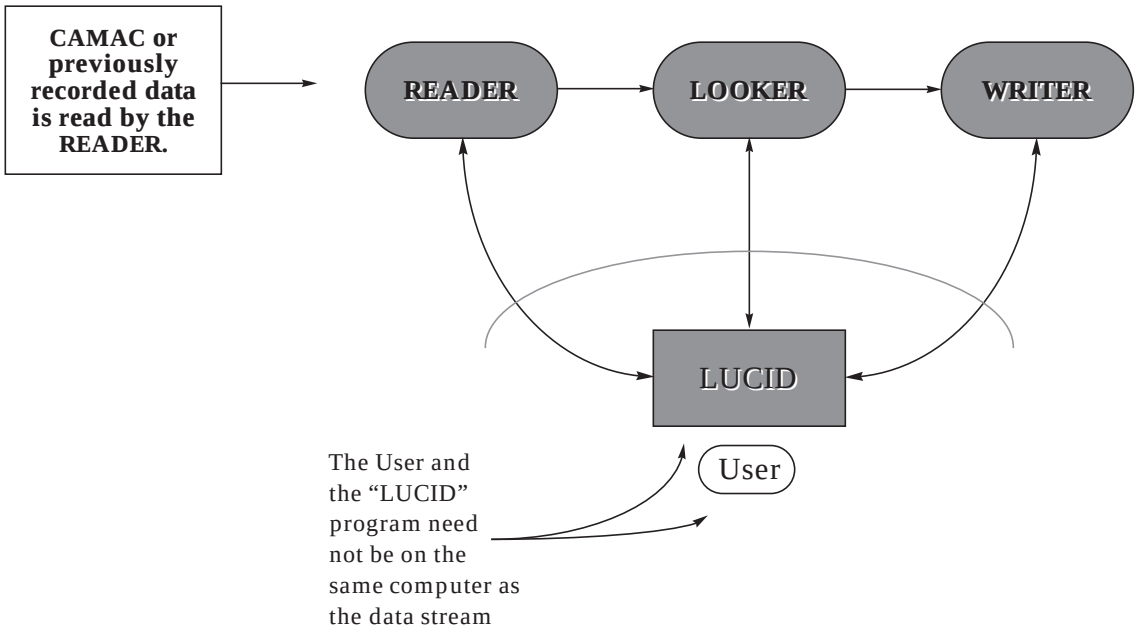
The experiment name appears at the bottom of the window when you're successfully connected.



1. You can press *return*, or select the ***Start/Connect*** button after typing the name. It is usually true that pressing *return* on the keyboard is the same as selecting the "confirmation" button (the button with the double border).

As the name `demo_doug@skatter` implies, part of this experiment will use the computer named "skatter". Specifically, we're going to do some simple offline analysis of data[2], using a **READER** and **LOOKER**. Both of these options will run remotely on "skatter" while we control them from the workstation.

This brings up an important point about how **LUCID** works. The data stream, including any of the **READER**, **LOOKER** or **WRITER** options that were chosen, is often started on a remote computer.



Keep in mind that when we speak of **LUCID** we're usually talking about the entire system shown in the previous diagram, and not just about the **LUCID** program through which we interact with the data stream.

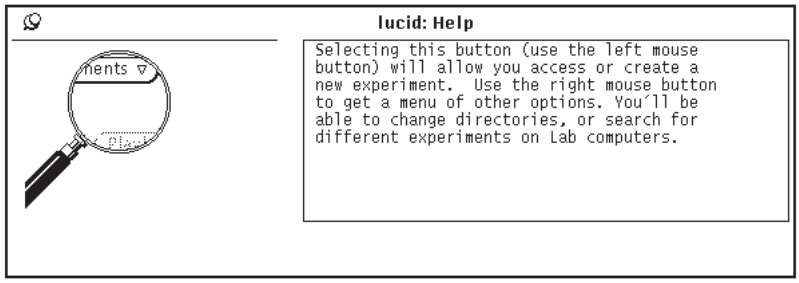
Using the Help Facility

Before continuing with the experiment, we should consider how **LUCID** will be able to help you after you lose this manual!

LUCID has a very simple **Help** facility. Just point the mouse to something within any **LUCID** window, and press the **Help** key on the keyboard[3].

2. In **LUCID**, the term *on-line* refers to the act of acquiring new data, and *off-line* refers to processing data which was acquired in a previous **LUCID** data stream.
3. Typically, the **Help** key is located on the far left of the keyboard, at the bottom. Some older keyboards don't have one; you'll have to press the **F1** key instead.

For instance, position the mouse pointer over the **Experiments** button. Press the **Help** key, and you'll get some information about it:



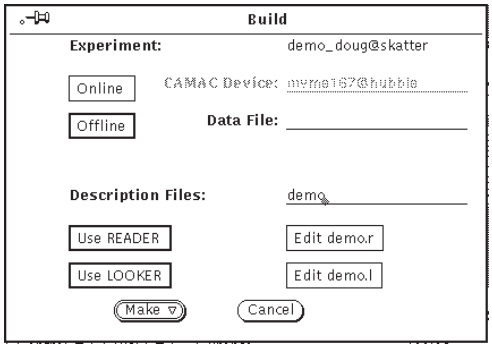
The important thing to remember is that pressing the **Help** key will give you help with any **LUCID** related object that you're pointing to. This is usually true, even with very small items, such as titles or small buttons; if you're not sure what something does, just press the **Help** key!

Building the Experiment

Now that we're successfully connected to the `demo_`*`name`* experiment, the next step is to **Build** the software. **LUCID** actually writes software for you, based on a simple description of your experiment. This is true if you're acquiring new or analyzing existing data.

The `demolucid` program has created two important files under your computer account before starting **LUCID**. The first is named **demodata** and contains some data which was acquired beforehand with **LUCID**. The second file is named **demo.l**, and contains a simple description of how one might analyze the data. We'll see that **LUCID** can write the analysis software very quickly.

Select the **Build** button in the **LUCID** window. The **Build** subwindow will be displayed. The first step is to select **Offline**, then enter the name of the data file, “demodata”, as shown in this example:



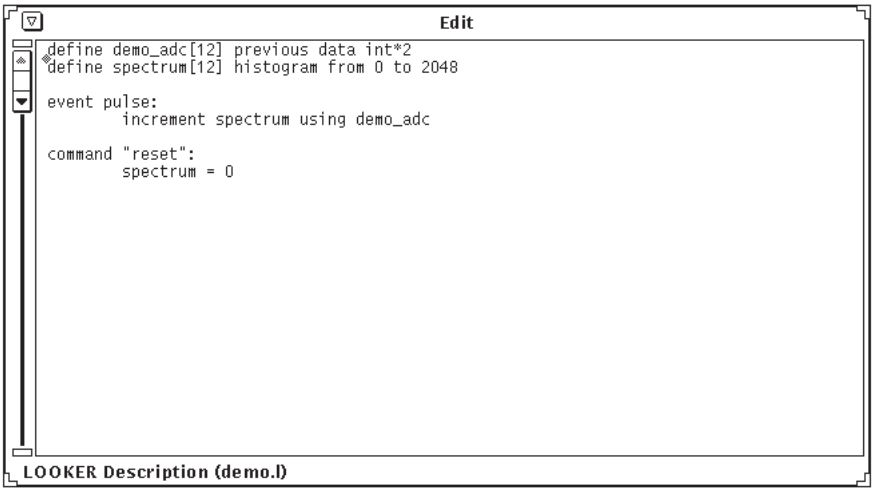
Next, you’ll have to tell **LUCID** that you want a **LOOKER** but not a **WRITER**; simply select the “looker” button if it isn’t already highlighted. Keep in mind that the **READER** always exists, so you won’t be able to “de-select” it.

The title above these buttons tells us that **LUCID** will use description files which start with the word “demo”. The **LOOKER** description file will therefore be named “demo.l”. Accordingly, the **READER** description file would be called “demo.r” if we were on-line.

This brings up an important point about description files. A **READER** description file is only required when data is being acquired online; the description file itself is saved along with the data! This means that when you eventually analyze the data, **LUCID** can just look at the data file and write the software automatically.

For this tutorial, the description files should all start with “demo”.

Now lets see what a description file looks like. At the right side of the window are the *edit* buttons, which allow you to create or change a description file. Choose the box labelled “Edit demo.l”, and a sub-window will appear containing a simple **LOOKER** description file:



The window that appears is actually a text editor. By positioning the pointer and selecting a position in the file, you’ll be able to easily change the contents. Chapter Four will discuss in detail what the description file can contain, but for now lets just look briefly at this small example.

```
define demo_adc[12]
  previous data int*2
```

- The first line tells **LUCID** that when the data was first acquired, an array of twelve integers was saved. This statement isn’t required because **LUCID** can find the array in the data file. It’s a good idea to include it, since it allows **LUCID** to check that it’s got the right data file. In our case, this array represents the twelve inputs on a peak-sensing ADC.

```
define spectrum[12]
 histogram from 0 to 2048
```

- This tells **LUCID** that the **LOOKER** will need an array of twelve histograms, each one accepting values from 0 to 2048. By default, histograms contain 1024 bins.

```
event demo_event:
```

- This line instructs the **LOOKER** to perform the next two lines whenever an event named “demo_event” is encountered in the data stream.

```
increment spectrum
 using demo_adc
```

- The appropriate bin in *each* of the twelve histograms is incremented by using the values from each of the corresponding twelve ADC elements.

```
command "reset":
```

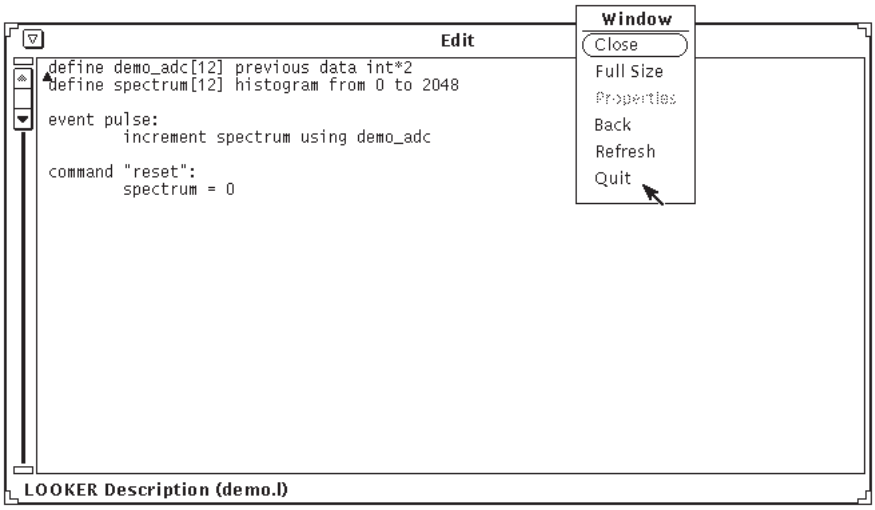
- Next, **LUCID** is told to perform some instructions whenever the experimenter gives a “reset” command;

```
     spectrum = 0
```

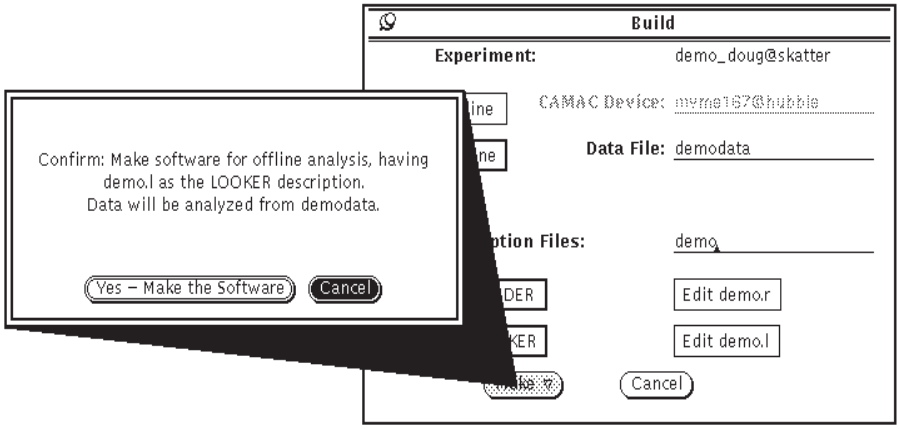
- Specifically, all bins of every histogram within the spectrum array will be set to zero.

These few lines are all that's needed to make **LUCID** write analysis software to histogram twelve ADC inputs. It really is quite simple!

When you're finished looking at the description file, quit the editor by getting the menu from the window's top label, and selecting "Quit" from it.



At this point, all you need is to point back to the **Build** sub-window, select the **Make** button, and the software will be generated. You'll be asked for a confirmation:



The resulting analysis software will be in the form of a program, about four or five hundred lines long, including comments and data structures. Very large experiments might end up with **LUCID**-generated programs containing ten thousand lines of "**C**" program source. Typically, we're not interested in looking through the resulting program, but **LUCID** adds lots of comments in case you're curious. Chapter 6 describes where the generated programs can be found.

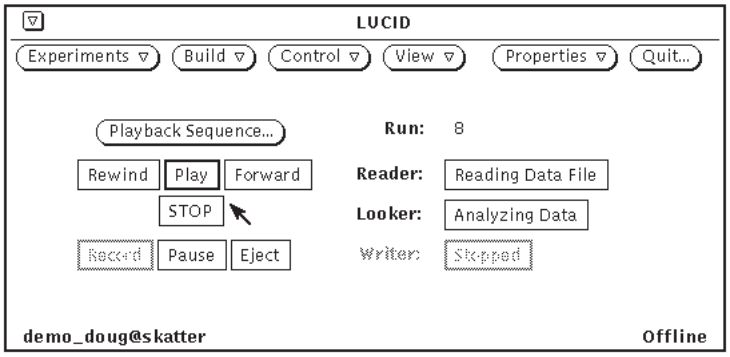
While **LUCID** is making your software and compiling it, a stopwatch figure will be displayed instead of the regular mouse pointer. When the pointer reappears, **LUCID** will have finished making the software. Most of the buttons in the main **LUCID** window will now be made usable, and the word “Offline” will appear in the lower right corner.

Controlling the Analysis

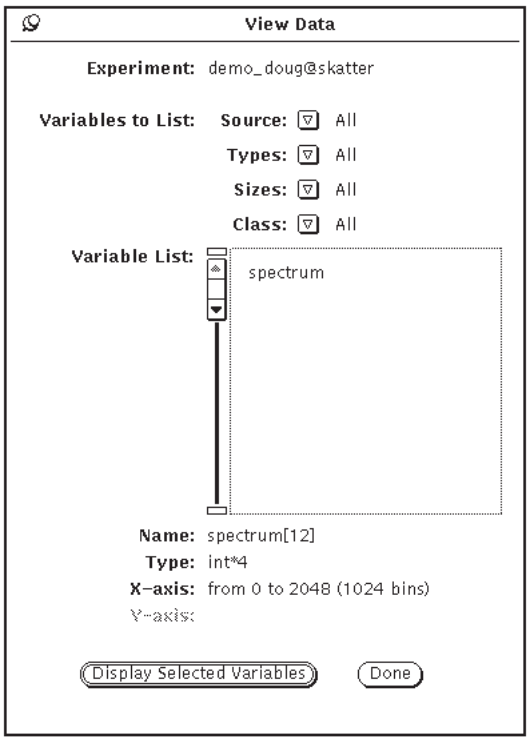
The main buttons in the **LUCID** window are meant to resemble controls of a simple tape deck or home CD player; you’re allowed to rewind or replay the “tape”, which can actually be a real data tape, but also a disk file or any other item containing your data. Actually, all of the windows we’ve seen here are used for online analysis and data acquisition, too.

We see that the “tape deck” has the **Stop** button selected. Also note that the **Record** button is masked out; it’s only available when we want to save data somewhere (when a **WRITER** exists).

If you’re ready to start the analysis, just select the **Play** button. This tells the **READER** to start reading and passing data to the **LOOKER**. When the **READER** reaches the end of the data stream, you’ll see the **Play** button pop out, and the **Stop** button will be highlighted again.



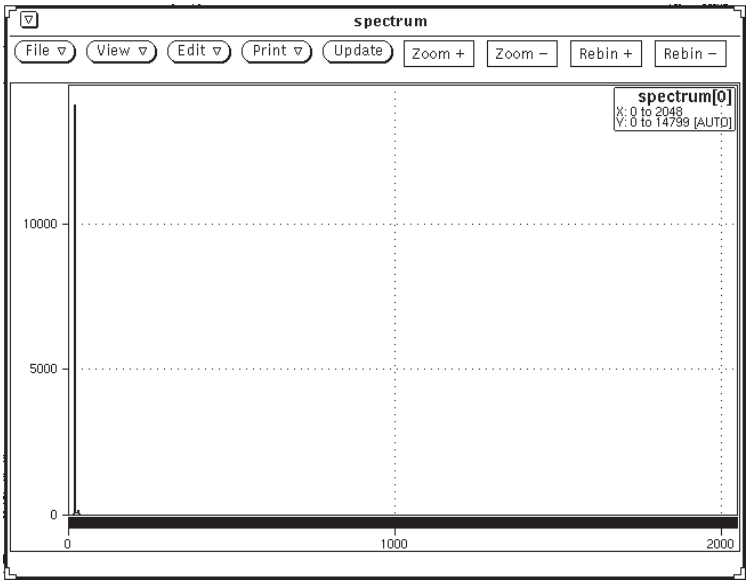
The next step is to look at the data. You're allowed to view any data defined in the **LOOKER** description file, whether the system is processing data or not. Select the **View** button, and the **View** sub-window will be displayed. Recall that our example only has one histogram variable defined:



Select the name **spectrum** on the List, then select the button labelled **Display Selected Variables**. A small icon will appear somewhere on your screen, which represents the data for that variable.

You can "open" the histogram's window by pointing the mouse arrow to the icon, and pressing the key labelled **Open**, located on the left side of the keyboard. The **Open** key is actually a toggle, so that if the mouse arrow is pointing to the window while you press **Open** again, the window will close.

When the histogram window is open, you will see something like this:



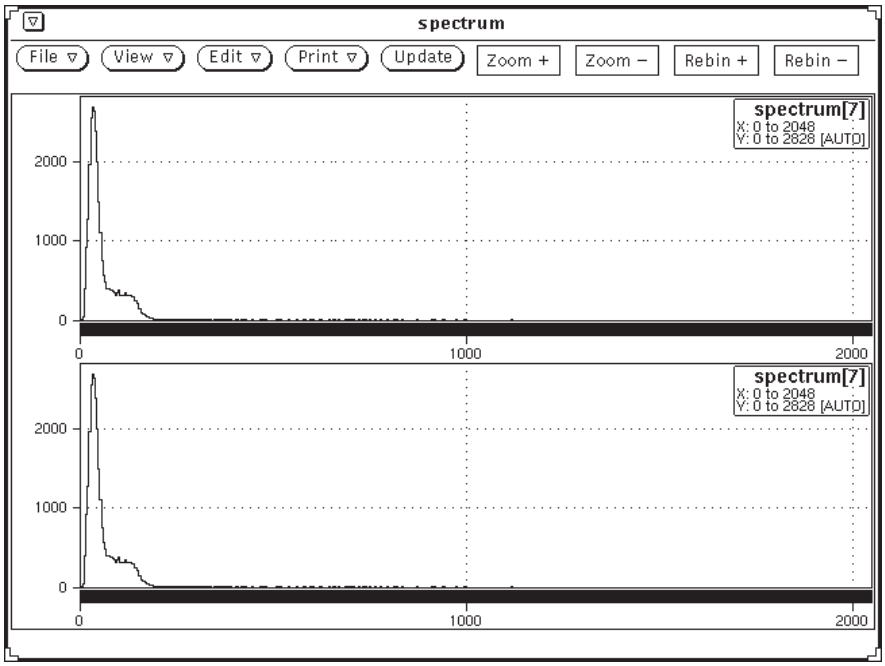
If the **Play** button is still selected, you might not have an up to date picture of the data, since it is still being processed. To update the view, move the mouse arrow into the histogram window and press ‘u’. The mouse arrow should always point to the window that you want to adjust.

Recall that the **spectrum** variable was actually defined as an array of 12 histograms. The title in the upper right corner of the window shows the name, but also the range covered by the X and Y axes. The word **Auto** appears beside the Y-axis, indicating that the Y-axis is being redrawn automatically so that the largest data value will be visible.

This particular view shows the first histogram in the array, **spectrum[0]**. Press the **Page Down** key on the right side of the keyboard, and the next element in the array will be displayed. Correspondingly, **Page Up** will cause the previous element to be displayed, and **Home** and **End** will display the first and last elements, respectively.

Most of the interesting data for this tutorial is located in the eighth histogram, that is **spectrum[7]**. Press the **Page Down** key a few times, until it is displayed.

A common requirement is that several different elements be seen at the same time. In fact, it is sometimes important to view the same element several times, in slightly different ways. By pressing the **Insert** key, also located on the right side of the keyboard, another view of the histogram will be inserted:



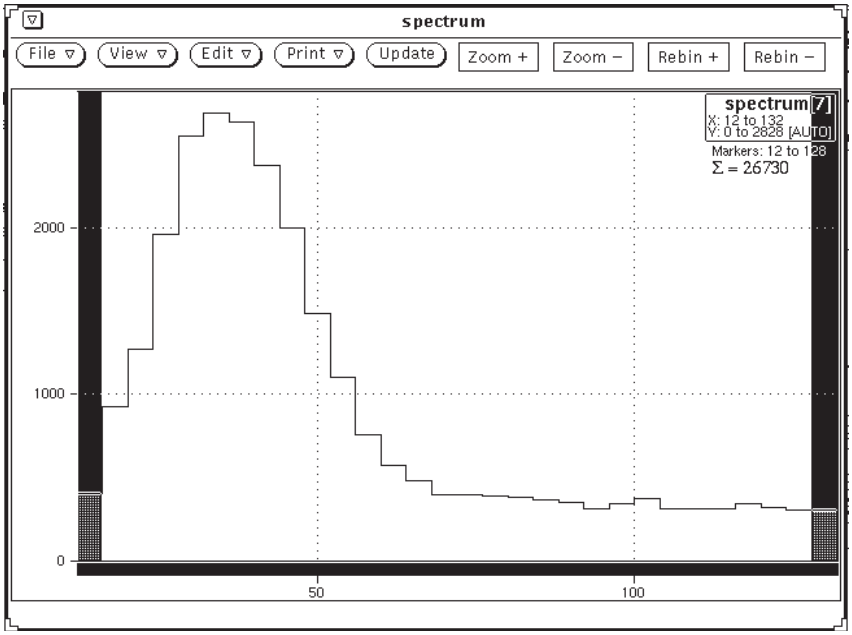
You can also press the **Delete** key to remove a view.

Note the corners of the window stand out in small wedge shapes. These are *resize corners*, and after the mouse pointer is positioned over one of them, the entire window can be resized by dragging the mouse while holding down the *select* (leftmost) button.

The next step is to "zoom in" on a peak in the data. The easiest way to do this is to draw *markers*, which delimit an area of interest. Move the mouse arrow over top of the histogram, and notice that it changes into the shape of a small plus symbol (crosshairs). When the left mouse button is clicked, a marker will be drawn on the corresponding bin in the histogram. You can actually *drag* the marker around, by holding the mouse button down while you move the mouse. It might not be apparent, but the markers are always drawn to the width of the current bin.

To put up a second marker, you must click (or drag) the middle mouse button. The second marker must appear on the right side of the first one. Notice that a “legend” is displayed beneath the title, telling the X-axis values which are currently marked, and what the sum of their Y values are. Also, the sum of counts in the marked area is displayed. This total *includes* counts in the bins which contain the markers.

To “zoom”, hold down the left **meta** key, which is located on the left side of the space bar on the keyboard; it has a small diamond symbol (◊) on it[4]. While holding that down, press the “5” key on the keypad (it’s the key on the far right with R11 written on the edge). This will zoom in to the marked area.



Notice that the marked bins now appear on the edges of what is displayed. Your markers might not have been placed on the same bins as in this example, so the display might not look identical.

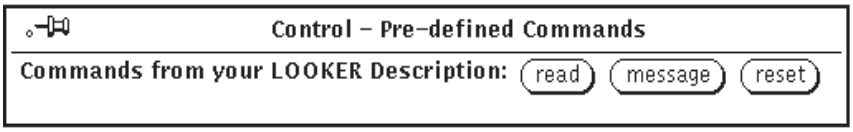
The leftmost meta key always means “zoom”, so you can zoom in or out vertically or horizontally, by using the arrow keys, with the meta key depressed. To zoom back to the initial view, press meta-Home.

4. Some older keyboards have the **meta** keys labelled with the words **left** and **right**, but are still located immediately next to the space bar, on either side.

Sending Pre-defined Commands

Recall that in our original **LOOKER** description file, we had a instruction that was to be performed whenever the user issued a "reset" command. Move the mouse arrow back to the original **LUCID** window, and select the **Control** button.

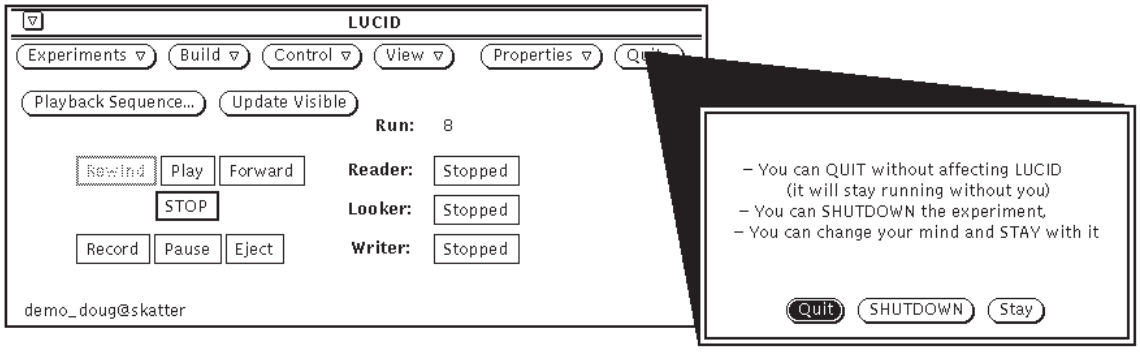
A window labelled **Control - Pre-defined Commands** will appear, and will contain a button for every command that you've included in the description file:



If you select the button labelled "reset", the spectrum histogram array will be set to zeroes, as was requested in the **LOOKER** description file. Remember that the histogram display itself won't change until you ask it to update the view.

How to Quit Lucid

At this point, we've done many of the things that are needed to effectively use **LUCID**, at least while offline. But one last thing which is often important is to have the system process your data "in the background". You typically don't want to wait around if the computer is going to take a long time to analyze your data. Select the **Quit** button on the main **LUCID** window, and you'll be asked to confirm:



LUCID wants to know how much of the system you really want to quit. By default, the data stream will stay running even after you've quit the **LUCID** program itself. This is important when someone else takes over the experiment from you. If the workstation computer should crash or stop, the data stream will still keep running without it when the experiment itself is running elsewhere.

- Selecting the **Quit** button will cause the **LUCID** program to stop, but keep the data stream active.
- Selecting **Shutdown** will make **LUCID** shut down all of the data stream processes, and then quit.
- The **Stay** button will cancel the "alert" window, and let you continue using **LUCID**.

Where to Go from Here

Now that we've seen how simple it is to use **LUCID**, you should go ahead and set up a real experiment. You're certainly welcome to "play around" with the demo experiment that was just set up, but **take note** that all demo experiments are periodically removed.

If you're just getting started with **LUCID**, you should find out what kinds of statements the description files can contain. These are described in Chapters 3 and 4.

Chapter 2 describes the **XLUCID** program in more detail, including how to start up and administer an experiment. It explains how to control the **READER**, **LOOKER** and **WRITER**, and monitor the progress of data through the data stream.

Chapter 3 introduces each of the statement types that you'll need to describe your experiment to the **READER**, mainly in terms of the equipment you have.

Chapter 4 explains how to set up a **LOOKER** description file, and describes histogramming and analysis in general.

Chapter 5 talks about **WRITER** considerations.

Chapter 6 gives an overview of how all the pieces of **LUCID** fit together.

Chapter 7 documents the many different support subroutines and programs that may be helpful when running a **LUCID** experiment.

Each section in this guide starts out with just enough information to let you use a particular feature. More details are given as you read further into the section, so you can probably skip a lot if you're getting started. When examples are given, **LUCID** keywords are printed in a "typewriter" typeface:

```
trigger scaler every 10 minutes
```

Brackets around a word indicate that the named words are *optional.*

We will assume that you're familiar with CAMAC and the variety of electronics that are available through it.

Chapter

2

Using LUCID

LUCID was designed to be easy to use.

It was also designed to help experimenters think about their work on a higher level. As you read this manual, try to forget the details of your experiment, and think in more general terms.

Most experiments are quite complex in nature, and the best way to solve the problems and get meaningful information is to design the experiment from the top down: start with a very general statement of what you want, then break the job into a few specific tasks. Each of these tasks should then be divided into jobs which are more detailed, and this process repeated until each task addresses a unique detail of the experiment.

LUCID fits into the middle of this hierarchy; it only needs to know about a few general tasks and it will worry about most of the details for you. For example, you should think about the physical occurrence which will signify an "ADC event", and not worry about how to access the ADC in the most efficient way.

Getting Started

The first thing to do is decide what you want to measure, when to make those measurements, then determine how those measurements will flow through **LUCID** and be recorded.

Events

If you're starting a new experiment, data will probably originate in some **CAMAC** equipment. You must determine how many different *event* types you'll need (an event is simply information that is grouped together to describe something). Some events may only contain a single ADC measurement, while others can contain as many as several thousand distinct values. The current version of **LUCID** limits an event to less than (approximately) 32,000 bytes. Furthermore, the rate at which data can be acquired is currently limited to about 245,000 bytes per second. **LUCID** allows 65535 types of events to be used in the same experiment.

In addition to the primary data for your experiment, other event types might be necessary; for example, you might want several scaler (counter) values to be recorded at regular intervals.

Triggers

A *trigger* simply determines *when* to make some measurement. A particle interacting with scintillation material could represent a trigger, signifying that **CAMAC** equipment should be read out at that point. The next chapter describes several different ways to trigger the acquisition of event data.

In many experiments, it may not be possible to relate a unique trigger to every different event. For instance, you might not know what *type* of interaction occurred until a bit pattern register is examined. **LUCID** handles this by letting you reject the current event and trigger a new event after performing some tests.

The best way to get started with an experiment is to list every different type of event the experiment will encounter, give each one a unique name, and describe how each event will be triggered. This is certainly the first step in deciding how to set up your electronics, and will make the job of describing the experiment to **LUCID** very easy. In the next chapter, we'll see that in order to acquire data, **LUCID** just needs to know where to get the data and when to read it.

Administering an Experiment

If you're interacting with existing data from a previous experiment, things are much easier because each event type and trigger has already been defined. You won't have to prepare any kind of description, because **LUCID** can extract the original description directly from the saved data. You *should* at least be familiar with the names of the variables that were used originally, on-line. If not, **LUCID** can list them for you (see **lucidview** on page 7-4).

LUCID allows you to create as many "experiments" as you want. An experiment can be thought of as a data stream, as described in the first chapter, and **LUCID** allows you to scan over the data as many times as you want. In fact, you can have several different experiment names set up to look at the same data, and several build names within an experiment, if you wish.

LUCID maintains a list of trusted users for each experiment. The person which initially creates the experiment is considered to be its administrator, and is responsible for keeping the user list up to date. See **User Permissions on** page 2-18.

If you want to access an experiment which has already been set up and may be currently running, make sure you know the following things:

- The name of the computer on which the experiment was created.

- The original name of the experiment, as it was registered with **LUCID**.
- That the administrator of the experiment (the person who first created it) has added your name to the list of users.

If you want to create a new experiment to acquire data, make sure you:

- Get the name of the computer which is connected to your electronics (**CAMAC** equipment, for example), and make sure that the system will let you log in; your login name and password will be the same on *all* computers.
- Choose a unique name for your experiment. When you first start up, **LUCID** can give you a list of all existing experiments on all Lab computers. If you're just learning about **LUCID** and don't care about the name, just choose one that isn't already listed.

Finally, if you want to create a new experiment to analyze existing data, you should:

- Find a computer which contains the data file, or the tape drive which will read your data file.
- Choose a unique name for your experiment, as described above in the on-line example.

One last point about getting started: when a number of people are involved with an on-line experiment, it is sometimes easiest for all involved if an **experiment account** is set up. This gives a common location for all **LUCID** user files, and a common account for controlling and monitoring the experiment. These are only allowed for **on-line** experiments. If this sounds useful to your situation, you should see the system administrator for details.

The LUCID Program

At this point, you should be familiar with the tutorial given in **Chapter 1**. The OpenWindows graphical user interface was introduced, and at this point we'll assume that is what you're using, and also that you're familiar with *how* to use it. If not, there is a good on-line tutorial available, as well as documentation. If you don't have the OpenWindows interface on your workstation, or you don't have access to the documentation or on-line tutorial, see your system administrator before continuing. The remainder of this chapter is a reference rather than a tutorial.

To start **LUCID**, you can just type the word "lucid" as a command in a window. On some systems, you might have to type "xlucid" to get the window version; see your system administrator if you're not sure.

You can also type

```
xlucid experiment_name@computer_name
```

This is a shortcut for having the program automatically connect to the named experiment on the named computer. In fact, there are lots of options that are available on the command line, all which will over-ride the corresponding entry from the property list. See **The Properties Menu** on page 2-15 for a more complete explanation of what each option does. The valid command line options are:

Command Line Options

-n	ignore the **X11** default properties for **LUCID**
-b file	specify the base description file name
-d number	set the program's debugging level
-e name	specify the text editor to be used
-i file	specify the input data file for off-line data
-m file	specify an alternate **CAMAC Modules** database
-s file	specify the device name for the frontend processor
-t number	set the maximum number of displayed messages
-u directory	specify the user directory to execute in
-F number	specify the **lucid_finder** debugging level
-L number	specify the **looker** debugging level
-M number	specify the **manager** debugging level
-R number	specify the **reader** debugging level
-S number	specify the debugging level for the front-end processor
-U number	change the status update interval
-W number	specify the **writer** debugging level

Any argument typed on the command line that is not a "flag" is taken to be the name of an experiment to run, as in the example above. Because you can only look at one experiment at a time, **LUCID** ignores any extra experiment names entered on the command line.

After typing the "lucid" command, the **LUCID** window will be displayed, looking like this:

The **LUCID** Icon.

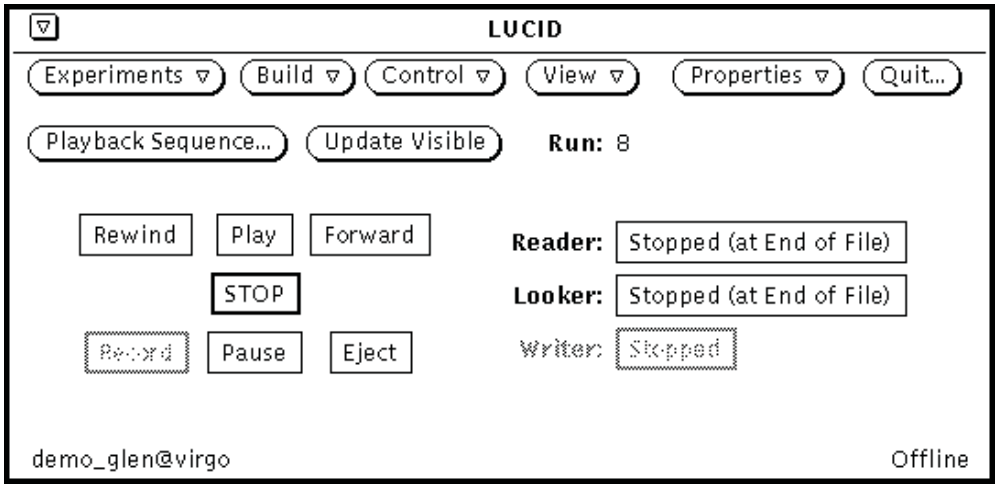


Figure 2.1. The Main LUCID Window

The window has three main sections.

- Menu buttons, located along the top of the window. It is from these buttons, or their menus, that the experiment can be managed.
- Control buttons, located on the left side of the window. In the example shown above, they are inactive. These buttons will become active only when software has been successfully generated.
- Status buttons, located on the right side of the window. In the example shown above, they are inactive. These buttons will become active only when software has been successfully generated.

The next three sections of this Chapter describe them in detail.

Menu Buttons

In the example of **Figure 2.1**, the user has not yet connected to an experiment; there is no experiment name displayed in the lower left hand corner of the window.

We'll first take a closer look at the menu buttons along the top of the window. They can be used to bring up the menus as shown in **Figure 2.2.**

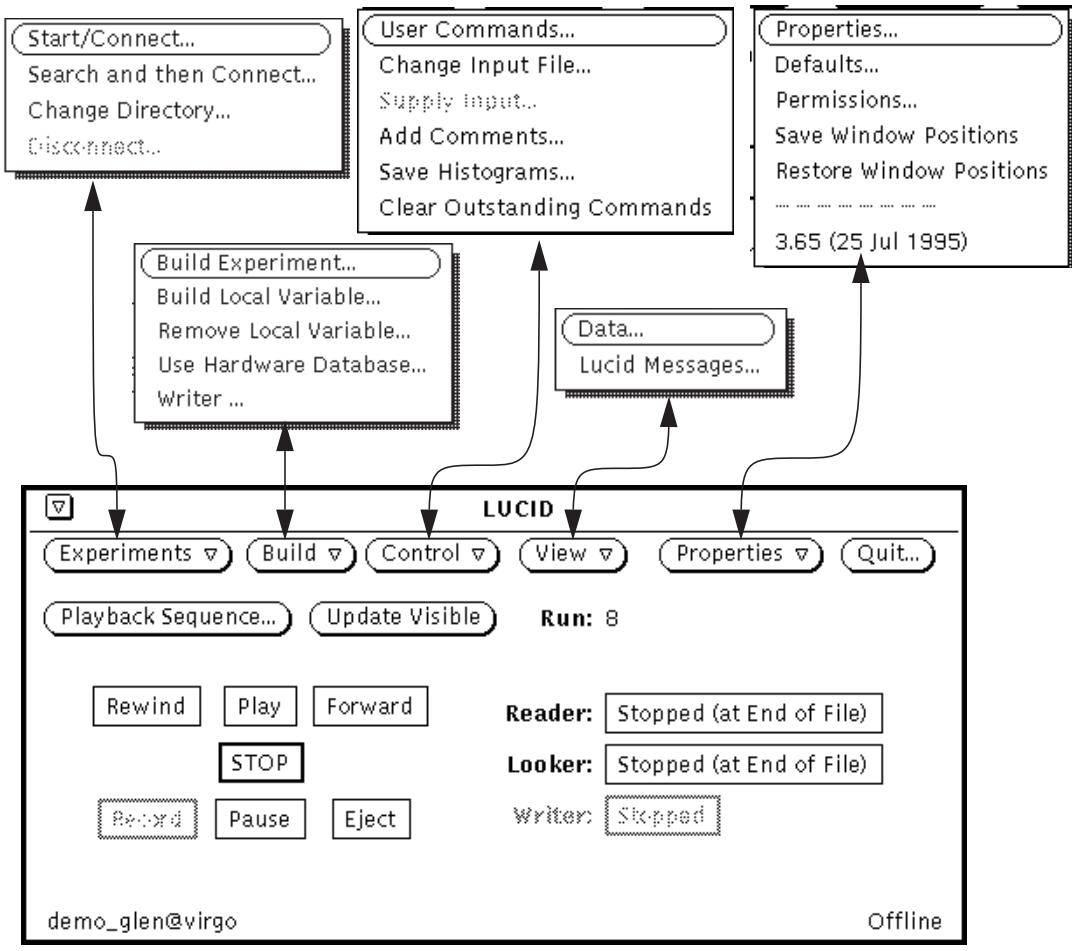


Figure 2.2. LUCID Menus

The *Experiments* Menu

The **Experiments** menu is shown in **Figure 2.3** and is usually the first one to be used in a given session. The default selection from it, **Start/Connect**, must be chosen before most other choices will work. It brings up a sub-window which allows the user to name the desired experiment. Giving the experiment name on the command line performs the same function, and most experienced users make use of this shortcut. Experiment names are commonly of the form "*name@computer*", and are not allowed to contain special symbols or blank spaces. If the name doesn't contain the "*@computer*" part, the experiment will start on the computer running the lucid program itself.

The user who first creates an experiment is considered the *experiment administrator*, and is allowed to name other users who can access the experiment. Regardless of this fact, the first person to gain access to a particular session of an experiment, assuming he

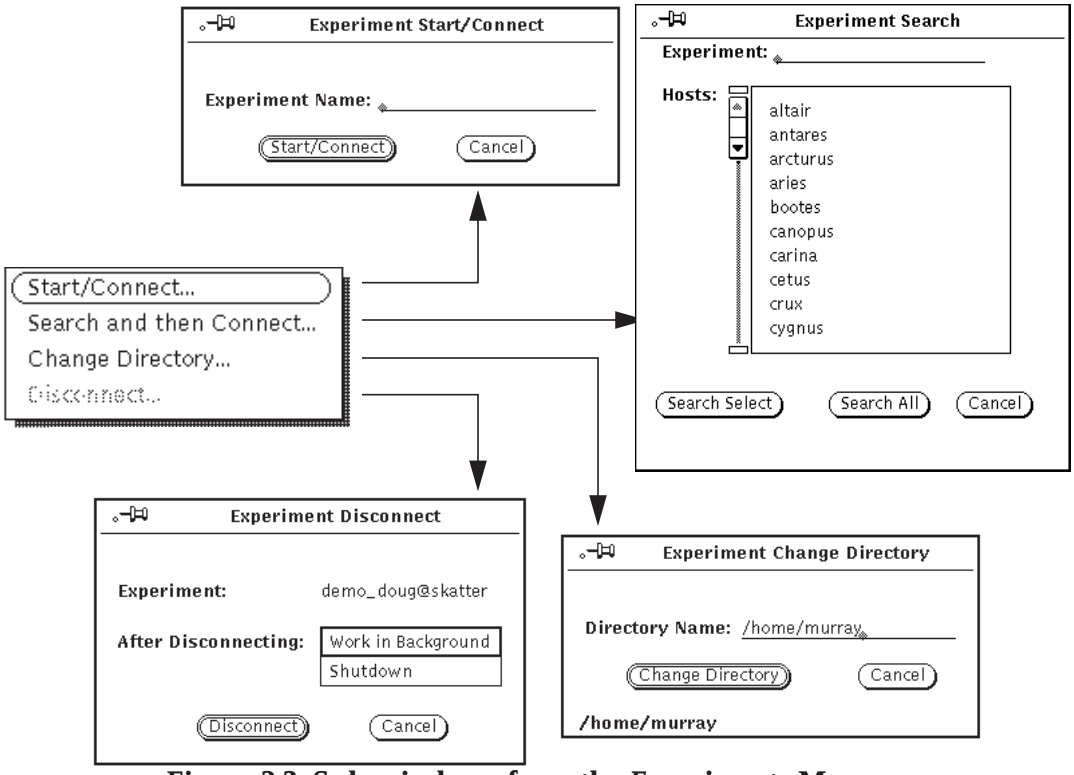


Figure 2.3. Sub-windows from the *Experiments* Menu.

has permission from the administrator, may use all functions described in this manual; other colleagues who subsequently connect to the same experiment at the same time are *visitors*, in that they are not allowed to make use of the control buttons on the main window. They are allowed to view data, and perform the most basic operations, as described later in this Chapter.

The **Search and then Connect** option causes a list to be generated of *all* **LUCID** computers in the Lab. A list of all experiments on all the local **LUCID** computers can be found by selecting **Search All**. The list of experiments on certain hosts can be obtained by selecting the desired hosts in the **Hosts:** list, then pressing **Search Select**. If there is a specific experiment name to search for, it can be entered in the **Experiment:** field, and the computers searched will only look for that particular experiment.

The **Change Directory** option causes a sub-window to appear which allows different working directories to be used. This option can only be used before the program is connected to an experiment. This is most useful when different directories in a computer account are used for different experiments, and one needs to switch between them. Remember that simply changing to the appropriate directory before starting the “lucid” program

achieves the same result, at least for the first experiment connection.

The **Disconnect** option allows the user to disconnect from the **READER**, **LOOKER** and / or **WRITER**, thereby allowing connection to another experiment. The user is given the option of shutting down the experiment after it is disconnected; that is, terminating any of the **READER**, **LOOKER** and / or **WRITER** pieces that may exist. As shown in the previous diagram, **Disconnect** is inactive if **LUCID** is not connected to an experiment.

The *Build* Menu

Once connected to the desired experiment, the user will typically go ahead to use options beneath the **Build** menu. This won't be the case if the experiment is already in progress when the connection was made. In such a case, someone has already built the experiment, and the control buttons will all be active.

Sub-windows which originate from the **Build** menu are shown in **Figure 2.4**.

Building the Software

The default option under **Build** is named **Build Experiment**. A sub-window is presented which allows **Editing** of description files, naming of the input file (or hardware to use if on-line), and ultimately, generation of the software. This window will be used every time description files are prepared, or used to generate software. The user must first decide whether the system is on-line or off-line, and select the corresponding button near the top of the window, as shown in **Figure 2.4**.

On-line or Off-line Mode, and Using Data Files

On-line or off-line, a filename is required to name the source of data. When on-line, the filename refers to the location of the preprocessing computer (see **Chapter 3**), and is of the form "processor_type@hostname". This indicates that the preprocessor is the **VME** processor named "hostname", which is a processor board of type "processor_type". When off-line, the name of the file containing the data must be entered. Tape drive units are named as files, and are typically of the form "/dev/nrst0". When reading from tape remember to include the letter "n" after the second "/" slash mark, which indicates that the *no-rewind* mode should be used. The **n** ensures that when finished, the system will *not* rewind the tape. thereby saving a lot of time! Also, **LUCID** understands remote tape drives; you can specify a tape drive on a remote workstation by using a name of the form "regulus:/dev/nrst0". The computer name (and a colon) precede the tape drive name. In any case, the user should check with the sys-

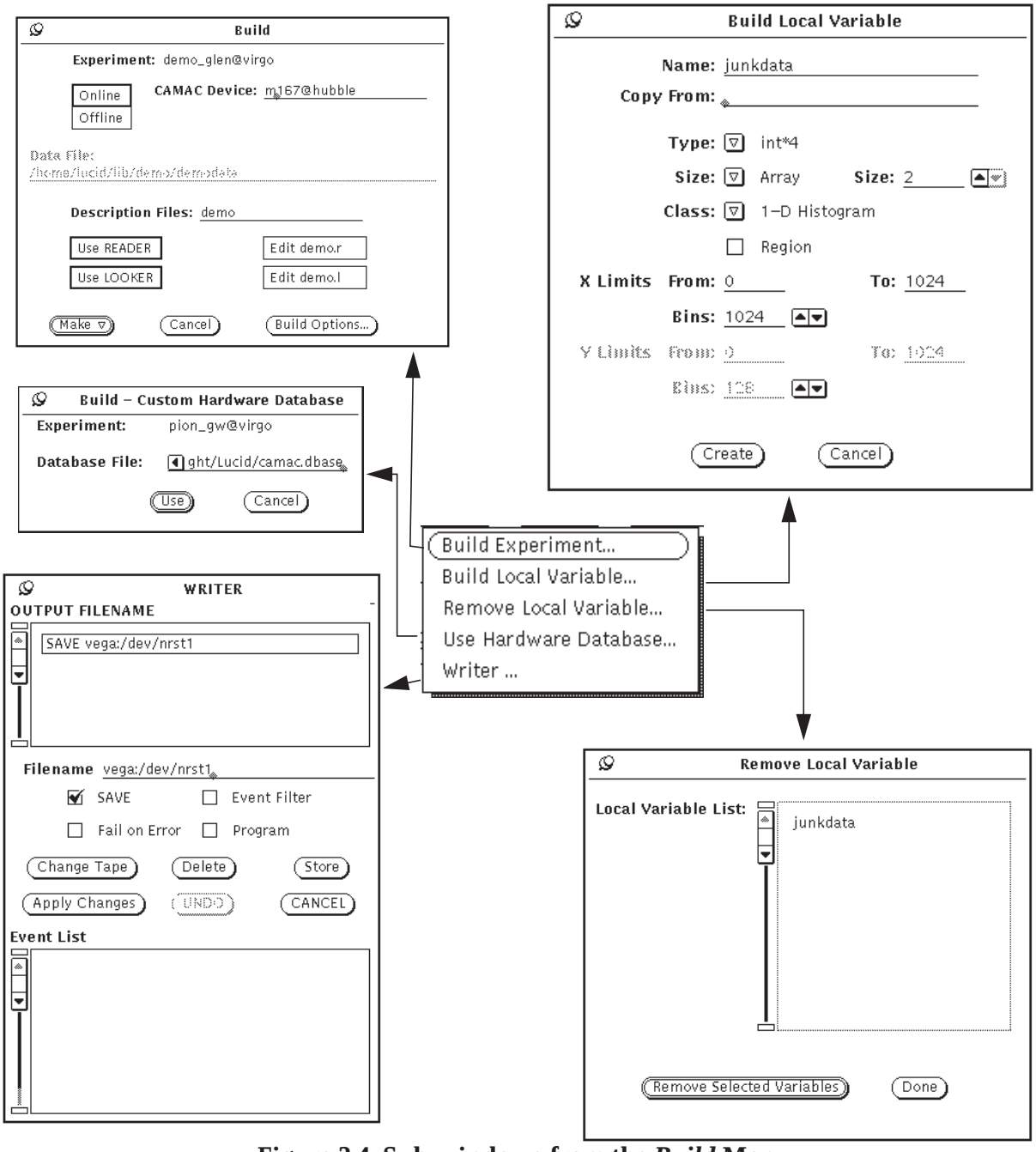


Figure 2.4. Sub-windows from the *Build* Menu.

tem administrator to determine the correct names of tape drive units.

The user is allowed to change the input file after the software is built, when off-line; see **Figure 2.7** under the **Change Input File** option for details.

The most important and time consuming step is the preparation of the experiment description files. Chapters 3 and 4 describe

what the files may contain. The description files can be prepared beforehand, using any text editor.

The user should first select the **LOOKER** button if a **LOOKER** description file is to be used. Recall that the **READER** is always present. Next, the user can select the appropriate **Edit** buttons to allow writing or modifying the descriptions. When finished, remember to get out of the editor completely by bringing up the menu from the top border of the edit window, and choose *Quit*.

Check for Status or Error Messages

The user can select the button labelled **Make**, at the bottom of the window. Software will be generated and compiled. If some mistakes were made in the description files, error and status messages will be displayed; look under the **View** menu to display messages.

The **Make** button is actually a menu button, having a menu which contains four options. The user may have the software generated, compiled, loaded and started all in one operation, which is the default choice. The second choice is to simply generate the software to make sure that the description files are acceptable. In this case, nothing is compiled, loaded nor started running. When debugging a looker file, the above two options are also available with *array limit checking*. This causes the generated looker code to check that all accesses to arrays are within bounds. **Figure 2.5** shows the menu available with the **Make** button.

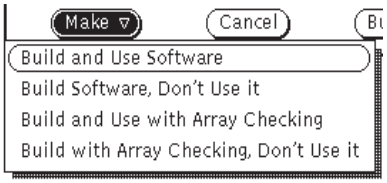


Figure 2.5. The *Make* Menu Button

The **Build Options** button brings up a window (not shown) that passes user values to the **MANAGER** for a greater degree of control over generating the **READER** and **LOOKER**. To use this, you should be familiar with the different programs that are used when generating **LUCID** software.

Build Local Variables

Other options are available from the main **Build** menu. The **Build Local Variable** option allows the user to define a variable, just as is done in a **LOOKER** description file, and use it for temporary storage of data. The intention is that eventually arithmetic involving histogram data may be done dynamically, and local variables can be used to store temporary results. Currently, temporary variables can only be used to read saved histogram files (See **The File Menu Button** on page 2-24) or for temporary

region variables when building a new region description for a histogram (See **Region Mode** on page 2-37).

The option to **Remove Local Variable** simply frees up the space associated with temporary, local variables that have been created by any means.

The **Use Hardware Database** option can be used when acquiring data on-line, when a new **READER** is built, to specify different **CAMAC** features than normal. Typically, a small system database is used to describe various features of **CAMAC** equipment. If desired, one can change these characteristics by supplying their own database file. The format of the file is described in **Appendix A** near the end of this manual. The name of the temporary database file should be set *before* the software is generated with the **Make** button in the **Build Experiment** sub-window.

For convenience, the modules specified in the system database are always read after the user's database. **LUCID** allows the user's definitions to take priority over existing ones.

[1]The *Build Writer* window gives the experimenter direct control over the output devices being used in an experiment (see **Figure 2.6**). The first part of this window lists all the output devices selected for output with this experiment; the first time this window is started, the list is empty. The second part shows the selected output device and its associated settings. The third part are the control buttons. The final part is the event list; this is only applicable if the current event is tagged as an *event filter*.

To add a new output file, ensure that no entries are currently selected, then enter the name in the *Filename* field and press return. The default flag setting of **SAVE** is displayed. If there is only one output, or this is the *important* output, set the **Fail on Error** box. If the filename entered is actually a program, select the **Program** box. Event filtering can be done by setting the **Event Filter** box.

The six control buttons provide the following features:

- **Change Tape** shuts down the current writer, ejects the tape, and waits for the user to acknowledge completion of the tape change.
- **Delete** removes the currently selected entry from the list. The **WRITER** is not informed of this change until the **Apply** button is pressed.
- **UNDO** puts the most recently deleted writer entry back on the list. The **WRITER** is not informed of this change until the **Apply** button is pressed.

1. Further discussion of the **LUCID** writer abilities is found in Chapter 5.

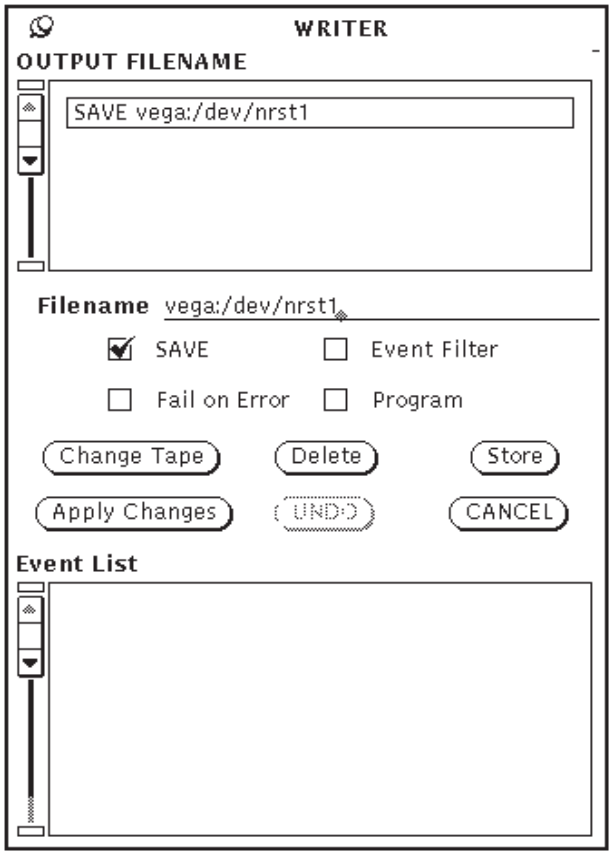


Figure 2.6. The *Build Writer* Window

- **Apply Changes** informs the **WRITER** of the net changes to the file list.
- **Store** places the current list of output devices, and their settings, in your **.Xdefaults** file. This is automatically re-read and used as the default settings the next time **Xlucid** is started from the same account.
- **CANCEL** quits the window without applying any changes. The next time the window is started, it will be back to the state that the **WRITER** is currently in.

The Event List shows the list of all events, and the status of the event for the current output device. Events can be saved completely, only at the output device speed, or never.[1]

The *Control* Menu

After software has been generated, the user will probably start a run by selecting the **Play** or **Record** button. All of the main control buttons are described in the next section, but other less com-

1. At the time of writing this section, the Event List feature had not been implemented in the WRITER.

mon features are found as options under the **Control** menu, as shown in **Figure 2.7**.

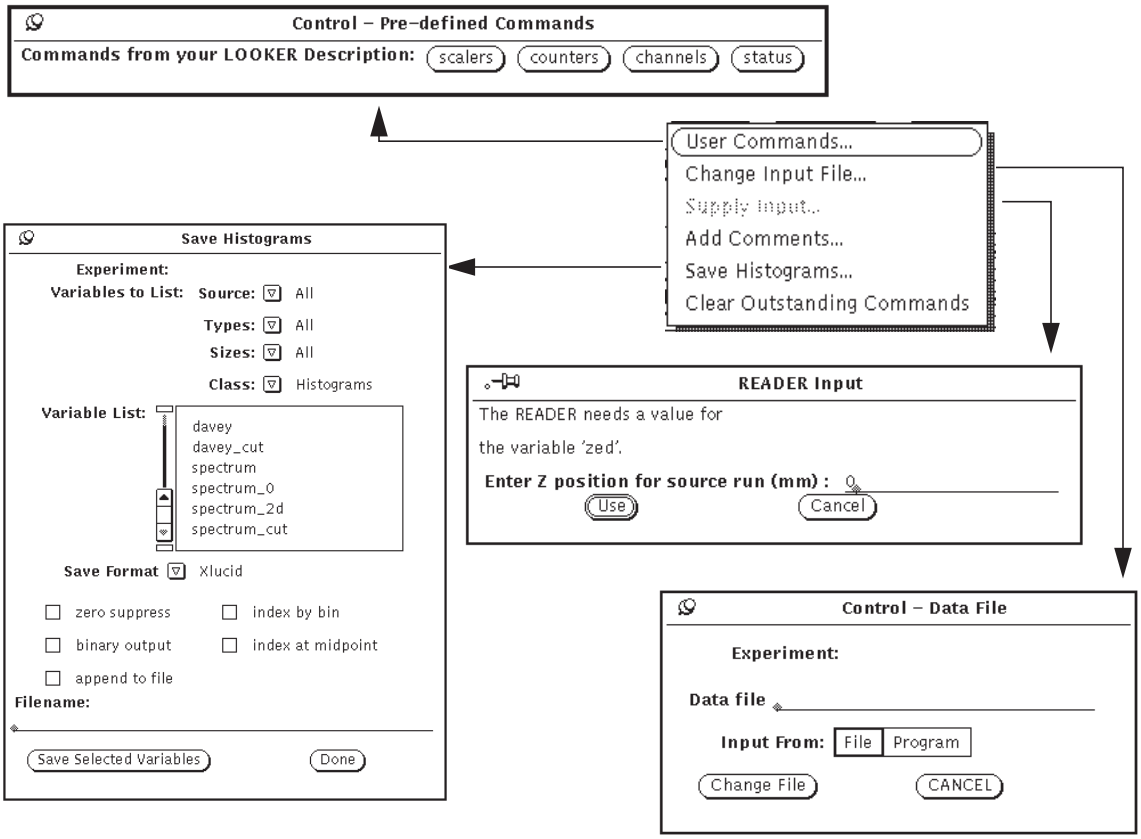


Figure 2.7. Sub-windows from the *Control* Menu

The **User Commands** menu option allows the user to issue commands which were specified in the **READER** or **LOOKER** description files. A sub-window will be displayed which contains a button for every command given in a description file. If the same command was listed in both the **READER** and **LOOKER**, then selecting that button will cause the command to be sent to both. The **User Commands** menu option will be made inactive if no user commands were defined.

Changing the Input Data File

The **Change Input File** option allows the user to change the source of data *only when off-line*. The file name should be a complete path name, meaning that it should start with a slash (/). The user should keep in mind that **LUCID** might rebuild the software if it encounters some change in the way data has been saved. For example, if the **READER** description file was changed halfway through an experiment so that several new variables were saved in some event, then **LUCID** will recognize that change whenever the new data is encountered. It will rebuild the

software to accommodate the changes and tell you about it. Input can also come from a program directly to the **READER** when off-line. This might happen when comparing acquired data to data from a simulation program.

If a description file was set up to ask the user for input at some point, then **LUCID** will stop the data flow, and bring up a window asking for the appropriate value. This happens automatically, and the experiment won't continue until the input is supplied. **LUCID** allows you to bring up the input window again, in case it was accidentally dismissed. The user can select the **Supply Input** option to redisplay the window, but the option itself will be inactive if there is not input expected at that time.

The **LUCID** data stream allows comment records to be inserted. Selecting **Add Comments** brings up a text window (not shown) into which the user can enter a comment, and then have the comment record entered into the data stream. This only works when on-line.

The **Clear Outstanding Commands** tells **LUCID** to forget about ever getting a response to a command sent to the **READER** or **LOOKER**. This should only be used when **LUCID** refuses to run new commands for the user. At this point, it is usual that the **LOOKER** or **READER** have failed already.

The *View* Menu

One of the most frequently used features of the system is the ability to display data. The **View** menu has two options which bring up sub-windows, as shown in **Figure 2.8**.

The first, from the **View Data** option, displays a list of variables which can be viewed. These are all variables defined in the **LOOKER**, except regions (see Chapter 4). Four abbreviated menu buttons are displayed near the top of the window, which can be used to limit the variables being displayed. In a large experiment having hundreds of variables, the user might ask that only arrays of two-dimensional histograms be listed, to make the selection easier. Regardless of what variables will be put in the list, the user can select as many of the variables as is needed. These variables will be displayed when the **Display Selected Variables** button at the bottom of the sub-window is selected.

When the user is finished displaying certain variables, their names can be de-selected from the list, and the **Display Selected Variables** button selected again. All of the other variables will still be displayed, but the de-selected ones will be removed.

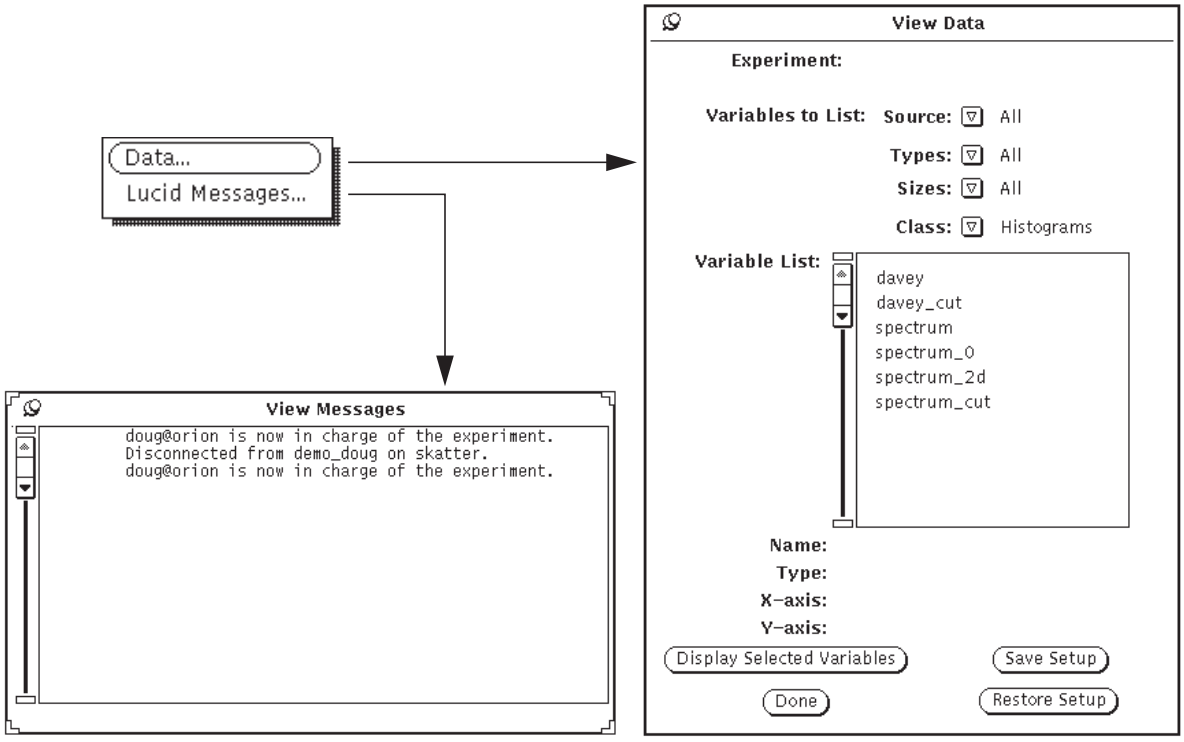


Figure 2.8. Sub-windows from the *View* Menu

The current setup of displayed histogram windows can be saved with the **Save Setup** button. A file is written in the user's current experiment directory which records which variables are currently displayed, their position on the screen, the window size, limits of the histogram displayed and marker positions. These settings can be restored by using the **Restore Setup** button.

Refer to the next section for information about using the histogram windows.

The second option on the **View** menu is for the **View Messages** sub-window. **LUCID** displays several messages here as the experiment proceeds, and it is useful to examine them occasionally. In particular, errors encountered while reading the user's description files will be printed here, along with corresponding line numbers in the description file. A limited number of messages are saved, and the scrollbar on the side of the window can be used to peruse back through them. Typically, the last 200 messages are saved.

The *Properties* Menu

As the user works with the "lucid" program, there are other administrative options which are frequently needed. The **Proper-**

ties menu contains three sub-windows and two other options, as shown in **Figure 2.9**.

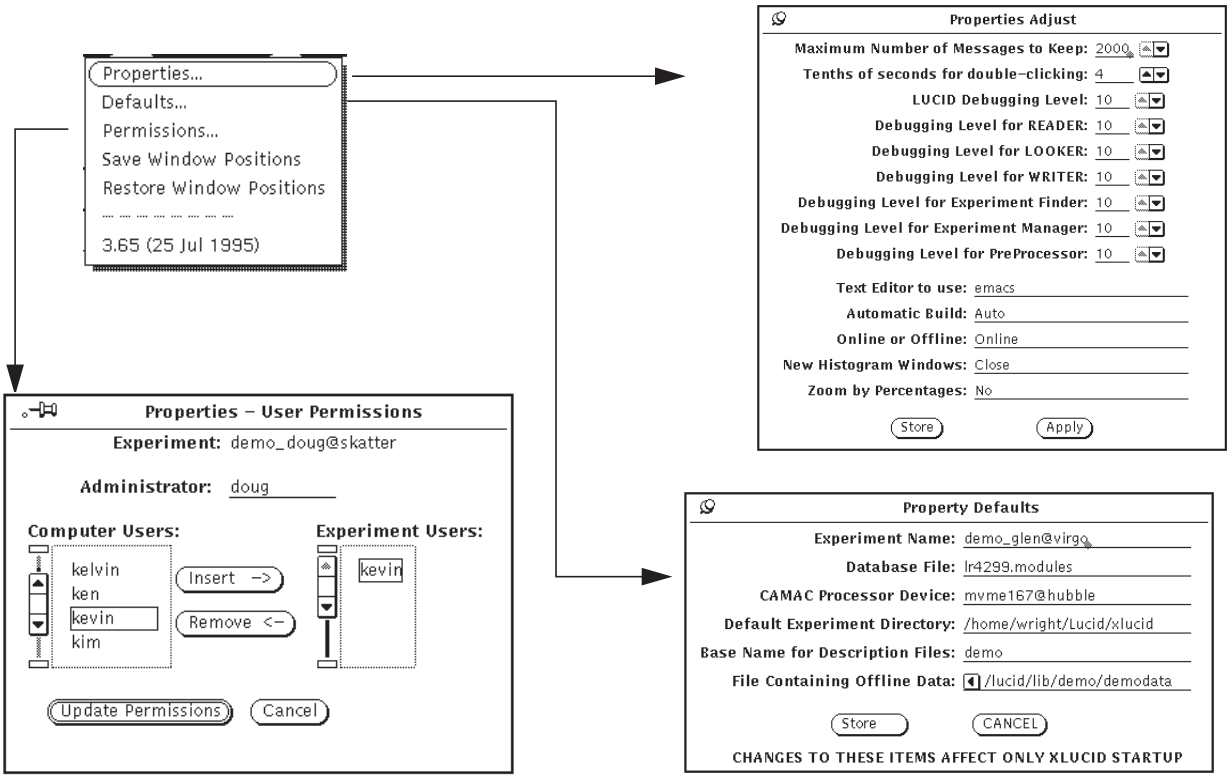


Figure 2.9. Sub-windows from the *Properties* Menu.

The **Properties** and **Defaults** options allow the user to tailor the system in a personal way. The subwindow for the **Defaults** selection allows setting values that only take effect when the **LUCID** program is started:

- **Experiment Name** names the experiment to which **LUCID** will automatically connect when started.
- **Database File** names the file which contains **CAMAC** module information when building. This will be the default when starting, but the user can change it.
- **Camac Processor Device** fills in the appropriate field in the **Build** sub-window.
- **Default Experiment Directory** names the directory to which **LUCID** will change when it starts, but before it connects to an experiment.
- **Base Name for Description Files** tells the system what the description files should start with when building software.
- **File Containing Offline Data** names the default data file to be used when building software.

The fields in the **Properties Adjust** window are effective immediately when changed.

- **Maximum Number of Messages to Keep** determines the size of the memory for the **View Messages** sub-window. It can be set to a very large number, if the user wants to see lots of output messages.
- **Tenths of Seconds for double-clicking** can be set to a large value for users who can't or don't want to double-click the mouse buttons quickly. The value is typically 3 or 4, but 10 would allow a 1 second delay between subsequent clicks. This delay is used in conjunction with the histogram windows, discussed in the next section.
- **LUCID Debugging Level** can be adjusted for 7 different parts of the system. It is used mostly by system administrators or programmers to determine problems within the system. Changing these numbers cause messages to be displayed which describe what the system is doing. The numbers can be thought of as *thresholds,* so that a value of 10 or more allows no extra messages to be printed. A value of 9 will cause a few extra status messages to come up, 8 will bring up even more, and so forth. **BE WARNED** that setting any of these values to a low number will cause a lot of output to be generated. For example, if the debugging level of the "manager" was set to 0, the simple action of building an experiment could generate a dozen pages of output.
- **Text Editor to Use** instructs the system which text editor you want to use when making (or changing) description files. It can be set to **LUCID**, *emacs*, *edt* or *vi*. If *edt* or *vi* is chosen, the system will start a terminal sub-window in which to run the editor.
- **Automatic Build** can be set to true or false; this allows the system to build the experiment automatically when it connects to an experiment which hasn't be built yet.
- **Online or Offline** selects the default state to be used in the **Build** sub-window, and hence the operating mode for the experiment.
- **New Histogram Windows** can be set to **Open** or **Close** to determine the status of newly displayed histogram windows.
- **Zoom by Percentages** affects the **ZOOM** buttons on the histogram window (See **The Histogram Windows** on page 2-23). If set to **Yes**, the windows zoom by a **stepsize** percentage, otherwise the windows zoom by a **stepsize** amount.

The user must select the **Store** button to ensure that the changed properties will stay in effect next time the system is used.

User Permissions

The **User Permissions** sub-window allows the user to check on the names of colleagues which are allowed access to the experiment. The window lists all valid users on the computer systems, as well as the users of the experiment. If the current user is the experiment's administrator (that is, the one who created the experiment), then the list can be changed. The setup allows moving names from the computer's user list over to the experiment's user list. Names can also be deleted from the experiment's user list completely. The creator of the experiment can also name any other user as the administrator by typing the new name into the appropriate field at the top of the sub-window, as seen in **Figure 2.9**. The *anybody* special user name allows any user to access the experiment. This is helpful when there is more than one person involved in running an experiment.

The options to **Save** and **Restore Window Positions** can be used to tailor the look of lucid on the screen, when it is started. Sub-windows should be moved and resized as desired, and then their positions saved. They will be restored to these positions and sizes when lucid is next started. This does not save the characteristics of histogram windows (**The View Menu** on page 2-14 describes how to save histogram windows).

The *Quit* Button

The **Quit** button on the main **LUCID** window is meant to be used to quit the lucid program itself. Ideally, users will start the experiment, monitor its progress for a few minutes, then quit the program while the experiment continues.

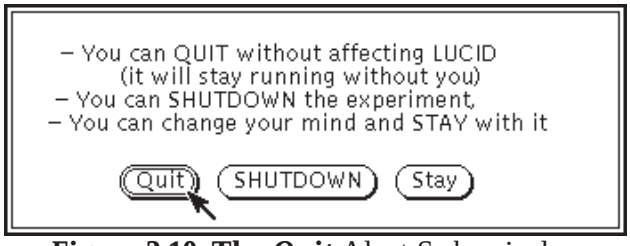


Figure 2.10. The *Quit* Alert Sub-window.

Selecting the **Quit** button brings a request for confirmation from the user.

- Choosing **Quit** is the default choice; the user can simply press *return* to select it. The lucid program itself will quit, but the experiment will remain, whether it is currently running or not.

- The Shutdown button will cause the READER, LOOKER and WRITER to finish the current run if necessary, quit and close down the experiment completely until next time.
- The Stay option tells LUCID to keep working as if nothing has happened.

If lucid is not connected to an experiment, a simple *yes/no* confirmation is asked for.

Control Buttons

The square buttons located on the left of the main LUCID window allow the user to control the data flow, after the experiment software has been built[1].

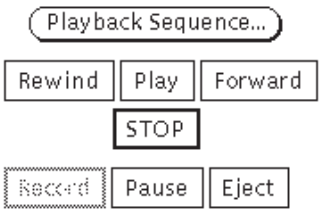


Figure 2.11. The *Control* Buttons

Before connecting to an experiment or building the software, the control buttons will be inactive. Once built, certain buttons may remain inactive. For instance, the *Record* button in Figure 2.11 is inactive because that example represented an off-line session having no WRITER destinations.

Once a control button is selected, a thick edge will be drawn to highlight its border. For instance, the currently selected button in Figure 2.11 is the *Stop* button. On a colour display, the button will have the 3-dimensional appearance of being pressed in.

If a user connects to an experiment in progress, the control buttons will be displayed to reflect the current operation in progress. Also, if the user is “just visiting”, the control buttons will change as the controlling user makes changes.

1. Technically the OpenLook specification refers to these “buttons” as *choices*. They are called buttons here for simplicity.

The *Record* Button

When the user requests that a WRITER be present in the experiment, the *Record* button will be made active. In such a case, the *Rewind* button will *not* be usable, since LUCID wants to write an ordered sequence of runs to the output.

A Run is started when the *Record* button is selected; the current run number is displayed near the top right-center of the main window. Every time the *Record* button is selected, the run number will be automatically incremented.

If no WRITER is present, the *Record* button will be inactive, and the *Play* button can be used to start the next run.

The *Play* Button

The *Play* button can be selected to begin a new run. It will only be usable when data is *not* being saved, such as when analyzing data off-line, or testing electronics or detectors on-line.

If the system is off-line, the *Play* button will remain highlighted until all data has been processed; the *Stop* button will then become highlighted and the *Play* button returned to normal.

If only certain runs are required to be analyzed off-line, the user may select the *Programmed Playback* button.

The *Playback Sequence* Button

It is often useful to replay only certain runs while off-line. The *Playback Sequence* button allows the user to specify various run numbers which should be processed.

The example in Figure 2.12 shows the sub-window which appears when the button is selected. The user must enter the first and last numbers which make up a range of run numbers, in the blank spaces provided, then select the button to add that range to the list. The “range” of run numbers can be a single run number.

Once a list of run number ranges has been made up, the user can select the *Start Programmed Playback* button, the sub-window will disappear, and the *Play* button will become highlighted.

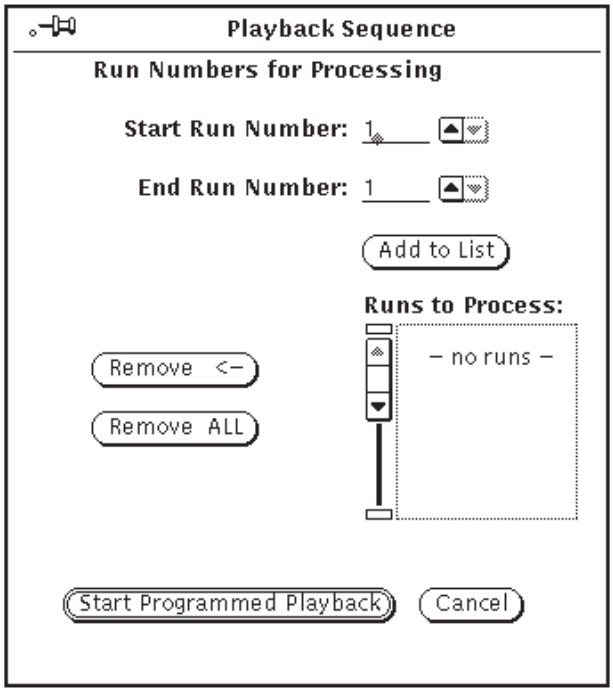


Figure 2.12. The *Playback Sequence* Window

The *Pause* Button

If the user wants to suspend the run temporarily, the *Pause* button may be selected. A blinking message will be displayed across the window to remind the user that the run is suspended.

To resume the run, the user need select the *Pause* button a second time, or re-select the button which was previously highlighted, such as the *Play* or *Record* button.

The *Pause* button will work whether the system is on-line or off.

The *Stop* Button

To stop processing the current run, on-line or off, the user may select the *Stop* button.

The *Rewind* Button

If the current session is off-line, the *Rewind* button may be selected. It simply rewinds the input data file to the beginning. This is true for tapes or regular disk files.If a WRITER is present and active, a warning message is displayed.

The *Forward* Button

The user may skip ahead to the next run in the data stream by selecting the *Forward* button. This button is only usable when the system is off-line.

The *Eject* Button

The *Eject* button is only usable when the system is off-line and the input file represents a tape drive unit. If the *Stop* button is selected, the *Eject* button can be used to rewind the tape and take it off-line from the computer. If the tape is a cartridge type, it will be physically ejected from the drive.

The *Update Visible* Button

The user can update all displayed histograms at once by pressing this button. It is the same as selecting *Update* in each visible histogram window.

Status Buttons

Status buttons appear on the right side of the LUCID window, beneath the Run Number display. They describe the current status of the READER, LOOKER and WRITER.

Figure 2.13. Status Buttons

In Figure 2.13 we see an example in which the READER and LOOKER exist, but the WRITER does not. Further, the READER and LOOKER are both stopped.

Each of the active status buttons may be selected to reveal one of four different messages; each subsequent selection will cycle through the subsequent messages until all four have been displayed. In sequence, they represent:

1. The actual state of the program, such as Stopped, Running, Starting, or Stopped (at End of File).
2. The number of bytes processed so far in the current run.

3. The number of bytes processed so far in the current session. The current session includes all runs since the last *Build* operation.

4. The approximate rate at which data is being acquired, analyzed or recorded.

Figure 2.14 shows some examples having different status labels being displayed.

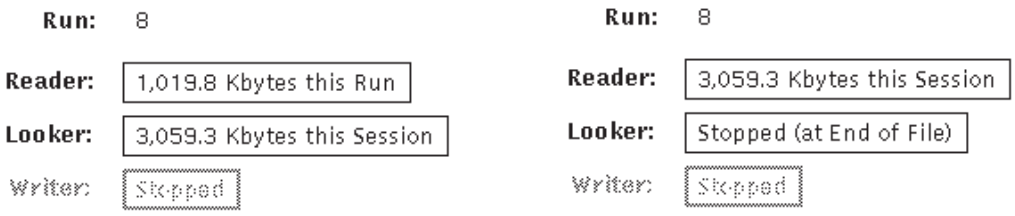


Figure 2.14. Different Status Labels

After a status button has been selected three times, it will cycle the label around to the initial status message. The process of approximating the data rate is done every 5 seconds (roughly), so initially, it might say *Inadequate Statistics* if it hasn't had enough time to make the calculation.

The Histogram Windows

The most frequently used feature of the LUCID program itself is the display of histograms. Histograms are described in detail under LOOKER Histograms on page 4-26, and the user is encouraged to read that section if the topic is not familiar to them.

Histograms are displayed by selecting variable names from the *View Data* sub-window, as described under The View Menu on page 2-14. Each histogram is placed in a separate window, and can therefore be individually resized or modified. They can also be closed to an iconic state.

Each histogram window has a set of four menu buttons which allow manipulation of the data or its representative image. The window depicted in Figure 2.15 show the *File, View, Edit* and *Print* menu buttons which are associated with *every* histogram.

Many features within these menus are used quite often, and LUCID understands several different keystrokes to perform the same functions as various menu selections. Users will most probably use these simple keyboard shortcuts more often than

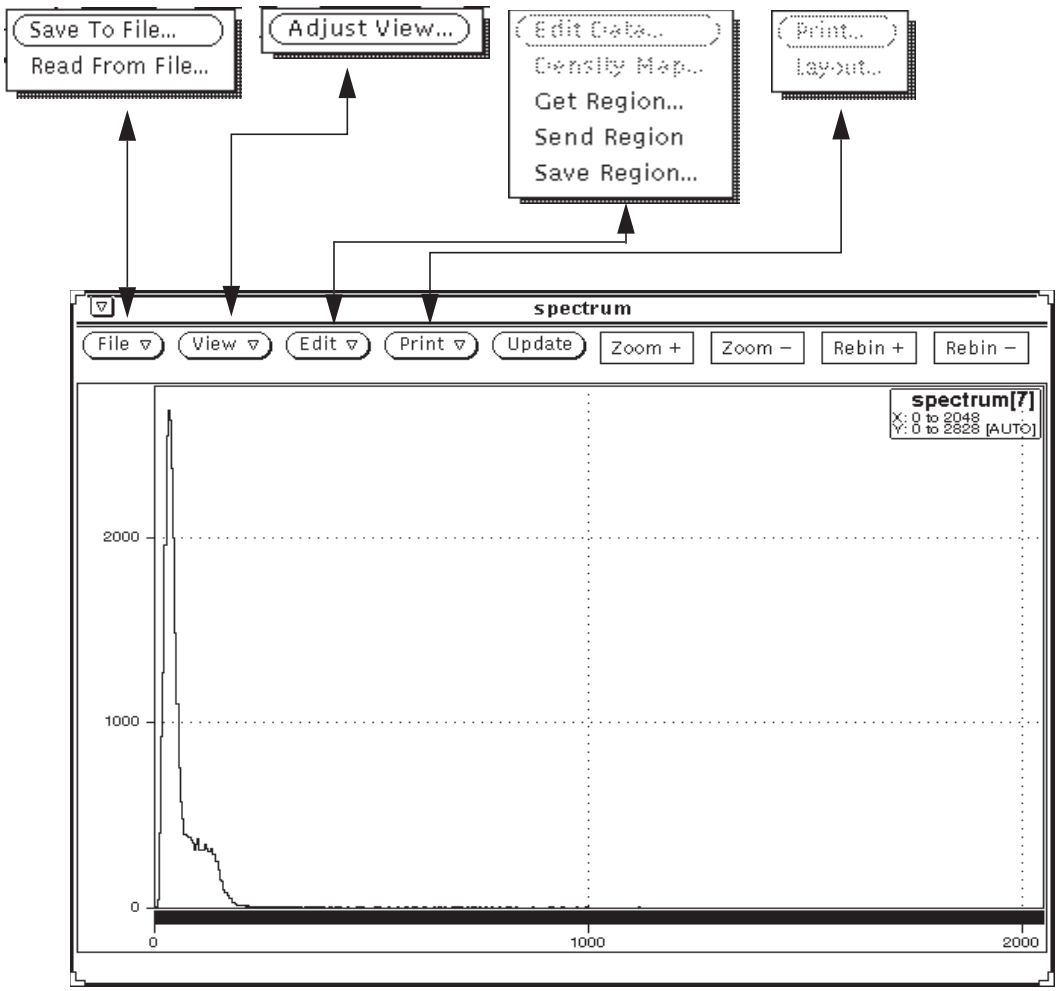


Figure 2.15. Menu Buttons in a Histogram Window

menus and sub-windows. Before discussing the shortcuts, we'll look at the menus.

The *File* Menu Button

Shortcut Key: s

The *File* menu contains two options. The first is labelled Save to File... and allows the user to save histogram data for future reference. When selected, a sub-window is displayed which allows different features to be used, as shown in Figure 2.16.

The user should select the desired options in the Save Data to File sub-window as subsequent use of the save feature uses the same settings, by default, *for all histograms*. The settings are not saved for the next time you run the lucid program.

The first field refers to the name of the file in which data will be saved. By default, it consists of the word Data followed by a period, then the name of the variable. If it is an array, another dot is

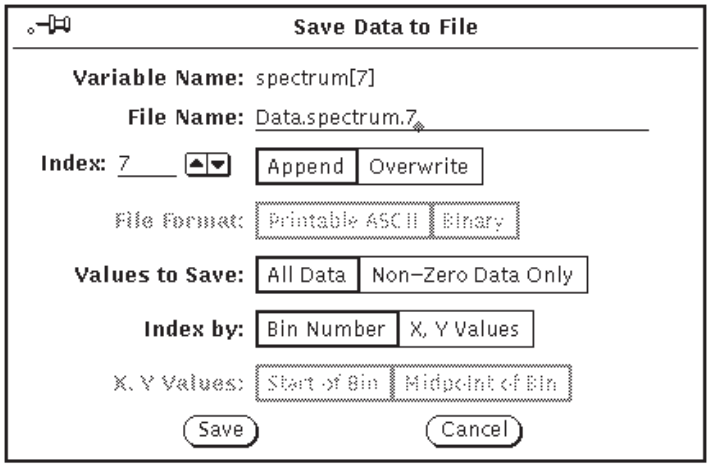


Figure 2.16. The *Save Data to File* Sub-window

appended to the name, followed by the current subscript being displayed.

The next choice which the user can make determines whether the output file will be appended to or overwritten by the saved data. By default, the action of saving data will overwrite the existing file.

The next choice is concerned with the format of the data to be saved. Currently, all data is saved as printable ascii characters, which can later be edited or used as input to other programs.

For 1-dimensional histograms, the X-axis value is printed, followed by a blank space, then the number of counts and a newline character, for every bin that is being saved.

For 2-dimensional histograms, the X-axis value is printed, followed by a space and the Y-axis value, then another space character and the number of counts in the bin, and a newline character.

In either case, the user may save bin numbers instead of X and Y axis values by selecting Display Bin Numbers for the histogram, as described on page 2-30.

The field labelled Values to Save allows the user to compress the amount of saved data by *not* saving data points which have zero counts. The default option is to save non-zero data only.

The user may restrict the amount of data to be saved by using the final option, labelled Amount to Save. By default, data from every bin in the entire histogram will be saved, assuming the previous option, Non-Zero Data Only is selected.

If the user opts to save Selected Bins Only, then only data associated with bins between the histogram's markers will be saved. Markers will be discussed momentarily, but if only one marker is present, then that single data value will be saved, according to the other criteria selected on this sub-window.

Once all the desired options have been chosen, the user may select the Save button, and data will be stored.

Read From File

The second option within the *File* menu is labelled Read from File... and allows the user to re-read histograms previously saved by the Save Data To File sub-window, as shown in Figure

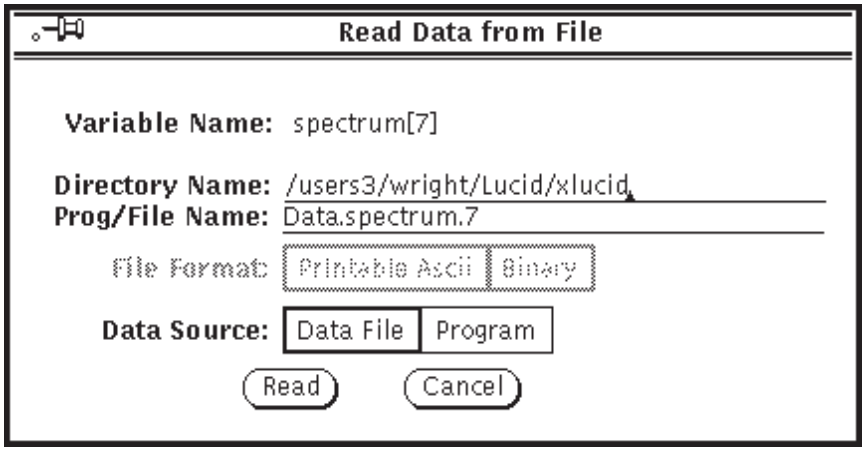


Figure 2.17. The *Read Data from File* Sub-window

2.17. The Directory Name field defaults to the current working directory. This field is included as it is quite possible that previously saved histograms are stored in a directory other the current experiment directory. The default for Prog/File Name field is the word Data, followed by a dot and the name of the histogram being read. If the histogram is an array, the current index is appended to the name. Selecting a data source of Program expects the Prog/File Name field to contain the name of a program. The program will be executed to generate the histogram data to be read.

The Read button causes the histogram to be read. Cancel removes the window without making any changes to the named histogram.

Shortcuts for the *File* Menu

The user can press "s" to save histogram data. The Save To File sub-window will not be displayed, so the user should select the appropriate options first. The user should remember that data for the histogram *currently being pointed to* will be saved.

The *View* Menu Button

The *View* menu contains a single option labelled Adjust View, and allows the display to be tailored in a variety of ways. The Adjust View sub-window is shown in Figure 2.18.

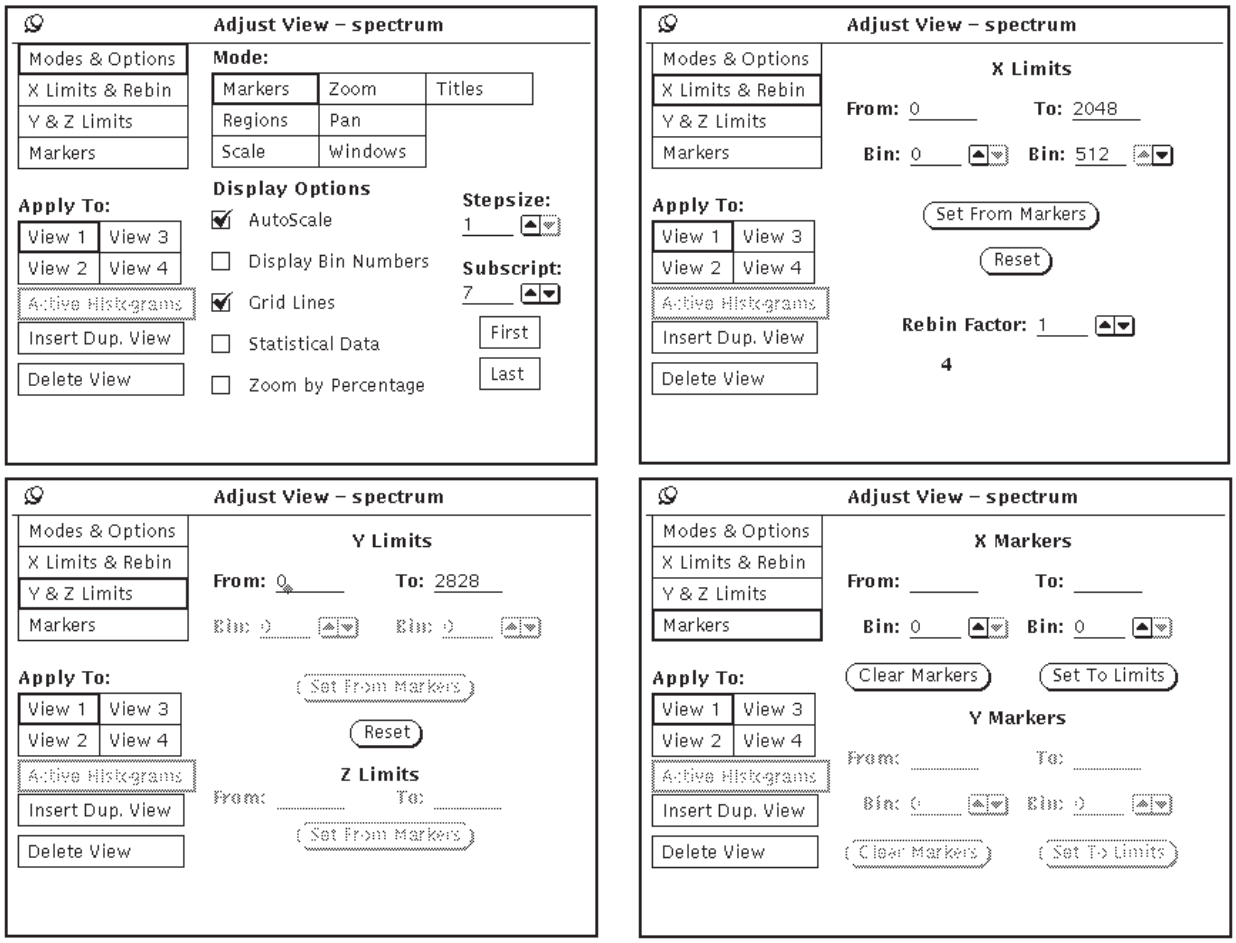


Figure 2.18. The *Adjust View* Sub-window

While it looks rather intimidating, the sub-window allows the user to go through simple and logical steps to adjust the display. Depending on the selection made in the upper left hand corner, one of the four sets of controls are displayed.The user should keep in mind that the data is *never* being changed when these settings are adjusted; the *view* is changing, and nothing else. Although this window can be called up by the View selection on any histogram window, there is never more than one Adjust View window on the screen at any given time.

Markers

Before discussing features of this sub-window, it is important to understand *markers.* Markers are simply two limiting lines drawn over bins in a histogram display. Many of the options for

adjusting the display use markers. When drawn on the display, a marker occupies the same width as the bin over which it is placed. Markers are discussed in more detail on page 2-32, and again on page 2-35.

In the Adjust View sub-window, the first set of choices allows the user to attach different meanings to mouse actions and keystrokes. By default, the system is in *Marker Mode*, as described below. Using the mouse buttons and keypad is discussed in more detail under Shortcuts for the View Menu on page 2-32, but it is important to realize that different modes will cause the system to react differently to commands.

- Marker Mode causes the mouse pointer and buttons to affect the markers on the screen. The keypad on the right side of the keyboard also becomes used to locate markers in this mode. This is the most common mode for interacting with the display.
- Region Mode causes the user input to adjust *Regions*, which are discussed under Histogram Regions on page 4-33. Briefly, adjusting regions on the display will allow your analysis to perform different instructions. At the time of writing this document, the Region Mode is operational for 1-dimensional histograms only.
- Scale Mode causes the mouse pointer and other controls to change the *scale* of the displayed data; for 1-dimensional histograms, user actions will adjust the height of the bin data (the Y-axis) being displayed.
- Zoom Mode allows the user to use mouse buttons and keystrokes to affect the viewed limits of the display, and effectively enlarge a particular area of interest.
- Pan Mode causes the mouse and keypad to change the area of the histogram being viewed. The user is allowed to easily “pan across” the data.
- Window Mode (or Border Mode) is not currently implemented.
- Title Mode allows the user to adjust the position of the histogram title within the display, or add new titles.

Using these different modes is described in more detail on page 2-34.

Histogram Sub-Views

Each histogram window can be split into several sub-views of the same data. Each sub-view is independently controllable, so that the user can view different parts of the same histogram, as shown in Figure 2.19. As mentioned below, the user can also view different subscripted histograms of the same array using sub-views.

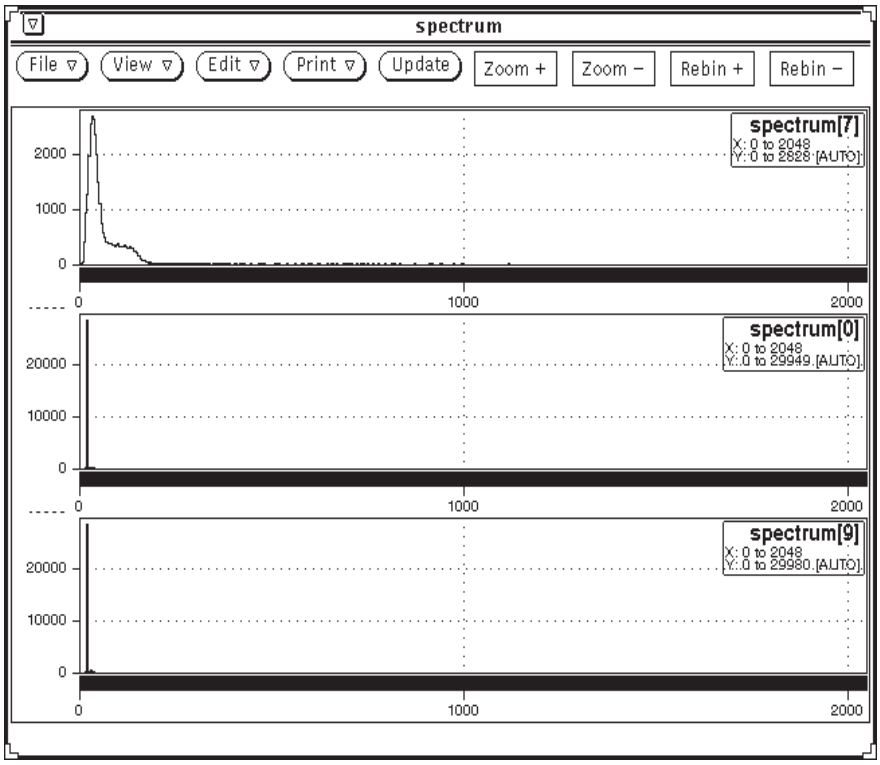


Figure 2.19. Multiple Sub-Views of the Same Histogram

The remainder of the fields in the sub-window are applied to a particular sub-view. The next set of choices is titled Apply To, and allows the user to determine which sub-view to adjust. The user won't be allowed to select a sub-view which doesn't exist.

To add another view, or delete an existing one, the user can simply select Insert Duplicate View or Delete View, located farther down in the sub-window. By selecting a current view first, a duplicate copy of it will be inserted, or when deleting, the chosen sub-view will be removed.

Display Options is a rather generic title, but these choices allow a variety of useful changes to be made. The various choices may be made by selecting the box, and having the check-mark appear or disappear.

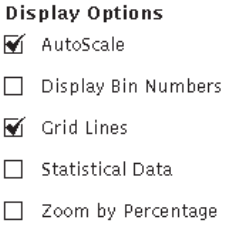


Figure 2.20. Display Options

Shortcut Key: a

- AutoScale causes the current histogram display to be automatically scaled to allow the data to fit. For 1-dimensional histograms, the Y-axis is adjusted so the data will fit visibly within the window. For 2-dimensional histograms, AutoScale will adjust the settings of the Z-axis. This option is selected by default.

Shortcut Key: g

- Grid Lines may be selected to have dashed lines drawn across the axes at regular intervals. This option is selected by default.

Shortcut Key: b

- Display Bin Numbers will cause *bin numbers* rather than actual X or Y axis values, to be printed on the display. As mentioned under The File Menu Button on page 2-24, this option will also affect the values saved to a file.

Shortcut Key: S

- Statistical Data will cause more information to be displayed under the title. Regardless of this option, the sum of all bins between the markers is printed when markers appear. When the option is selected, other information such as the maxima and minima values and the mean are displayed.
- Zoom By Percentages is used in conjunction with AutoScale and Zoom mode. See Zoom Mode on page 2-41 for a description of how this option affects zooming.

The Stepsize option allows the user to enter a scaling factor. When a request is made to move a marker or adjust the viewing limits, this step size will be used. By default, the step is 1, so that markers are moved 1 bin at a time, and so forth. The step size function is discussed again on page 2-33.

The Subscript field allows the user to change the subscript of a displayed histogram array. This field will be inactive if the associated histogram is not an array.

Zoom In or Out

The next choices in the sub-window allow the user to change the limits of the viewed data, and to enlarge or shrink the viewed area. Figure 2.21 shows some typical values in the various fields.

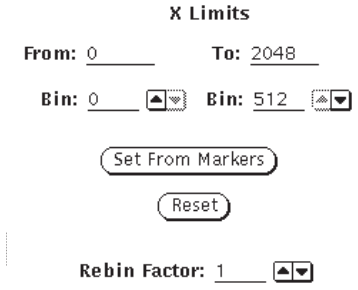


Figure 2.21. Adjusting Viewable Limits

The X Limits may be set to any values within the extents of the histogram, and the area delimited by these values will be redrawn to fill the entire window. Bin values may also be set, or adjusted by selecting the up-down arrow buttons; again the area enclosed between these values will be redrawn to fill the window. These values apply only to the current sub-view, if more than 1 exists. If a value is entered for X or Y axis values, it will be rounded to the nearest bin boundary.

The button labelled Set From Markers may be selected to allow zooming in to the area enclosed between markers on the display. For the Y-axis limits of a 1-D histogram, the button labelled Set From Markers performs a slightly different function: the extents of the Y-axis are set to match the extent of the data values in the area between the markers. This essentially sets the Y-axis limits so that data between the markers will fill the display.

When a 2-dimensional histogram is being viewed, the Y-axis limits button labelled Set From Markers will perform exactly the same function as the button does for the X-axis. Further, the button labelled Set From Markers for the Z-axis is usable, and will set the Z-axis limits to the limits of the data contained within the marked area.

If the user adjusts the Y limits for a 1-D histogram, or the Z limits for a 2-D histogram, the Autoscale option is turned off automatically. Values below the from: limit are drawn as being at the from: limit, and values above the to: limit are drawn as being at the to: limit. Set From Markers for these axes will copy the high value from within the area delimited by the markers.

The Reset buttons allow the user to easily reset the limits to their default, full scale values.

Rebinning

The Rebin Factor provides a simple dynamic integer rebinning facility for XLUCID. The bin width indicates the number of X axis bins that are summed to obtain the bin height. When rebinning, the actual bins displayed will be adjusted to start the first bin on an even multiple of the rebin factor. For example, with a histogram displayed with no rebinning, and the first displayed bin being 12: if a rebinning value of 5 were used, then the histogram would now start at bin 2 which would be made up of the original bins 10 to 14. The display is also adjusted so the last bin is made up of a number of original bins equal to the rebinning factor, and the last displayed bin would include the last shown bin number from the original histogram display.

Currently, when histograms are *scaled*, *panned*, or *zoomed*, the calculations for the start and end positions are based on the original

bin numbers. Setting the step size to the rebin factor will make these functions behave as if they had a bin size of 1.

Rebinning is not yet available for 2-dimensional histograms.

Adjusting Markers

The final set of values in the sub-window allows positioning the markers within a sub-view. Values are entered for them in the same way as for limits. Any value within the extents of the histogram may be used, but the system will round the value, if necessary, to a value on a bin boundary. Using the Bin fields will avoid confusion, since they are always whole numbers and always aligned on a bin boundary. Marker values need not be within the current limits for X / Y axes, they will be positioned on the correct bins, but just not visible on the display.

Pressing Clear Markers simply removes the corresponding X or Y markers from the display. Double-clicking on a histogram in Marker mode clears both the X and the Y markers.

The Set To Limits button allows the user to set markers at the current limits of the *viewed data*. That is, markers will be placed on the first and last bins which are currently visible in the sub-view.

Representing 2-D Histograms Visually

With a 2-dimensional histogram, markers appear visually as a pair of lines, one marking an X-axis bin and the other marking a Y-axis bin. The result is that only 1 bin is selected, as represented by the intersection point of these two lines. An area between two markers is therefore the rectangle enclosed by the four lines, as shown in Figure 2.22. The data within the bins of a 2-dimensional histogram are represented on the screen as smaller rectangles which are filled to various degrees. For instance, if a bin contained as much or more data as specified by the upper Z-axis limit, then the bin's rectangle would appear to be filled. If the bin contained very little data as compared to the lower limit, then the bin‘s rectangle would be empty. A rectangle which contains a filled area of 50% means that the bin contains roughly half the data which the Z-axis limits include.

Shortcuts for the *View* Menu

There are several shortcuts which the user can make, allowing quick adjustment of the view. First, the four display options can be changed by pressing single characters on the keyboard.

AutoScale

An upper or lower-case “a” will turn on the autoscale option. Another one will turn the option off.

Display Bin Numbers

Pressing an upper or lower-case “b” will enable or disable Display Bin Numbers in a similar fashion.

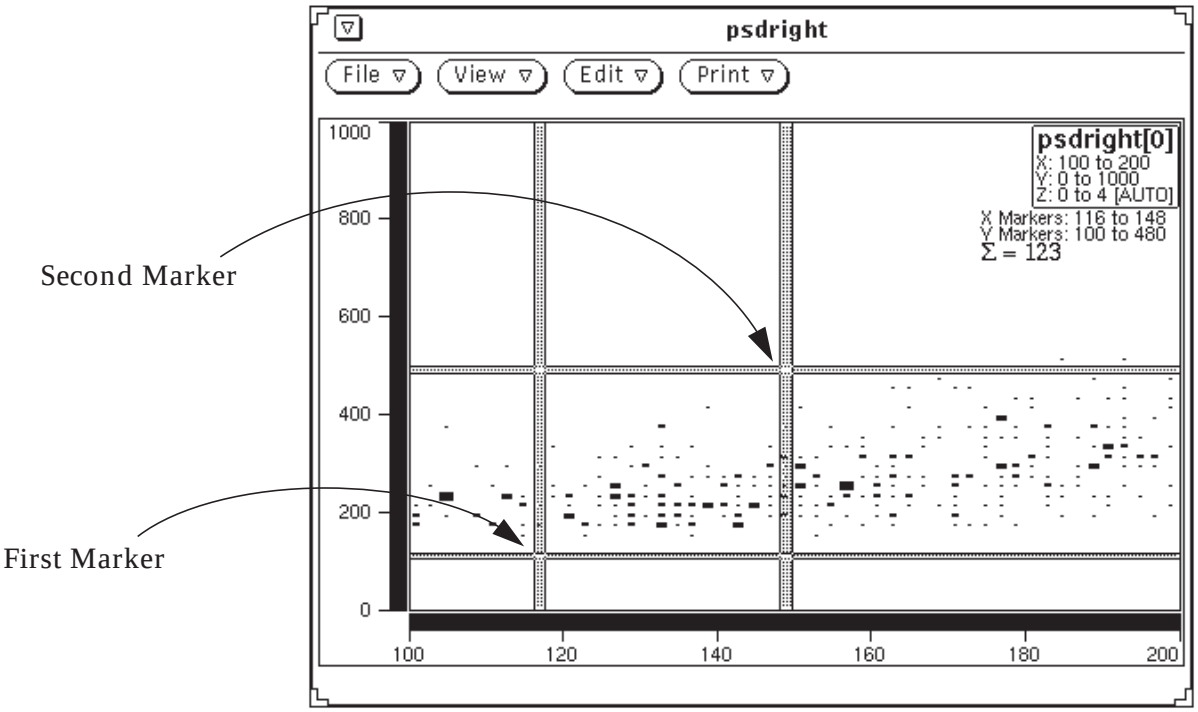


Figure 2.22. Markers on a 2-Dimensional Histogram

Grid Lines

Accordingly, typing a “g” will draw a grid of dashed lines over the display, and pressing it a second time will erase them. A capital “G” will perform the same function.

Statistical Information

A capital “S” will toggle the display of statistical information. A lower-case “s” will *not* do the same; it causes data to be saved, as explained under The File Menu Button on page 2-24.

Update the Display

Pressing the “u” will cause the display to be updated with new data. An upper-case “U” will do the same thing.

Changing the Stepsize

The Stepsize function can easily be changed by simply typing a number. The step size is used as a factor for most operations, as mentioned on page 2-30. For instance, if the user normally presses some keystroke to move a marker 1 bin to the right on the display, it would move 10 bins after the stepsize has been applied. The user must remember to use the digit keys along the *top* of the keyboard, and *not* the numeric keypad, since they have different meanings!

A temporary step size can be used by holding down either *Shift* key on the keyboard. So long as it is held down, a step size of 5 times the entered stepsize will be in effect. The *Control* key can be held down to produce a step size multiplier of 10. If both the *Shift* and *Control* keys are held down at once, then a multiplier of 50 will be used.

Select the Current Mode

The most useful shortcuts involve adjusting the view, by using the mouse buttons in conjunction with the numeric keypad on the right side of the keyboard. As described on page 2-28, the current *mode* of the program will affect how certain keystrokes are interpreted. The mode of the program may be changed quite easily by selecting various function keys, located along the top of the keyboard. None of the keyboard shortcuts mentioned thus far are affected, but the shortcuts involving the view, as mentioned below, depend entirely on the current mode.

Rather than selecting the current mode from the Adjust View sub-window, the user may press the following function keys:

Mode	Function Key
Marker	F2
Region	F3
Scale	F4
Zoom	F5
Pan	F6
Window	F7
Title	F8

When a mode is selected, a reminder will be displayed in the lower left corner of the histogram window currently being pointed to. The mode remains in effect *for all histograms*, until another mode is selected.

The user may also specify a temporary mode change, so that only the current keystroke will be performed in the desired mode. For example, the user will most often work in marker mode, but by performing an extra keystroke, zoom mode can be enabled temporarily to perform one command. Temporary mode changes will be discussed near the end of this section.

Some keys on the numeric keypad keep the same meaning regardless of mode; we'll consider them first. Figure 2.23 shows the layout of the keypad.

Insert or Delete Sub-Views

The insert and delete keys at the bottom of the keypad will insert and delete sub-views from histograms, respectively. The function of these buttons doesn't change from one mode to another; they can always be used to change the number of sub-views. The mouse pointer must be pointing to the histogram sub-view which is to be duplicated or deleted.

Marker Mode

Shortcut Key: F2

This is the most commonly used operating mode, and is the default when the program starts.

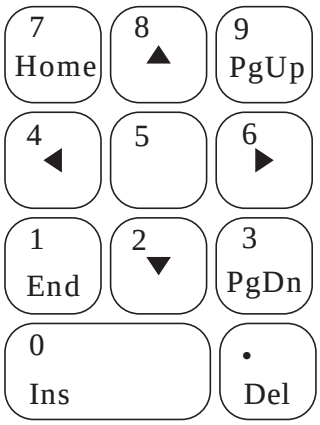


Figure 2.23. Layout of the Numeric Keypad

Markers may be placed by simply *selecting* a bin in the histogram display; the first marker must be placed by clicking the left-most mouse button, while pointing at the appropriate bin. While hold-

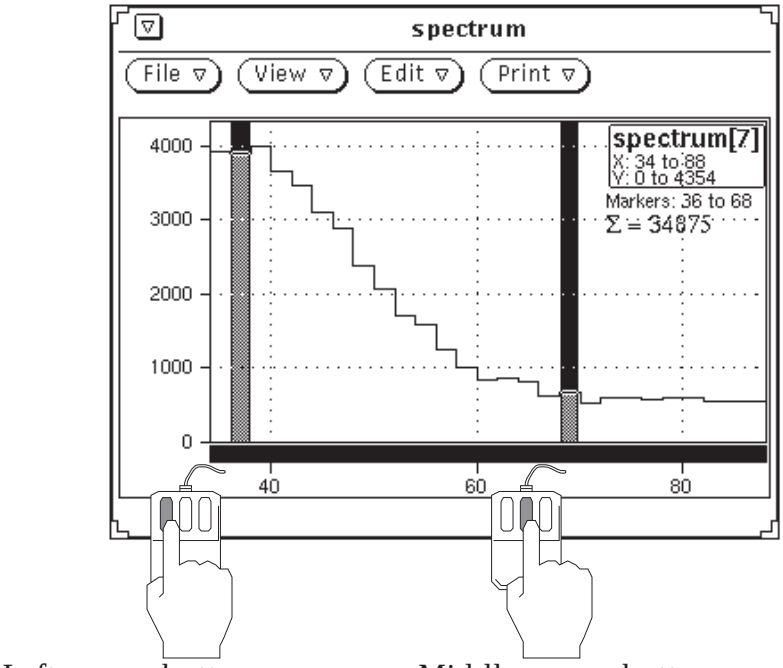


Figure 2.24. Placing Markers with the Mouse

ing the mouse button down, the pointer may be dragged through various bins until the correct one is highlighted. Once positioned, the title of the histogram will display the position of the marker. If bin numbers are being displayed (see DISPLAY OPTIONS on page 2-29) the bin number containing the marker will be shown instead.

Accordingly, the middle mouse button is used to place the second marker. It can be dragged, as the first one. Once a second marker is positioned, the histogram's title will show the sum of counts of the bins surrounded by the markers. Double-clicking the left mouse button will erase both markers. Figure 2.24 shows an example of placed markers in which the markers are drawn to the current width of the bin on the screen; this particular example has been zoomed in.

The keypad may be used to fine-tune marker placement. Figure 2.25 shows the additional functions available on the keypad while the program is in marker mode. Recall that *Stepsize* is normally 1, and the up and down arrows have no effect on 1-dimensional histograms.

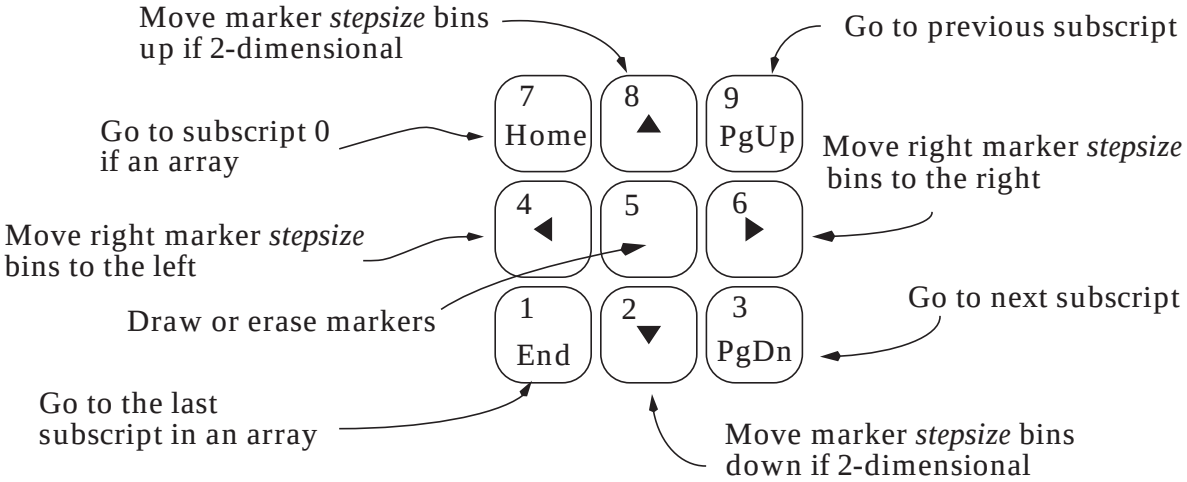


Figure 2.25. Keypad Functions in Marker Mode

If the right or left arrow keys are pressed so that the marker would be positioned outside of acceptable histogram limits, the marker does not wrap around, but stops at the limit.

The center key, 5, is quite useful. In most modes, it means "start with markers". In marker mode, the user can press it once to position the first marker over the leftmost bin currently shown on the display. A second press will position the second marker over the rightmost bin being shown. This is a quick way to sum all bins on the display at any time, since the sum of counts between markers is displayed after the second marker appears. Pressing the "5" a third time erases both markers.

To review for a moment, the user could create a marker and position it ten bins from the left side of the histogram display by pressing the center "5" key on the keypad, followed by *control*-right arrow; that is, hold down the control key while pressing the keypad "6".

The Home and End keys will cause the first and last subscripted histograms in an array to be displayed, respectively. Pressing the Page Down (PgDn) key will cause the next subscript to be displayed. For instance, if spectrum[0] is being displayed, the Page Down key will cause spectrum[1] to be shown. The user can think of this as "going down through the array", since we usually visualize an array as a vertical stack of subscripted items. When the last subscript is reached, pressing the Page Down button again will cycle the subscript back up to 0.

Correspondingly, the Page Up button (PgUp) will "go up in the array"; the previous subscript will be shown until 0 is reached, then wrap around to the bottom of the array again.

Region Mode

Shortcut Key: F3

The arrow keys can be used to position limits of *regions*, in much the same way as markers.The most apparent difference between regions and markers is multiple areas for regions, requiring a concept of a *selected region area*. The selected area is the only part of the current region that is affected by keyboard or mouse commands.

Region mode is only useful if a region has been attached to the current histogram by the Get Region... menu selection. The bins included in the region of a 1D histogram are indicated by shading the background of the histogram. The region bins in a 2D histogram are outlined. The selected area is "ghosted" to provide visual differentiation.

1D Regions

Adjusting a region can be done completely with the mouse pointer and buttons. Clicking the left mouse button once on an area *selects* that area. The selected area can be dragged left or right by holding the left button down after making a selection, and then moving the mouse left or right. If an area has been selected, pressing the middle mouse button adjusts the left edge of the area to the current bin, and the right mouse button adjusts the right edge of the area to the current bin. Double clicking the left mouse pointer does one of four different things, depending on where the pointer is. If the pointer is between the markers, the bins between the markers (inclusive) are added to the region if not all the bins are currently in the region. If all bins between the markers are already part of the region, the bins between the markers are removed from the region. If the pointer is outside the markers, and the bin under the pointer is not in the region, the bin is added to the region. If the pointer is over an area outside the markers, the area is removed from the region.

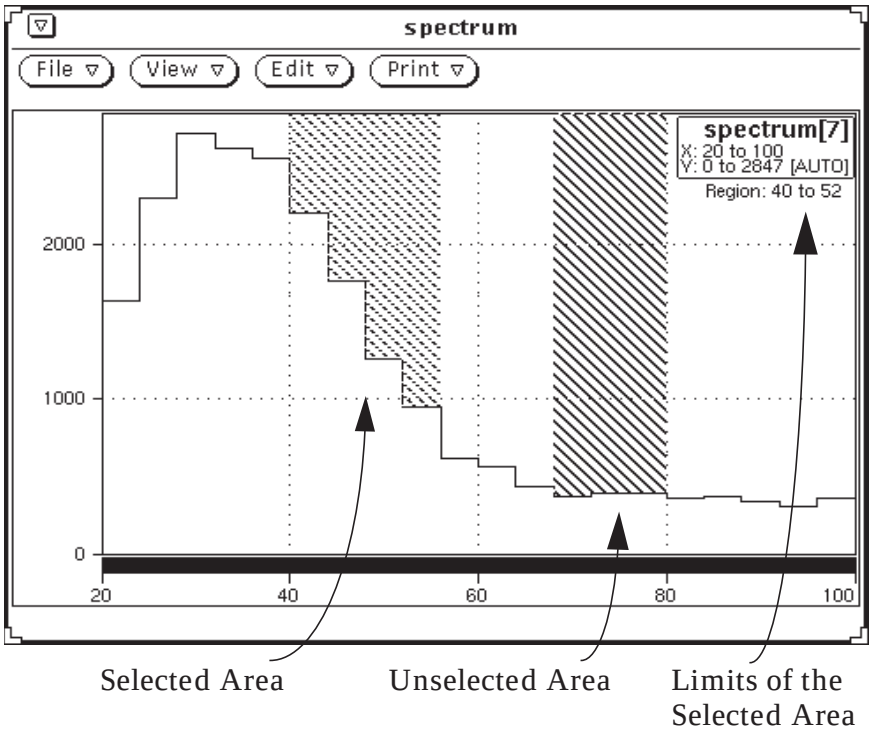


Figure 2.26. Viewing 1D Regions

2D Regions

With 2D regions, there are three types of objects to work with: se-

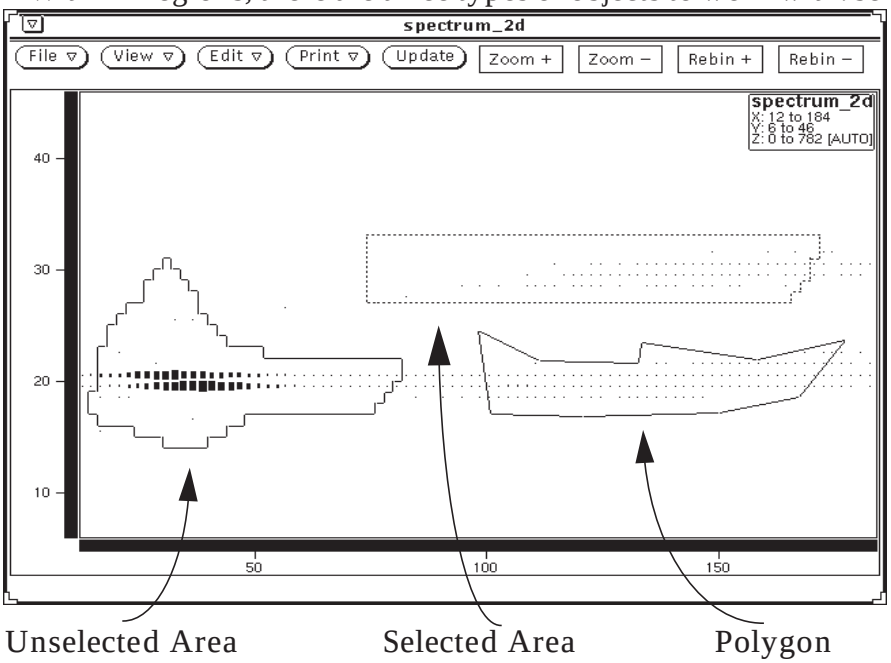


Figure 2.27. Viewing 2D Regions

lected areas, unselected areas, and polygons. An *area* is a group of one or more adjacent bins, and the outline follows the border of the bins. *Polygons* are user-drawn lines, and go from screen point to screen point, ignoring bin boundaries; see Figure 2.27.

Polygons are drawn on the screen to fill or empty an area. Drawing a polygon is done by pressing the SELECT button when the pointer is outside of any region area, then moving the pointer to the next location and pressing again, and continuing until all points have been selected. Pressing the 5 key on the numeric keypad causes the area within the polygon to become a *selected* area. Pressing 5 a second time causes the area to be cleared. If a mistake is made when drawing, the ADJUST button erases the current polygon.

Unselected areas cannot be modified directly, but must be changed into a selected area by placing the pointer over it and pressing the SELECT button.

A *selected* area can be moved around the screen with the four arrow keys. If the selected area is shifted beyond the limits of the histogram, the edge of the selected area will *fall off!* A selected area can also be dragged by selecting an area and not releasing the SELECT button, then moving the mouse. Releasing the SELECT button stops the dragging. Be warned that the screen updates more moving an area can be quite slow, and it can take many tens of seconds for XLUCID to catch up with drawing the dragged area!

The CUT key deletes all selected areas. The COPY key creates an unselected area at the same location as each selected area. The PASTE key unselects all selected areas.

Send Region

Changes made to a region do not take effect in the looker until Send Region is selected from the histogram menu.

Regions can also be adjusted using the keypad, although this currently provides less functionality than the mouse. Keypad functions for 1 dimensional histograms are shown in Figure 2.28, and keypad functions for 2D histograms are shown in Figure 2.28.

Scale Mode

Shortcut Key: F4

Scale mode can be used to zoom in on the data. This mode is the same as zoom mode for 1-dimensional histograms. For 2-dimensional histograms, scale mode allows the user to adjust the scale of the data; that is, to adjust the Z-axis. Figure 2.30 describes the functions available from the keypad while in scale mode.

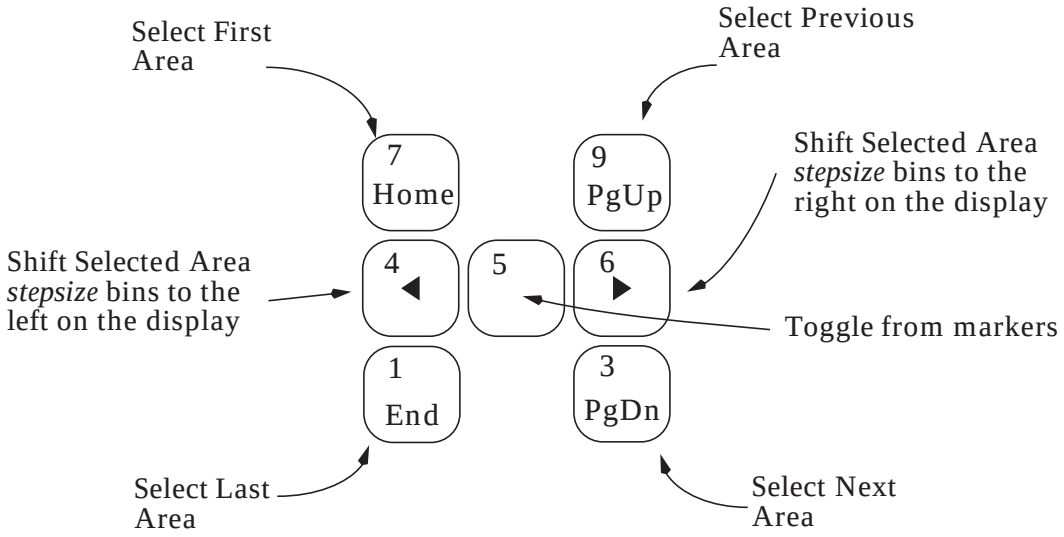


Figure 2.28. Keypad Functions in Region Mode - 1D

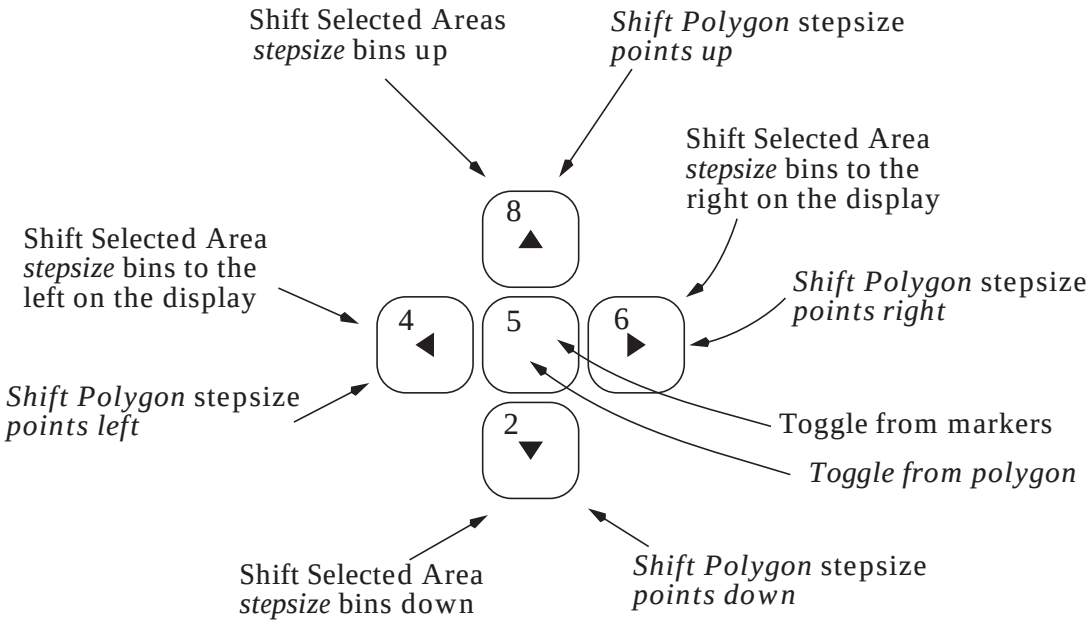


Figure 2.29. Keypad Functions in Region Mode - 2D

This represents the most difficult array of functions to use in the entire program, because it is inherently difficult to visualize Z-axis transformations.

The center “5” key is interesting because it allows the user to select an area with markers, then set the Z-axis data limits so that the marked data fits exactly within.

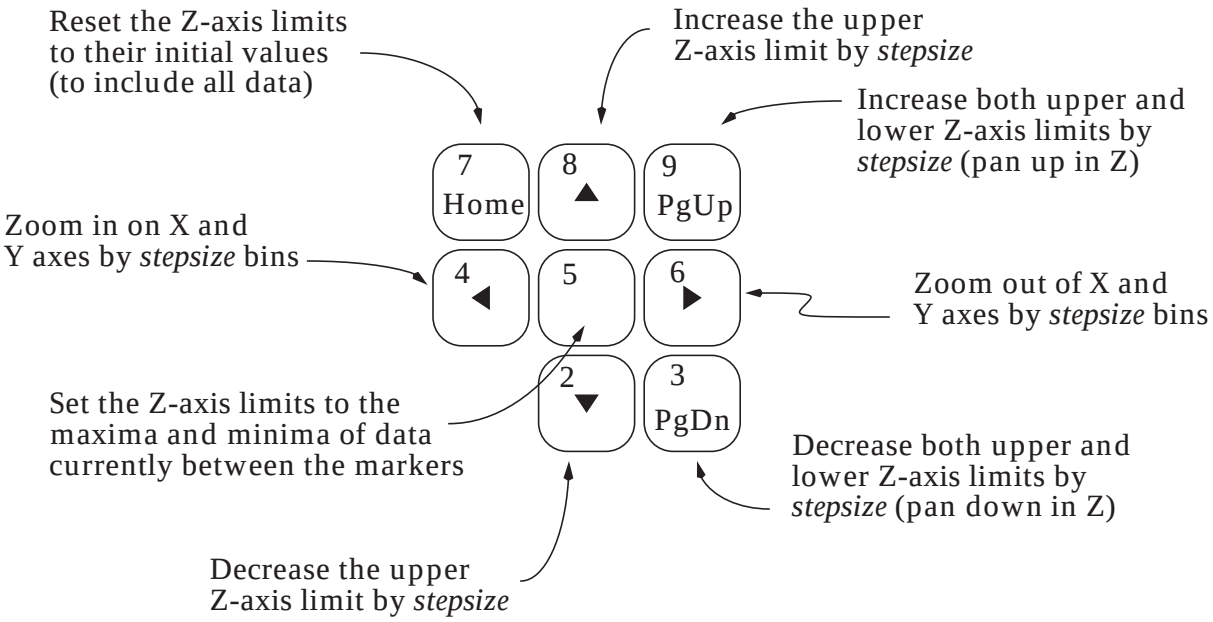


Figure 2.30. Keypad Functions in Scale Mode

The page up and page down buttons are also useful; they move the Z-axis limits up or down by the stepsize, allowing the user to pan through the data in the Z direction.

Zoom Mode

Shortcut Key: F5

Although the user may run Zoom mode in the same way as other modes, it is usually most convenient to simply use the temporary zoom mode described on page 2-43. The numeric keypad takes on different meanings when in zoom mode, temporary or not, as described in Figure 2.31.

Again, the center button is most useful; the user may remain in marker mode to select a marker area, then press left-meta-5 to zoom in on it. The user should keep in mind that the stepsize determines the amount of zoom that is performed. For instance, *shift-left arrow* will zoom in on the X-axis by 5·*stepsize* bins.

With 1-Dimensional histograms, zooming in and out on the Y-axis can be done by either *stepsize* bin units or by *stepsize* percent increments. Selecting the Zoom by Percentage and AutoScale options on the Adjust View window sets the autoscaling display to be a percentage of maximum. The default displayed percentage is 100%, this can be reduced to 1%. The important point is that the percentage does not change when the histogram is up-

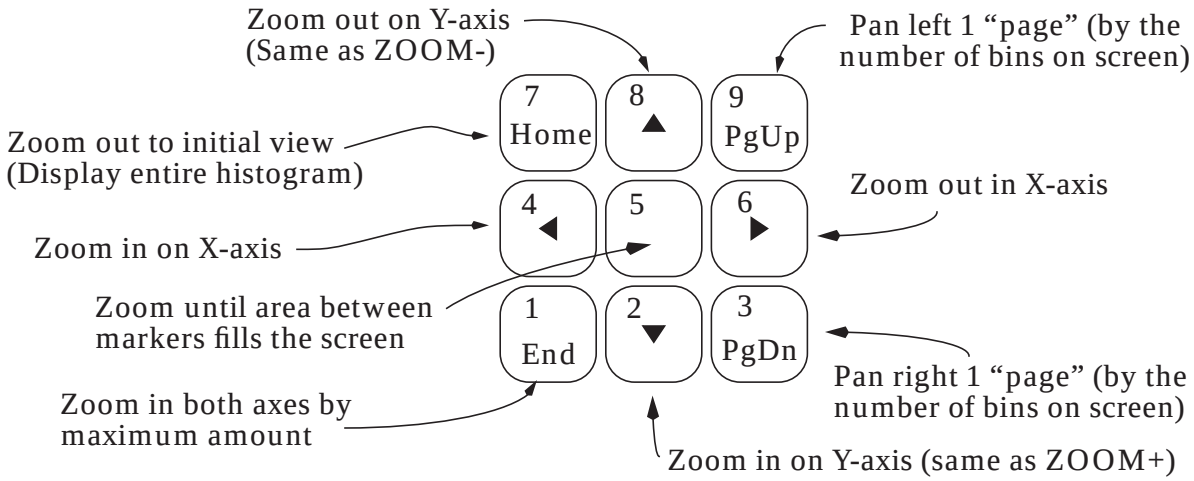


Figure 2.31. Keypad Functions in Zoom Mode

dated, rather the range changes to maintain the same percentage display of the available data.

Note the use of Page Up and Page Down buttons to "scroll" across the histogram, once zoomed in. The number of bins displayed on the X-axis is taken as a "page" and the Page Down button will pan across the histogram, a screenfull at a time.

The user should keep in mind that the mouse buttons have different meanings in zoom mode; by dragging the mouse while the left button is pressed, the display will zoom in or out depending on the direction of motion.

Pan Mode

Shortcut Key: F6

Pan mode refers to simple displacement of the histogram image, and is only useful when zoomed into some area of the display.

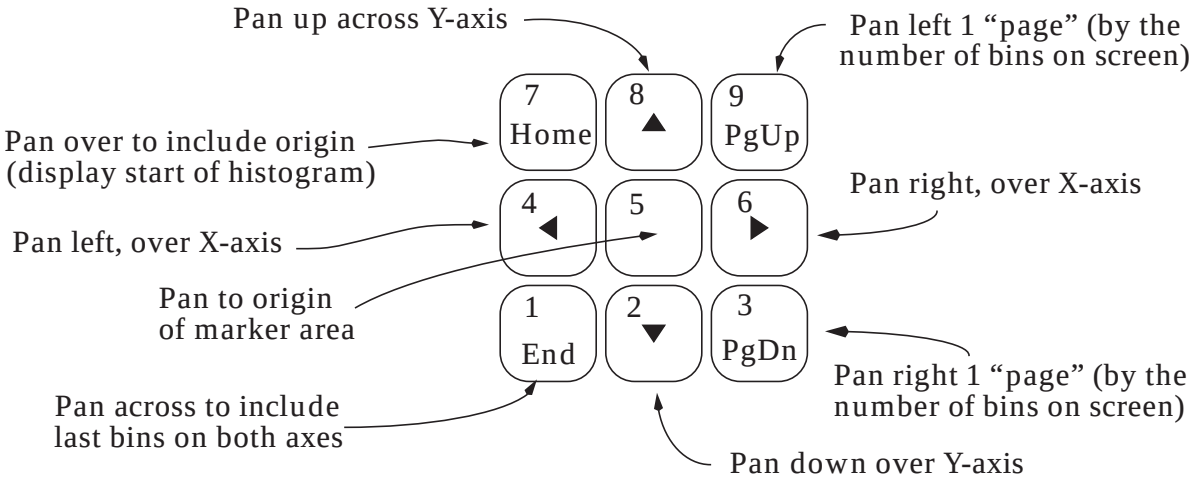


Figure 2.32. Keypad Functions in Pan Mode

Figure 2.32 describes the functions available in panning mode. As with zoom mode, panning across a display is most commonly performed with the temporary mode keys, as described on page 2-43. The arrow keys will shift the view in the direction of the arrow by the stepsize amount (*not* by the size of the pan window). The Home button causes the lower left portion of the histogram to be displayed, the End button causes the upper right portion of the histogram to be displayed. The Page Up and Page Down buttons work the same for both 1-dimensional and 2-dimensional histograms: the histogram is shifted left or right by they number of bins on the page.

The display can be *dragged* using any mouse button. As might be expected, the histogram will follow the movement of the mouse when a button is pressed inside the histogram and the mouse is moved left or right.

The user should keep in mind that the stepsize determines the amount of panning that is performed. For instance, *shift-left arrow* will move the display 5·*stepsize* bins to the left.

Temporary Scale, Zoom and Pan Modes

The user is allowed to temporarily go into scale, zoom or pan modes by holding down a *modifier key*, as shown in Figure 2.33, regardless of the current mode. A modifier key acts like a *shift* or *control* key; the user must hold it down while some other key is pressed. For example, the temporary scale mode is identical to regular scale mode describe above, except that it is in effect so long as the user holds down the right *meta* key. Therefore, *right-meta* Home would reset the Z-axis limits, regardless of the current program mode.

Typically, keyboards have 2 meta keys immediately on either side of the space bar, as shown in Figure 2.33. The figure shows

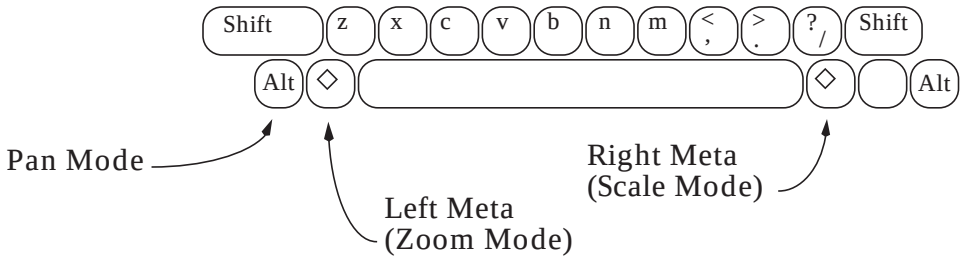


Figure 2.33. Modifier Keys for Temporary Modes

that temporary zoom mode is enabled by holding down the left-meta key. This makes sense when one remembers that scale mode and zoom mode are quite similar, indeed identical for 1-di-

mensional histograms. Temporary pan mode is enabled by holding down the *Alt* key.

Window Mode

This mode is currently not implemented. Many of the intended functions for Window mode have been moved to the View window.

Title Mode

This mode is not completely implemented. Currently, it only allows the mouse pointer and arrow keys to position the main title. This is intended to help histogram printing. Eventually, the user will be able to highlight the display with textual messages and simple graphics.

The *Edit* Menu Button

The *Edit Data* Option

Currently not implemented. The intention of this window is to allow the experimenter to modify data in looker variables (but *not* in the data acquisition stream).

The *Density Map* Option

This feature, when implemented, will allow a simple re-mapping of displayed density blocks on scatter plots. Currently, the option brings up a density map window which affects nothing. The final implementation of this window will let the experimenter adjust the displayed density size for differing bin content values.

The *Get Region* Option

Choosing *Get Region* starts a window that lists all the region variables in the looker having the same number of subscripts, dimensions, and bins per dimension as the displayed histogram, as shown in Figure 2.34. Only one region can be selected or assigned to a histogram at any one time. The DONE button avoids doing any assignment. Pressing Assign Selected Region causes the named region to be associated with the current histogram and a request made to the looker to return the current region value. Notice that if the region is already associated with another

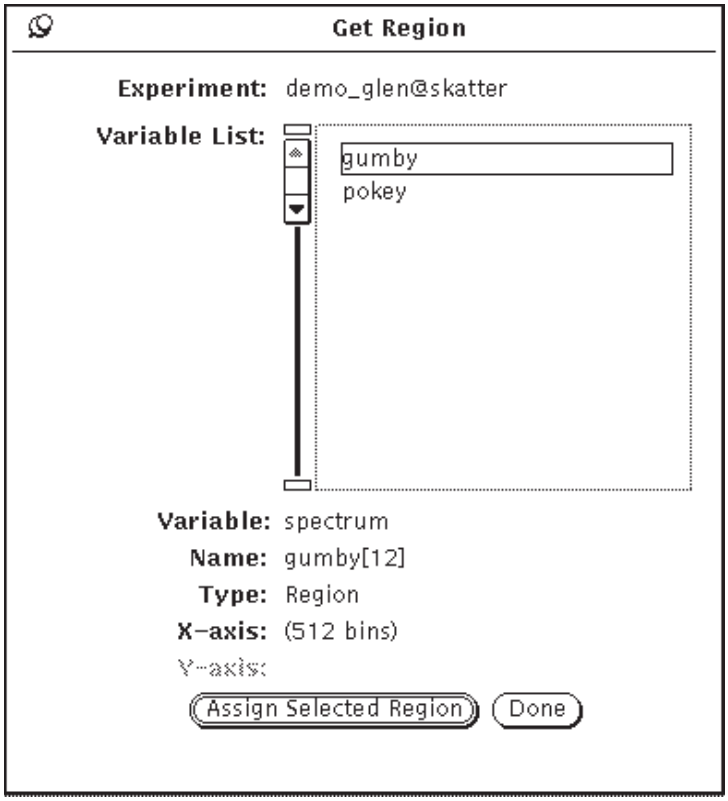


Figure 2.34. The Get Region Window

histogram, the region values are overwritten with the looker values.

The *Send Region* Option

The region values associated with the current histogram are returned to the looker. Any further calculations made by the looker using the named region will use the new value. These values last until the next Send Region or the next build.

The *Save Region* Option

Making the menu selection of Save Region brings up a window requesting a directory name and a file name (see Figure 2.35). Two check boxes allow saving of only the selected area of the region and choosing to append to an existing file or overwrite the existing file. At the current time, the Selected Area Only option has no effect for 1D histograms.

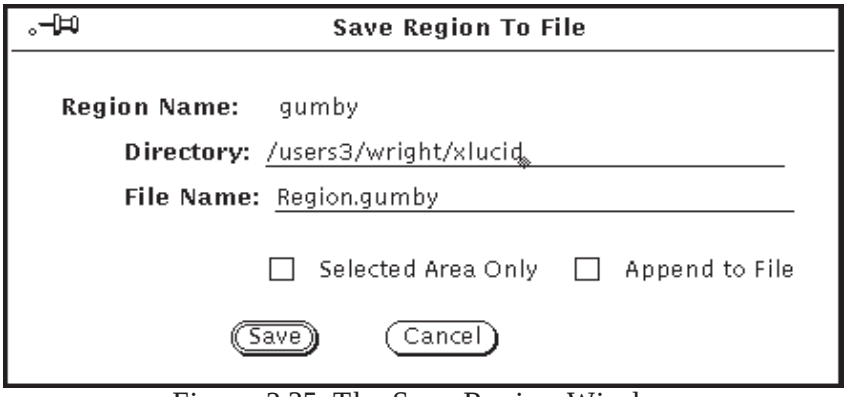


Figure 2.35. The Save Region Window

The *Print* Menu Button

Printing of histograms is intended to be done through the *Print* Menu.

The *Print* Option

Currently not implemented. The intention is to bring up a window which allows adding titles for printing with the histogram, and selection of a printer.

The *Layout* Option

Currently not implemented. This option is a more involved version of the *Print* option. The experimenter will be allowed to select which histograms will be printed on a page, and be allowed to position and size the histograms on the page. All the intended features of the *Print* option will be available with the *Layout* option.

Chapter

3

The Reader

The READER is that part of LUCID which acquires data. It is a program that LUCID writes for you, based on a description of your experiment. Having the programs written automatically from an experiment description relieves you of the task of writing all the low-level acquisition routines yourself.

The first section in this chapter deals with the way the data acquisition hardware must be set up. The remainder of the chapter explains the details of writing an experiment description that LUCID will turn into the data acquisition programs.

Camac Hardware Layout

If you're not concerned with CAMAC at this point, you can skip ahead to the next section entitled Making a Reader Description File on page 3-2.

CAMAC Requirements

LUCID is a multiprocessor data acquisition system. The hardware consists of one or more front-end processors which acquire data from CAMAC, FASTBUS and VME modules. The front-end processors forward the acquired data via an Ethernet link to an acquisition processor. The acquisition processor writes the data to some storage device and makes the data available for further analysis and display.

CAMAC modules are connected to the front-end processors by a parallel branch highway. There can be up to eight parallel branches each with up to seven CAMAC crates. Crate controllers in crates with modules that will be used to generate LAM's must be equipped with a LAM grader card. This card must be wired to connect the crate LAM lines directly to the parallel branch graded LAM lines. For example, the LAM line from station 1 must drive graded LAM line 1, station 2 must drive graded LAM line 2, and so on till station 23 and graded LAM line 23. Graded LAM line 24 is not used.

Using Lams

LAMS (*interrupts*) are often used to determine that an event has taken place. If an event is made up of data from several ADC modules for example, only one of them should send a LAM. Most CAMAC modules can have LAMs disabled, either by a CAMAC command or by setting a switch on the module's side panel. If a module's LAM disable (or enable) is controlled by CAMAC command, then LUCID will enable and disable the correct modules, based on your description file. If a switch needs setting, you must do that before trying to acquire data.

It is important to remember that modules such as waveform digitizers or TDCs may have lengthy conversion times. When using them in an experiment, you should have some other module generate the event's LAM if possible, because the READER can be reading other modules while these "slow" ones are still converting.

Making a Reader Description File

When you ask LUCID to build the programs for your experiment, it needs to know exactly what you want. It gets some of that information from a READER description file, which is a text file that can be made with any text editor. LUCID only needs to see this file once, at the very beginning of your experiment when it acquires the raw data. You won't have to remake the description in the future to look at the data off-line.

The READER description file should be named appropriately for your experiment, but must end with '.r' for Reader. You should not use any special symbols such as asterisks or question marks in the filename, but it may contain digits and upper-case letters such as in He3.r, or carbon14.r.

Remember that LUCID only needs a description of your experiment when raw data are being acquired from CAMAC. If you want to get data from some other place, such as a tape or a disk file, then you can skip ahead to the next chapter; you *don't* need a READER description file to read previously acquired data.

The contents of the description file tell LUCID what data to read, when to read it, and what decisions are to be made before the data is considered acceptable. Here's an example of a simple READER description file:

Example READER Description File

```
#
# acquire data from LeCroy 2249W charge
# integrating ADC.
#
define delta_e "2249W" B(0) C(1) N(5)

trigger background every delta_e lam

event background:
         read and save delta_e
```

The first four lines are comments which LUCID will ignore. The remaining lines can be grouped into four sections:

Four Sections of a Description File

1. Definitions of variables: LUCID treats CAMAC modules as variables, such as `delta_e` in this example.
2. Trigger specifications: In LUCID a trigger is defined as something which causes an event to occur, such as a LAM or some amount of time passing. A trigger can also be generated by the occurrence of other events, which allows you to implement *software events*. This particular example has a LAM from the `delta_e` variable being a trigger of an event named `background`.
3. User Code specifications: LUCID allows the user to specify object files that are to be linked with the generated READER program, or C code that is to be included directly in the front-end reader.
4. Event specifications: An event is a description of what to do when a trigger is received. This might include what CAMAC modules to read, or what calculations or tests to perform. Our example has an event named `background` which will be performed when a `delta_e` LAM occurs.

The four sections of the READER description file must be in order. The definitions must be first, followed by the triggers, then the user code (if any), and then the event section. Each section may contain any number of lines, and the statements within each of these sections are completely *free format*. Notice that each statement begins with a name appropriate to its section: define, trigger, load or program, or event.

LUCID makes no assumptions about columns in the file as FORTRAN does, but experience recommends that each line be kept short so as to be easy to change with a text editor. Related statements should be indented (with a TAB character, for instance) by the same amount. Blank lines are acceptable and encouraged, to allow easy reading by other people. An octothorpe (#) is recognized as the start of a comment. LUCID ignores this symbol and all other characters, up to the end of that line. The only exception to this is when that symbol appears between quotes as part of a string constant.

The DEFINE Section

LUCID allows you to do computations with your data as it arrives, so you can check its validity. Results of these computations can be saved as part of an event, along with the raw data. LUCID allows you to define variables, very much like variables in most programming languages. Furthermore, LUCID treats CAMAC modules as variables, allowing easy access and simple manipulation.

The first section of the READER description file tells LUCID what variables you want to use in your experiment. All variables must be defined so that LUCID can build correct programs, and to make sure there are no typing mistakes in your description. LUCID can also make other checks about the consistency of data from CAMAC modules. For example, it will warn you if you attempt to assign a twenty-four bit CAMAC module variable into a sixteen bit integer variable.

Define CAMAC Variables

CAMAC modules are defined so that variable names can be associated with them in your description file. The following example shows the definition for a Kinetic Systems model 3615 scaler.

```
define doubles "3615" B(0) C(1) N(10) A(0-2)
```

This particular statement tells LUCID that you have such a scaler in station 10 of CAMAC crate 1 on branch 0. It also tells LUCID that you are interested in only the first three inputs, subaddresses numbered 0 through 2. The module can be referred to with the variable name `doubles`, and we will see that it is actually an array containing three values.

CAMAC Database

LUCID knows details of how to access this module because it has access to a CAMAC database (see Appendix A, The CAMAC Module Database). All common modules have entries in this system database, so that each experimenter need only specify a minimum amount of information to have the module accessed properly. In this example, the identifier `"3615"` was recognized because it is listed in the database, along with a description of the module being a 100 MHz scaler. Another name for the same module could be used, such as `"KS3615"` or `"Kinetic Systems 3615"`, both of which appear as alternate names in the system file. This system module database is simply a text file that has been set up (and is maintained) by the system administrator. LUCID users are allowed to read through the file, and even make their own additions. A user is allowed to supply his own database file if he needs to access a module in a non-standard way. LUCID will read the user's file and then the system database, using the first entry with a matching name. When searching for module names in the database, LUCID considers only alphanumeric characters. There is no differentiation between upper and lower case characters. For example, the names `LeCroy 2228`, `LeCroy2228`, and `lecroy2228` all match the same module, `LeCroy 2228 Time to Digital Converter`.

The format of the module database file is described in detail in Appendix A of this manual. The entries are in alphabetical order for easier reading. When creating your own module names and aliases, try and maintain a consistency with the database. E.g.,

```
4508, LeCroy 4508
3615, ks3615, ksc3615, Kinetic Systems 3615
```

The `B and C` keywords

The B and C keywords in our example are optional. LUCID assumes that the module is in branch 0 crate 1 unless told otherwise. Referring back to the original example:

```
define delta_e "2249W" N(1)
```

CAMAC subaddress codes

Although this module is more complicated than the scaler of the previous example, it's definition is smaller because all inputs are to be read when it is accessed. The sub-address code is optional and LUCID assumes all sub-address codes are to be used if you don't list any in your definition.

Module Timeouts

For some experiments, it may be efficient for certain modules not to interrupt the computer. Rather than waiting for a LAM, LUCID can continuously poll a module to determine when data is ready. This is important when you have a module with a slow conversion time that is to be read out after a LAM is received from some other module. The details of how you wait for the data will be discussed under the heading Polling Camac Modules on page 3-20, but at this point the concern is timing.

It usually isn't a good idea to just wait forever for a module to become ready, so LUCID allows you to specify a timeout period. This tells the READER that if it must wait for the module, then it must not take longer than a certain amount of time. If it does take longer, the data is considered bad. A CAMAC module's timeout value must be given with the definition, as shown in the example below:

```
define energy "LeCroy 2259" N(10)
                timeout 10 microseconds
```

Times up to several minutes may be specified, although only a few microseconds should ever be necessary for most applications. You may actually use the following keywords to specify times: `microseconds, milliseconds, seconds, minutes`. A default timeout is included in the CAMAC module database for most "slow" modules.

How to DEFINE Variables

LUCID variables are defined in a way similar to FORTRAN, as shown in this example:

```
define energy real*4
```

This statement defines a single precision variable named *"energy"*. Several variables of the same type may be defined in the same statement:

```
define e1,e2,e3 integer*2
```

Variable Data Types

The following table shows the allowable variable types.

Definition	Range of Values
`real*4`	1.2×10^{-38} to 3.4×10^{38} appr. 7 decimal digits of precision
`integer*2`	-32768 to 32767
`integer*4`	-2147483648 to 2147483647
`character*`*N*	Only used to store text. *N* may range from 1 to 255.
`boolean`	0 or 1

Defining Arrays

One dimensional arrays of any type can be used, and are defined by adding a number in brackets to the name. For example:

```
define calib[100] integer*2
```

This allocates an array of 100 integers, each containing two bytes. When the array is actually used, a subscript is often used to identify a single element in the array. The subscript always starts at zero, so that elements in this array can be accessed with subscripts from 0 to 99. This is important for CAMAC operations, since sub-address 0 from a module is commonly the first element in an array. Arrays are further discussed under Using Variables on page 3-12.

Variable Names

When thinking of names for your variables, try descriptive ones that will be obvious to your colleagues. Remember that these names will be associated with your data when analyzed off-line, possibly months after the experiment is finished. You must also avoid using keywords for your variable names, such as "`read`" or "`define`". This can *really* confuse LUCID, so make sure you read Words Reserved for the Reader Description on page 3-31. A variable name should not be longer than 32 letters, and the first character *cannot* be a digit. An underscore can be used as part of a variable name, and lowercase letters are considered to be different than uppercase ones.

Memory restrictions

There is no preset limit on the size of arrays, or on the number of variables that you use, other than the amount of memory available on the associated computer. Large amounts of space could affect data throughput, since buffers share the same memory as your variables. In practice however, the definition section con-

sists of as many definition statements as you care to include. Keep in mind that although an array may be very large, you aren't allowed to save more than approximately 32,000 bytes in any one event. A point to remember is that all LUCID variables are guaranteed to be zero when the program starts. This doesn't mean they will be reset to zero before every run, but they will be zero before the first start command is issued.

A few sample definitions are listed here:

Definition Examples

```
define energy1 integer*2
define energy2 real*4
define pi1,pi2,pi3,pion[40] integer*2
define time1 "LeCroy 2229" N(10) A(6-11)
define counts "lrs4448" N(11)
```

The TRIGGER Section

Normally, we think of an event as some happening or occurrence which causes a result. LUCID defines an event as a collection of data items. When data is recorded on tape for example, it is basically a list of thousands of events, one after another. A trigger is defined as some occurrence which causes the event to be processed. If the source of the data is a CAMAC module, then a trigger is often a CAMAC LAM.

Triggers must be described in the second section of the READER description file. The section contains statements which associate some occurrence with the action of acquiring the data. There are several types of triggers that LUCID can respond to:

Trigger Types

- CAMAC LAMs (interrupts).
- CAMAC modules which set a "ready" status bit. This implies polling for a "data ready" state.
- Some amount of time passing.
- A certain number of valid events occurring.
- Executing a *trigger* statement in an event specification.
- A change in LUCID's state, such as the start or end of a run.
- A request from the experimenter.
- Some external source making data available.

The syntax of the trigger definition statement is shown in the following example:

```
trigger electron_arm every
          delta_e lam or every 3 seconds
```

The Trigger Statement

The trigger statement always starts with the keyword `trigger`. The next portion tells the READER to perform some action called an "`electron_arm event`" when the triggers arrive. Details of that `electron_arm` event would be described in the event section of the file. This particular example associates two different triggers with the event, specifically a LAM from the CAMAC module named `delta_e`, and a time interval of 3 seconds. When either of these things occur, the trigger is performed. LUCID allows several triggers to cause an event. If both things happen at the same time, LUCID will see both of them; nothing will be lost, because triggers remain pending until serviced.

LUCID State Triggers

In many experiments, the user will want to perform some kind of "housekeeping" duties at the end of a run, such as reading scalers or disabling high voltage equipment. LUCID generates a trigger automatically when you give a command to start or stop a run. For instance, it is acceptable to say:

```
trigger scaler every endrun
```

There are four such "internal" trigger names, called `beginrun`, `endrun`, `suspendrun` and `resumerun`. As shown in the example, they are associated with events in the same way as a CAMAC LAM or a time unit would be. These triggers will be issued when a `start`, `stop`, `suspend`, or `resume` command is given to the LUCID program. When triggering from a `start` or `resume` command, LUCID will produce the trigger before doing anything else. Correspondingly, a `stop` or `suspend` command will cause LUCID to finish whatever it's doing, and issue those triggers last.

Software Triggers

Sometimes housekeeping duties are required to be performed before a run is finished, possibly after a certain amount of data has been collected. LUCID allows you to specify that a certain number of processed events will act as a trigger. For instance, you could use the following statement:

```
trigger scaler every
          1000 delta_e
```

This allows you to read the scalers after processing one thousand real data events, ideally to guard against counters overflowing. Note that the trigger is not really put into effect until after the one-thousandth "delta_e" event has been successfully processed.

An event can trigger another event. See Triggering Another Event on page 3-26 for a description of how this is done.

Keyboard Triggers

LUCID allows you to attach a command on your terminal to a trigger. This is often used for testing various parts of your equipment before the experiment actually starts. For example:

```
trigger reset every "reset" command
```

This attaches a trigger to an event called *"reset"*, which will be performed whenever you type the command reset to the LUCID program. Any sequence of keystrokes can be used, including control characters. Notice that the command name must be enclosed in quotes.

If the LUCID program is being used with a different interface, such as a mouse on a workstation, then it will install your command on a software button. In that case, you could simply point the mouse arrow at the appropriate title and click the mouse button. See Control Buttons on page 2-19 for more information.

To enter a control character as part of the triggering command, type a *circumflex* in front of the character's symbol. For example,

```
trigger reset every "^R"
            command or every highvoltage lam
```

This will cause the `reset` event to occur whenever you type a *control-r* to the LUCID[1] program, or if the "highvoltage" module gives a lam. If you really want to have the circumflex (^) as part of a command, you should enter two of them in the command, one after another (^^).

Module Ready Triggers

The ready status of a module can initiate an event through the `when` *`module`* `ready` trigger. LUCID then checks for a Q response from the module continuously, and the event executes when a Q response occurs.

Continuous Triggers (*Polling)*

Finally, there is the simplest of all triggers. If you enter the keyword `constantly` as part of the statement, LUCID will trigger the associated event as often as it can, but allow other triggers to take precedence when they arrive. For example:

```
trigger tdc_readout constantly
```

This statement will cause the "tdc_readout" event to be performed as fast as the computer will allow. All other triggers listed in the file always take priority over this type of trigger, except for other ones being performed constantly.

1. The XLUCID program does not recognize control characters: it only allows commands to be run by button selection. For this reason, it makes more sense to have commands with meaningful names rather than single key presses.

Trigger Examples

Some examples of trigger statements are listed here:

```
trigger good_hit every tdc lam
trigger counting every 2 seconds
trigger delta every "test" command
           or every coincidence lam or every 10
           overflow event or every 60 minutes
trigger calibrate every minutes
```

The USER CODE Section

It is common for a non-standard module in an experiment to require some special handling which the READER language can not easily provide. The user code section has two statement types, one of which allows direct inclusion of C code, and the second which allows specifying object modules or libraries to be used.

C code is included directly with the program keyword; e.g.

```
program "/home/wright/Lucid/bitswap.c"
```

This statement causes the named file to be included in the generated frontend program. The C code can declare global variables and functions, and have access to the variables and definitions created from the DEFINE section. Variables and functions defined here can only be accessed by user code entries in the EVENT section (see User-written Code on page 3-29)

The second type of inclusion is with the load keyword; e.g.

```
load "/home/wright/Lucid/readercommon.o"
load "/home/wright/Lucid/readerlib.a"
```

Files named with the load statement may have been generated from FORTRAN, ASSEMBLER, or C, or may be a library or object files so generated. The only restriction on the compiler used to create the object file is the functions must be C callable, or the global variables must be C accessible. Object files or libraries built in this way do not have direct access to the variables or definitions created from the DEFINE section[1]; instead, calling routines must pass the variables (or their addresses) to the routines in the object files.

1. The source files for loaded object files and libraries could include *frontend*.buildname.*h* file in the experiment directory to gain access to the DEFINES and structure definitions. The user must find the path to this file (see The LUCID Experiment Directory on page 6-4). The user's source files are not automatically recompiled when a change in the reader file causes a change in the *.h* file.

A word of caution is necessary at this point. LUCID will record the reader description file to the output device, but does not attempt to record any files referenced by a program or load statement. This means it is no longer possible to re-create exactly the environment an experiment had when it was run. Code included in the reader in this manner should be printed and included in the experiment log book, and all changes to the code should likewise be documented.

The EVENT Section

The event section of the READER description file is usually the largest, because it tells LUCID what to do after receiving a trigger.

The section consists of one or more event specifications, one after another in the file. Each specification may contain many statements, but the first statement is an event name, similar to a subroutine name in a programming language. For example, the statement

```
event electron:
```

associates all subsequent statements (up to the next event specification) with the event name `electron`. Notice that the first word is `event`, being similar to the first word of the `define` and `trigger` sections.

LUCID makes sure that every event name that you specify is triggered by something. If LUCID finds an event without a trigger, it will warn you that the event will never be performed.

The rest of the statements in the event specification are similar to the statements in most programming languages, allowing you to access any variables (including CAMAC variables) that were defined in the first section of your file, and perform tests or some arithmetic on the associated data.

Using Variables

The acceptable range of values for numeric arguments was listed under How to DEFINE Variables on page 3-6. A variable defined as `character` can contain any valid ASCII character between 40 octal and 176 octal (i.e. between the *space* character and the *tilde* (~) character), the *return* and *newline* characters, and the *tab* character. The null byte is reserved for internal use, and marks the premature end of a string of characters in a larger variable. `Boolean` variables are numeric variables which may only contain a one or zero. A value of one denotes a true state, and zero a false one.

Constants

LUCID understands integer constants in binary, octal or hexadecimal (base two, eight, and sixteen, respectively). To specify a constant in base eight for example, you should make the first digit a "0", as in:

```
0351
```

Similarly,

```
0b11101001 and 0xe9
```

refer to the same number (233) in base two and hexadecimal. Base 16 numbers can use either upper or lower case letters "A" through "F".

Statements in an event specification can access variables, including CAMAC modules and arrays. `Character` variables are used infrequently, usually to record an operator's name or the name of an experiment within the data stream. Recall that all variables must be defined in the define section of the description file.

When elements of an array are accessed, an index must be given between brackets following the array name, such as "`phi[3]`". Remember that all arrays are indexed starting at zero, so that this particular example is referring to the fourth element of the array called "phi". An index of a subscript may be any arbitrary expression that evaluates to an integer.

Arrays may be named without subscripts, allowing operations to be carried out on *every* element. For instance, if we assume that the array "theta" contains fifty integers, the statement

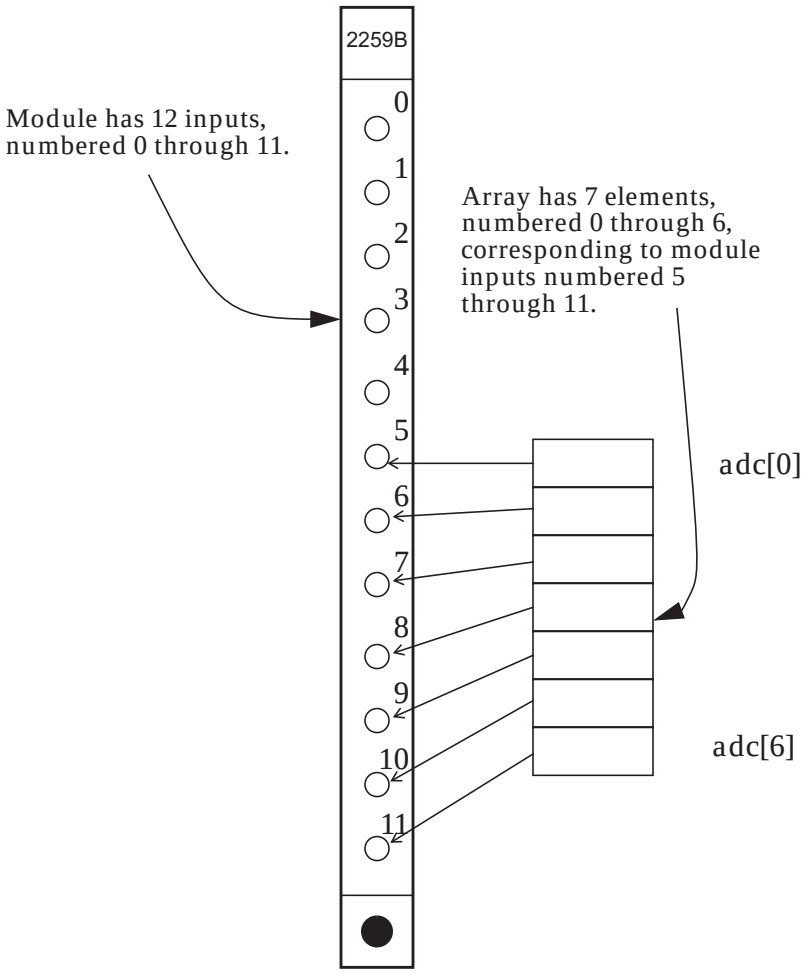
```
theta = 100
```

will set every element of the array "theta" to be one hundred.

Because most CAMAC modules have several inputs, CAMAC variables represent arrays having subscripts which correspond to their sub-address (*A*) codes. All CAMAC variables are identical to arrays in that they are indexed starting at zero. For example, let's assume that the definition of a CAMAC variable "adc" indicates that only sub-address codes five through eleven of the CAMAC module are to be accessed.

```
define adc "lrs2259B" N(4) A(5-11)
```

This means that "adc" can be used as an array of size *seven*, with the sixth module input corresponding to "adc[0]".



Similarly, the twelfth input (numbered 11) corresponds to the last array element, "adc[6]". If a particular CAMAC module has only one input, or responds to only one sub-address code, it is still treated as an array. The user need not be concerned with this however, since using "array" is identical to using "array[0]" in such a case.

Reading Data

Most programming languages allow input of data to variables in a program. The `read` statement can be used to read a value into a LUCID variable, just as the read statement in FORTRAN does. The statement:

```
read angle
```

will cause the READER to print a message on the user's terminal, in the form

```
Enter angle:
```

The READER will then stop and wait for the user to enter a value that is consistent with the definition of `angle`. We are assuming that "angle" is defined as a true variable, and not a CAMAC variable or an array. When prompting for input, the READER simply prints the word `Enter`, followed by the name of the variable. If you would like to give the operator more information about what's expected for input, you can specify a message in the read statement, after the variable name. For example, the statement:

```
read angle "the spectrometer angle, 20-60 degrees"
```

would display

```
Enter angle (the spectrometer angle, 20-60 degrees):
```

to the operator.

Reading Array Data

If the variable is actually an array element, then the default prompt would indicate that fact to the user. For instance, the statement

```
read adjust[0]
```

would cause the READER to prompt with

```
Enter adjust[0]:
```

If the name of an array is given without a subscript, the READER will prompt for each element's value in turn, until each one has been read.

```
Enter adjust[0]:
Enter adjust[1]:
Enter adjust[2]:
     •
     •
     •
```

Although LUCID allows you to enter certain values interactively as your experiment proceeds, this isn't always convenient if you're not sure when they will happen, or if an inexperienced colleague is filling in for you. LUCID offers two flexible options for reading values into variables.

Reading from a File

The first option allows you to read values from a file. The statement:

```
read adjust from "/home/wright/adjustments"
```

will cause the READER to get the data from the named file, rather than from the user's terminal. This allows large arrays to be read directly from a file while an experiment is running. The named file must contain a filesystem *pathname*, and refers to a file on the computer on which the experiment is running (as specified as part of the experiment name, not necessarily the computer the LUCID command was run on). Naturally, you must have permission to read that particular file.

File Format

LUCID will use the first value in the file for the variable, so the rest of the file could contain anything. If you're reading an entire array, each value should be separated by blank spaces, tab characters, or appear on different lines. As with single variable files, LUCID will ignore the rest of the file after it has retrieved enough information to fill the array.

Notice that this option contains the keyword `from`, which tells LUCID that the name in quotes refers to a *file*, rather than a *prompt* for interactive reading. *Only one array or variable can be read from a file.*

Reading from a program's output

The second option allows you to use the output from some program as the value to use for the variable. For instance, the statement:

```
read adjust from program
          "/home/wright/bin/getadjustments"
```

when executed, tells LUCID to start the *"/home/wright/bin/getadjustments"* program running on the computer which is running the READER, LOOKER and WRITER. LUCID takes the first piece of information output from the program, and places it in the named variable. If the variable is an array name, then LUCID will try and get the correct number of values from the output, and assign each one to successive array elements. The program will output the values separated from one another by blank spaces or tabs, or place each value on a separate line.

Program Options

As with filenames, the program name must refer to a complete filesystem pathname, and you must have permission to run that program. Within quotes, you specify the program name just as you would if you were sitting at a terminal. For example, a colleague might have a program which can read settings of various mechanical components of a detector. At your terminal, you might normally type

```
/home/wilson/getvalue B302
```

to find the setting of some component named B302. From within your LUCID program, you can type

```
read energy from
program "/home/wilson/getvalue B302"
```

The program can be any executable program, including a command script. Furthermore, you can specify input redirection and use pipes between the quotes to start a pipeline of several programs, just as you would from a terminal. The programs are allowed to access other computers, or do anything that an ordinary program can do, but it must produce output in the way mentioned above. Also, when LUCID runs the program(s), they aren't allowed to have any input from the user. Parameters can only be passed to program(s) on the command line which starts them.

When to use *read*

So far, the read statement sounds like a simple but reasonable statement for a programming language to use. However, when we consider that LUCID is meant to acquire experimental data at high rates, it seems out of place. Actually, the intended purpose is to allow certain parameters to be input at the beginning of a run, for example. This might include data which was not easily accessible on line, such as a spectrometer angle, or the mixture of gas in a wire chamber. Remember that the READER simply stops and waits for input when it encounters this statement, essentially stopping the experiment. If a user is not running the LUCID command when input is needed, the READER will simply wait until one exists before prompting for input.

Reading CAMAC Variables

Now, we'll consider something a bit more useful. If the variable used in the read statement is a CAMAC variable, then the corresponding CAMAC module will be read. If you attempt to supply a prompt, such as the one in the previous examples, LUCID will give you an error when you build the software. Recall that CAMAC variables are considered to be arrays, because they may correspond to modules with several inputs. Let's refer back to our example of an adc module which was defined as using inputs five through eleven; the following statement could be used to acquire that data:

```
read adc
```

As mentioned in the last section, referencing an array without using a subscript causes the entire array to be accessed. Theoretically, one could replace this example with seven read statements, each using "adc" with a different subscript, but achieving the same goal. Such a technique would work, but would be much slower than reading the entire array in one statement.

The READER determines how to read the modules by checking the system CAMAC database, as described in Appendix A.

Saving Data

Probably the second most important feature of LUCID after reading the data is saving it someplace. We've mentioned that the WRITER will actually record the data, but the READER must determine which variables should be saved. The `save` statement tells the READER to save a variable for the rest of the system to use. Remember that *reading* a variable does not mean that it will be *saved*.

The `save` statement is usually combined with the read statement:

```
read and save adc
```

***NOTE*:** This method is the most common way of acquiring data, but it places a restriction on the rest of your event. Specifically, if LUCID finds a statement of this form, where the variable is being read and saved on the same line, it will move the data directly from CAMAC to the data stream. This is the fastest way of acquiring data, but it means that the data won't be available to the rest of your `event` statements, or even within this event statement. It is quite acceptable to use two separate statements:

```
read adc
save adc
```

This is less efficient than the previous example, but it tells LUCID to save the data for future computations by the READER.

As one might expect, using the save statement on an array with no subscript will save the entire array. You should also be aware of a restriction placed on the save statement when the if statement is used. Specifically, you aren't allowed to save data within an if statement.

`SAVE` Restriction

Format of Event Data

The structure of an event "record" is determined by the order in which variables are `saved` in the event specification. For those users wanting to write their own software, the build command puts a definition of the event record into a LUCID file. A C "include" file containing the structures describing the event data is also produced. There are also subroutines available to read the event record, so it is unnecessary to manually write code which knows the actual event record layout. These features are further described in Chapter 7.

A special form of the `save` command is:

```
save data
```

This statement causes any data buffered by the front-end computer to be transmitted as soon as possible. For an experiment with a low acquisition rate, this statement could be placed in an event triggered every few seconds, or possibly in an event triggered by a user command.

Alternatively, the special save command can be entered as

save data and wait

As with the `save data` command, this immediately causes the data buffer to be transmitted to the user's LUCID session. However, acquisition will not continue until the LUCID session has processed this data! The effect is that none of the different software sections of the experiment get too far ahead of any other section, and responsiveness to user commands (including "stop" and "pause") is maintained. Because this statement can create large dead times, depending on the amount of processing occuring in the LUCID LOOKER, it should not be used for production runs.

Other CAMAC Operations

To process some events, other CAMAC operations must be performed after the data is read. This could include clearing an input, resetting a LAM, or adjusting other module parameters.

Clearing CAMAC Modules

It is often important to clear a CAMAC module before proceeding with an event. The following statement can be used:

```
clear adc
```

In most cases, this issues a CAMAC F(9) command, which usually clears the lam status bit as well as clearing the module. The system CAMAC database is used to determine the correct function code.

Read and Clear Operation

Many CAMAC applications require that a module be cleared after it is read. Most modules have an single read-and-clear operation, in addition to a simple read function. By using the statement

```
read and clear adc
```

the READER will perform the correct CAMAC function (usually an F(2) command). The READER will use a different function code simply because the `read` keyword appears. The `clear` statement has a different meaning when used independently. Using `clear` on a normal variable (i.e. one that isn't a CAMAC variable) will simply set the variable (or entire array) to zero if defined numeric, null if defined as a `character`, or false if defined as a `boolean`.

The rule to remember is that when read and clear commands appear in the same statement, and if the module supports the read-and-clear operation, the module is issued a single CAMAC command containing (usually) F(2). If the commands are in separate statements, or the module has no single read-and-clear operation, then two separate CAMAC commands will be issued.

Read, Save and Clear Operation

The clear command can be combined with the earlier read and save command, giving

```
read, save and clear adc
```

This is particularly efficient method for many modules.

Polling Camac Modules

The term "polling a module" refers to the action of repeatedly accessing a module until it says that data is ready.It is essential when reading from modules that have a lengthy conversion time, but do not issue a lam when finished. Note that a lam source signifies a unique trigger, and LUCID can't perform a portion of an event when a trigger arrives. Therefore, if several modules are to be read, only one of them can issue a lam. As mentioned on page 3-6, modules with lengthy conversion times should not be the ones to issue the LAM, and they should be read *later* in the event, giving them time to finish. In such cases, polling the module is required.

The `wait` command

The keyword `wait` can be used to poll a module by giving it the name of an associated CAMAC variable. For example:

```
wait adc
```

will cause the reader to continuously send a CAMAC command which tests for data being ready. Most modules respond to the CAMAC F(8) command by issuing an appropriate Q response. A Q response of true (a one), indicates data is ready. The CAMAC function code that will be used is looked up in the system CAMAC database for the module, and LUCID will warn you if a particular module has no polling mechanism.

Because it is possible the *wait* command will timeout, the *qresponse* variable should be tested after a wait statement.

```
wait adc
if (qresponse == 0) then
        # Adc not ready due to timeout.
        # Reject this event.
        reject
endif
```

As in previous examples, the `wait` keyword can be used with certain other keywords. The most common example is:

```
wait, read, clear and save adc
```

Each such keyword can be separated from the next by a comma or the `and` keyword for readability. The `wait` command has no effect on non-CAMAC variables, and LUCID will warn about such usage.

CAMAC Q Response

The `wait` command can be very useful for certain modules, but could give incorrect data in other cases; the timeout period that was specified for a module may have expired, and the data that was acquired could be meaningless[1]. To let you check for this condition, LUCID provides a `boolean` variable, called `qresponse`. It does not have to be defined like other variables, since LUCID defines it for you. It can be used wherever a normal `boolean` variable could appear. Is best suited to the `if` command, described under Making Decisions about the Data on page 3-25.

The `qresponse` variable will contain a one (true) value if the last CAMAC operation returned a Q response, and it will be zero when there was no Q response. You should not assign some value to the `qresponse` variable as you would other variables as it is considered *"read only"*.

Compressing Array Data

In many experiments, a sparse matrix of data may be acquired. For instance, a wire chamber could have hundreds of TDC channels, most of which will contain irrelevant data when a trigger arrives. When a computer goes to read and save the data, most of it is unimportant.

LUCID allows you to `compress` an array by specifying what you consider to be unimportant. The compression actually refers to discarding all unimportant data, and adding an *index* for each good value. For instance, if an array of one hundred TDC values contained "unimportant" data except for the last two elements, the compressed array would contain four elements: the number ninety-eight and the second last TDC value, followed by the number ninety-nine and the last TDC value.

1. The definition of a timeout delay is discussed on page 3-6.

Any CAMAC variable can be compressed. In some cases, it could be quite inefficient to do so, as the saved data contains both the saved values and the saved indices. For instance, if less than half of the inputs to a module have "unimportant" values, compressing the data would result in an array larger than the original!

The `compress` Statement

To use this feature, simply use the `compress` keyword in the same way as `read` or `save`. For example, the following statement is valid, assuming that "timing" is defined as a CAMAC variable:

```
read, compress and save timing
```

In this default case, LUCID assumes that zero values are unimportant, and all non-zero values are saved. The `compress` feature only makes sense when used with the `read and save` command. The `compress` keyword will be ignored if `read and save` is not specified on the same line.

More General Software Discrimination

It is often important to have some range of unimportant values for compression, such as the case of a TDC module operating in common-start mode. In this instance, all full-scale values would be unimportant. At this point, we can think of the `compress` statement as a software discriminator. The following example shows how this can be done:

```
read, compress and save timing (discard 200-255)
```

This is similar to the previous example, except any values in the range of two hundred to two fifty-five inclusive are removed when the compression is done. The `discard` keyword can accept a *list* of values as well. For example:

```
read, compress and save timing
      (discard 0-100,180,200-255)
```

This statement might be useful when correcting for hardware problems, such as a bad wire in a wire chamber (i.e. when you want to ignore a particular channel).

Q-Stop Compression

A second type of compression is performed for Q-Stop modules. Again, the READER is saving a variable number of entries for the module. However, only the values are saved. The array's indexing is assumed to go from 0 to the number of values saved.

Several Variables at Once

As in previous examples, the `compress` statement can be used with `read`, `clear`, `wait`, and so forth. Several variables may be specified for any of these operations.

```
read and save adc, delta_e and e_prime
```

Each of the three variables will be read and then saved, as if three individual `read and save` statements were used. If the `discard` feature of data compression is used, it will apply to all data saved in that statement.

White space, a comma, or the `and` keyword may be used to separate variables which appear in the same statement, just as operation keywords are separated. For example:

```
wait, read, clear and save coincidence, scalers,
adcs and tdcs
```

Calculations and Variables

LUCID allows arbitrary calculations to be performed on variables. The most common form of this is:

```
variable = expression
```

The variable can also be an array name without a subscript, as discussed in previous sections. The `expression` can be a regular arithmetic expression using the following operators:

LUCID Operators

Symbol	Function
**	exponentiation
*	multiplication
/	division
+	addition
-	subtraction
<<	bitwise left shift
>>	bitwise right shift
%	integer modulus (remainder from division)
&	bitwise “and”
|	bitwise “or”
^	bitwise exclusive “or”
~	one’s complement
-	negation
!	boolean “not”

The first part of the table contains binary operators requiring two operands, and the second part lists unary operators. Parentheses can be used for grouping. The bit operators can be used on any integer variables.

Using Arrays Without Subscripts

An expression cannot contain a subscript-less array reference unless it is used in an assignment to an array. For instance, it is legal to use an assignment like:

```
corrections = corrections * 4
```

where 'corrections' is previously defined as some kind of numeric array. However, it would be incorrect to use

```
path[7] = corrections * 4
```

where 'corrections' is the same numeric array. In this example, note that 'path' is an array containing at least eight elements (indexed 0 through 7), so the entire 'corrections' array can't fit into a single element of 'path'. A further restriction is that the array on the left side of the assignment must be the same size as an array on the right side. If there is no array mentioned on the right side of an assignment, then each element of the left-hand-side array will be set to the same value.

CAMAC variables are not accessible after a read and save operation. If such data are to be used in calculations, the read and save operations must appear in different statements.

Printing Messages During Event Processing

It is necessary to display messages, usually during an initial set-up period, and occasionally when an unusual type of event is encountered. Because printing can slow event handling significantly, message printing should be limited to special circumstances.

LUCID allows you to print messages with the print statement. Variables and character strings may be printed in any combination, and are separated by commas, as shown in this example:

```
print "Signal from scintillator", scin,
          ",value was", tdc[scin]
```

In this example, the variable "scin" contains some input register value, and "tdc" is an array of time values from a CAMAC module. The resulting message might say:

```
Signal from scintillator 3, value was 511
```

The message is sent to every screen which is currently using the LUCID command, and will be displayed in their message areas. If there are no LUCID programs running, the messages are placed in a log file. As mentioned, the print statement is not meant to be used when common events are encountered, because it can re-

ally slow down the data acquisition. Printing can take up a lot of computer time, and even worse, a lot of disk space if log files are being used. On the other hand, it can be very handy to know when an error occurs during the experiment, or when some interesting but infrequent event is found.

A variation of the print statement exists to help specifically during setup or test periods. The `display` statement has exactly the same form as the print statement, but it will print the *name* of each variable given, followed by a colon (:), a space and the variable's value. Character strings are displayed in the same way as the `print` statement prints them.

Making Decisions about the Data

It is important to perform tests on the data when processing an event. This could include checking for a certain bit pattern in a coincidence register, or checking the result of a calculation. This is done with an *if-then-else* statement, much as is used in other languages.

The following example could be used to check the value of an input or coincidence register. The variable coincidence has been read from a CAMAC module which has three input signals each which is associated with a different event type. There will be one bit in the register corresponding to each event, and the electronics doesn't allow more than one bit to be set at one time:

```
if ( coincidence == 0b001) then
      trigger calibrate
elif ( coincidence == 0b010) then
      trigger hadron
elif ( coincidence == 0b100) then
      trigger electron
else
      bad_coincidence = bad_coincidence + 1
endif
```

Other statements are introduced in this example. The *trigger* statement is a software trigger of the named event, allowing different event generation from one initial trigger type.

Triggering Another Event

Some event triggers may not provide sufficient information for the READER to trigger the desired event type directly. In this circumstance, a *conditional trigger* can be made which causes another event to be triggered at the end of the current event, *even if the current event is rejected*! It is even possible to trigger multiple events to occur after the current event; the events occur in the order they are listed in the current event. A conditional trigger appears as:

```
if ( coincidence == 1) then
        trigger eventname
endif
```

Rejecting an Event

After making certain tests, you might determine that some data is in error, or unsuitable for what you wanted to do. In such a case, you can use the `reject` statement to throw away the event. When the READER sees the `reject` statement, it simply stops what it was doing (it backtracks over anything that you might have saved to that point), and goes back to waiting for a trigger.

There is no indication in the event stream that a rejected event was encountered.

The `reject` statement is very useful in conjunction with the `qresponse` variable within an *if-then-else* statement. You can `then reject` an event if a module doesn't come ready in time.

After a `reject`, any conditionally-triggered events are triggered. Note, however, that events that are to be software triggered `(i.e. every N other_event, see page 3-9)` will never occur after a reject.

Writing to Camac Modules

LUCID allows writing to CAMAC modules. This might not seem a common thing for a data acquisition system to do, but it proves very useful in a variety of situations. For example, you could load an entire programmable logic unit when the experiment is started, or when a power supply overcurrent trigger arrives, you could write a zero value into its controller.

You can use the `write` statement, as shown in the following example.

```
write 100 into dac
```

This simply puts the named value into the module named dac, using the default subaddress and function codes for that CAMAC module, as defined in the CAMAC database. To write larger amounts of data, you could use an array name. For instance, to load an entire file into a programmable logic unit, you could use the following event statements:

```
read pludata from "/home/norum/plu.datafile"
write pludata into plu
```

This is inefficient use of computer memory when the pludata array is only used once in the entire program, taking valuable space away from data areas. An alternative to this solution is found with the load statement.

Loading Data into a CAMAC Module

The write statement is useful for loading values into CAMAC modules, but is inconvenient for loading large amounts of data. To handle such cases, LUCID provides a load statement, which transfers data from a file *directly* into a module. For example, the following statement would perform exactly the same function as the last example in the previous section:

```
load plu from "/home/norum/plu.datafile"
```

This is an efficient way of getting values into a module, and it doesn't waste any valuable data space.

In many cases, an experimenter might load a waveform generator, or a programmable logic unit with numbers produced by a program. LUCID allows you to load the output directly into the module, without making an intermediate file. For example:

```
load wavegen from
          program "/home/pywell/bin/makewave"
```

This is identical to the read from program statement previously discussed. Keep in mind that the exact contents of the file are loaded into the module; no conversions are done, so that modules requiring binary information must be loaded from binary files. It's also a good idea to check the CAMAC module database for the number of bits that the module needs for a valid transfer, either sixteen or thirty-two. Your data file (or program output) must contain the exact image of what is to be loaded into the module.

Specialized Camac Access

LUCID allows you to specify arbitrary CAMAC commands. For example:

```
define controls "sal controller" B(0) C(10) N(1)
          .
          .
          .
camac controls A(0-11) F(16) data cdat
```

Here, it is assumed that `cdat` is an integer variable. In this example, the CAMAC operation is a write command, and `cdat` contains the data to write out. Similarly in a read command, the READER will put the data that was read into the named variable. LUCID knows whether you are reading or writing data by checking the value of the function code. If the code specifies a *control* function, then you can't have the `data` keyword or a variable name associated with the statement. Although the data variable is optional, its presence depends upon the type of function code used. If writing, the expression following the `data` keyword may also be a constant value.

The subaddress may contain a range of a-codes, in which case the READER will repeat the function code with each one. Both the function codes and the subaddress code(s) must be integers, as opposed to variables.

You can also specify a branch, crate and station as in

```
camac B(0) C(10) N(1) A(0-11) F(16) data cdat
```

A missing branch specification is taken to be 0, and a missing crate specification is taken to be 1. Use of this form of the `camac` command is discouraged as it is harder to track down and change the statements when a module is moved.

Camac Crate Operations

Certain CAMAC modules allow you to stop taking data momentarily, by using a signal on the CAMAC dataway. The inhibit (I) signal is passed to every station in a CAMAC crate, and is normally in an off state. Some modules will suspend their actions when this signal becomes true, and restart when the signal becomes false again.

LUCID allows you to set or clear this inhibit line from your READER description file. Before using it, you should carefully check the hardware manuals for every module within the crate. Some modules react differently to the inhibit signal, which is sent to every station.

To set the inhibit line, use the following command:

```
dataway inhibit
```

If the branch or crate are not specified, the branch is assumed to be 0, and the crate is assumed to be 1. A more general form of the command allows the inhibit signal to be set in different crates:

```
dataway C(4) inhibit
```

Specifying crate 0 gets the entire branch at once.

Similarly, to clear the inhibit signal and resume module operation, use:

```
dataway uninhibit
```

In addition to inhibit and uninhibit, the `dataway clear` and `dataway initialize` operations are also permitted.

User-written Code

The situation can arise where the READER description language is inadequate for a particular job. Rather than giving the READER a full-blown general purpose programming language, it makes more sense to allow access to existing general purpose languages. The `program` statement does just that by allowing user C code to be included at the current point in the event.

```
define data
    average
    compress inputs
    program "/home/wright/Lucid/reader_calc.c"
```

The above example tells the reader that the C code in the file "/home/wright/Lucid/reader_calc.c" will place the values of `average` and `inputs` in the event record, the latter as a compressed array. The variables `average` and `inputs` must have been defined earlier in the READER.

There are cases where it is more convenient to include the C code directly in the reader rather than in a separate file. This is done simply by the sequence

```
program { /* divide average by 2 */
        dp->Vaverage /= 2;
}
```

To make coding easier, the following definitions and declarations are established by the READER:

```
variable names are Vname (e.g., Vaverage above)
variables in the event record are dp->Vname (e.g.,
dp->Vaverage)
each CAMAC module defined has the C defines
Cmodule_branch, Cmodule_crate, Cmodule_station,
Cmodule_lowaddress, Cmodule_highaddress (e.g.,
Cdoubles_branch)
```

Saving compressed variables takes a special effort. The local variable cmpvar_offset contains the offset in bytes from the start of the event record (`dp`) where compressed data can be recorded. At completion of the user code, `cmpvar_offset` must point to the first byte after the end of the saved compressed data that is on a LUCIDALIGN_RECORD boundary. The event-record variable (`dp->Vaverage` in the above example) is not an array, but rather a structure of type `Lucidcompressoffset`.The `lco_begin` field must point to the original `cmpvar_offset` value, and the `lco_end` field must point to the first byte past the end of the compressed data.

Any subroutines that were included in the User-defined Code section can be accessed only through a program statement.

Stopping or Suspending an Experiment

An experimental run is usually stopped by a command from the user at a computer terminal. He might decide that there is enough of a data sample to stop then start a new run, or might want to temporarily suspend the run for some reason. LUCID allows a run to be stopped or suspended from inside an event. For example, the following event statement will cause the current experimental run to stop after this event is processed.

```
endrun
```

No further triggers will be handled, and no other events performed. Similarly, the following statement will cause the experiment to be suspended:

```
suspendrun
```

These statements allow an experimental run to consist of a precise number of events, or to contain data acquired over an exact time period; For instance, you could specify that a special event is to be triggered every 60 minutes. This special event could contain a single `endrun` statement.

Both statements will produce results identical to the corresponding commands from the LUCID program. In any case, a user must manually restart or resume the experiment by issuing the appropriate commands from the terminal.

Words Reserved for the Reader Description

LUCID will understand the following words as *keywords* when checking your READER description file. It will complain at length, probably cryptically, when you try and use any of them as variable names. LUCID considers upper and lower case characters to be different, so that n is not a keyword (and can be a valid, albeit confusing, variable) but defining N as a variable would cause a build-time error message.

A
B
C
F
N
all
and
ascii
beginrun
bin
binary
bins
binsize
bits
bool
boolean
by
call
camac
char
character
clear
command
compress
compressed
constantly
contains
data
dataway
decr
decrement
def
define
discard
display
elif
else
endgroup
endif
endloop
endrepeat
endrun
event
events
every
first
from
group
hist
histogram
if
incr
increment
inhibit
initialize
int
integer
integrate
into
label
lam
last
load
microseconds
milliseconds
minutes
notregion
or
previous
print
program
read
ready
real
rebin
region
reject
repeat
resumerun
save
seconds
some
suspendrun
then
timeout
to
trigger
uninhibit
using
wait
when
write

Chapter

4

The Looker

The LOOKER is the part of LUCID which lets you analyze experimental data. It is a program which LUCID writes for you, based on a description of what you want to do. It will analyze data *on-line* or *off-line*, depending only on where the READER is getting the data.

You might think of the LOOKER as a program that lets you look at your data. In fact, it doesn't produce any graphs or pictures at all. It only interprets and arranges the data so that other programs can produce displays. It is called the LOOKER because it looks at your data, not because it lets *you* look at it.

The LUCID program itself displays the information as the LOOKER is doing its work. This implies that the LOOKER need not be concerned with the type of display screen you're using, the type of graphs you want displayed, or even the kind of computer you have. Another program can be used to draw objects to make up a picture, and do the work of drawing scatter plots or contour plots. You should read Chapter 2 for more information on the LUCID command, and the different ways in which it lets you interact with your data.

How the LOOKER Works

The LOOKER extracts data from the LUCID data stream after the READER acquires and saves it. The original source of "raw" data could be CAMAC, or a tape or disk file, and the READER produces a data stream which is independent of that source. The LOOKER simply watches the data stream and picks out the events that are important to it.

The LOOKER description file consists of two parts: a `define` section to create new variables; and an `event` section, which contains the instructions that will be performed when specified events are recognized.

The LOOKER cannot define new event types and or save new events in the data stream. It can, when running off-line, filter out unwanted events or modify values in existing events for saving a reduced data set. The WRITER is also capable of eliminating certain event types from a data stream, as described in Chapter 5.

Making a LOOKER Description File

When you ask LUCID to `build` your experiment, it produces a LOOKER based on a description of what you want to analyze. Your description must be typed into a file, very similar in format to the READER description file which was discussed on page 3-2. For instance, the same rules for comments and "free format" statements apply. The most notable difference is in the name of the description file itself. It should have the same prefix as the READER description file, but must end with a '.l' to show that it's a LOOKER file. Thus, a READER file called He3.r should have an accompanying LOOKER file called He3.l.

As mentioned, the file itself has only two distinct parts, the `define` section, and the `event` section. As implied in the following example, they must appear in the file in that order, but each section may contain any number of lines:

```
#
# sample LOOKER description file.
# The assumption is made that the variables
# adc_value[12] and tdc_value (and events
# adcread and tdcread) were defined
# in the READER.
#
define adc_value[12],tdc_value previous data
define spectrum[12] histogram from 0 to 2048
define timings histogram from 0 to 1024 (512 bins)

event adcread:
        adc_value -= 102
        increment spectrum using adc_value
event tdcread:
        increment timings using tdc_value
```

In this example, the `define` section has only 3 statements; the first one tells the LOOKER that an array and a variable will be present in the data stream, which were previously saved by the READER. The second line defines an array of 12 histograms, and the third defines a single histogram; a *histogram* is described under LOOKER Histograms on page 4-26.

The two `event` statements introduce instructions which will be performed when those named events are encountered in the data stream. As the comments tell us, these events are named after the original event names in the READER description file. Furthermore, the variables `adc_value` and `tdc_value` are *not* defined in the LOOKER description file because they are READER variables passed through the data stream. As we'll see, you are allowed to define regular variables in the LOOKER, just as you could in the READER.

The DEFINE Section

LUCID allows you to define variables in the LOOKER, just as a general programming language would. The LOOKER also has access to every variable that was previously *saved* in the READER. The LOOKER automatically makes those variables available to you when it starts, or when they are changed during an experiment.

LUCID will warn you if you `define` a new variable for the LOOKER which you've also `defined` (and *saved*) from within the READER. Remember that only the names of variables that were *saved* by the READER are automatically defined for you.

If you don't want LUCID to automatically redefine the READER variables, you may redefine them manually using the keywords `previous data`, as shown below. This forces the LOOKER to make sure it is working with the correct READER.

Types of Variables

The LOOKER understands all types of variables that the READER knows about, except for CAMAC variables. When the LOOKER finds a CAMAC variable in the data stream from a READER event, it is treated as an integer (or array of integers). It knows nothing about accessing CAMAC or other hardware. On the other hand, the LOOKER understands different types of variables which the READER can't handle.

In any case, variables are defined in exactly the same way as in the READER file, as shown in the following example:

```
define energy real*4
```

This defines a single precision variable called `energy`. The same rules for defining READER variables apply in the LOOKER.

The format for defining variables is somewhat standard; the word `define` is followed by a list of names, which is followed by the type of variable:

```
define delta_e real*4
define hits,ecounts,fcounts int*4
define valid boolean
```

Notice that the word `integer` can be abbreviated to `int`.

Defining Data Stream Variables

Variables which are expected to be seen in the data stream may be defined in the LOOKER. Such variables have already been defined in the READER and defining some of them a second time in the LOOKER allows it to check that its using the right data stream. In fact new space for these variables isn't allocated as with other variables; the definitions are used only for "type checking". To define these READER variables, simply add the keywords `previous data` after the variable names. For example:

```
define d_a previous data
define o_tdc previous data int*2
```

The first example tells the LOOKER to make sure that some variable named `d_a` exists in the input stream. The second line makes sure that `o_tdc` exists, but also makes sure that it is of type `integer*2`.

Defining READER variables again in the LOOKER is good programming practice, but is not required. A good compromise is to define one or two commonly used variables from the READER.

Defining Histograms[1]

A histogram definition might appear as:

```
define tdc_hist histogram from 0 to 256
```

which creates a simple one-dimensional histogram. A two-dimensional histogram is defined similarly, such as:

```
define tdc_hist histogram from 0 to 256
                        and from 1024 to 2048
```

By default, histograms are created with each bin as type integer*4. This data type can be changed, for instance, if a histogram could have bin values beyond the range that this data type holds (roughly 4 billion counts per bin) then the histogram could be defined as

```
define tdc_hist histogram real*8 from 0 to 256
```

1. A more complete description of Histogram Definitions appears in LOOKER Histograms on page 4-26.

The type of the histogram bin may be any of integer*2, integer*4, real*4, or real*8.

The default number of bins in a one-dimensional histogram is 1024, and 128 for each dimension of a two-dimensional histogram. When this is inappropriate, the number of bins can be specified as part of the definition:

```
define tdc_hist_2D histogram
            from 0 to 256 (1024 bins)
          and from 1024 to 2048 (16 bins)
```

The range of **tdc_hist_2D** specified using from and to are *inclusive* on the from value, but *exclusive* of the to value. In the above example, the first dimension has a bin size of 0.25, the first bin ranges from 0.00 to 0.24, the second bin ranges from 0.25 to 0.49, and the last bin ranges from 255.75 to 255.99. Using range specifiers for first and last *includes* the high value of the range, causing the bins to be that much wider.

Defining Regions[1]

Although regions are used with histograms, they are defined independently in the DEFINE section of the looker. The important point to remember is that a region must have the same number of bins, dimensions, and array elements as the histogram that will be using the region. For the sake of clarity, regions should also have the same ranges defined as their corresponding histograms.

A simple two-dimensional region definition is almost identical to a histogram definition:

```
define tdc_region region
              from 0 to 256 (1024 bins)
            and from 1024 to 2048 (16 bins)
     contains:  (from 1024 to 1536) from 0 to 128
                (from 1536 to 2048) from 128 to 256
```

The tdc_region could now be used with the tdc_hist_2D in the earlier example. The contains: entries define a checkerboard, as shown in Figure 4.1: the low half of the Y range (in parenthesis) with the low half of the X range, and the high half of the Y range with the high half of the X range. The contains: entry is optional, and if excluded, the region defaults to the entire range.

1. A more complete discussion of using Regions can be found in Region Definition on page 4-34.

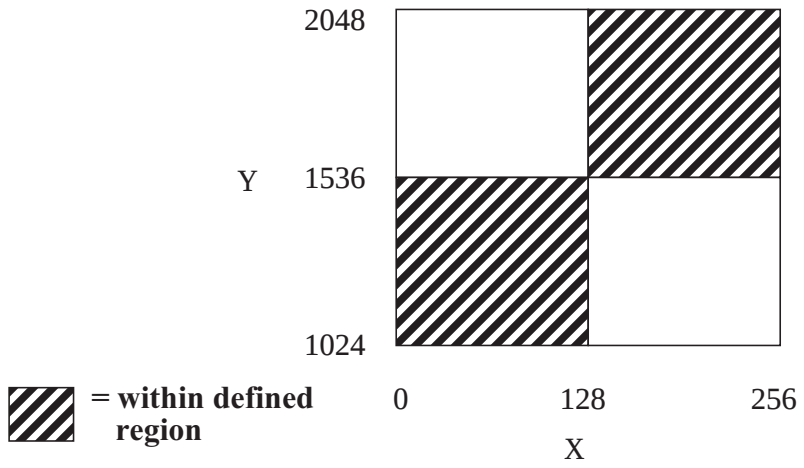


Figure 4.1:Example Defined Region

An array of regions is defined as above, with the addition of a contains: section for every array element. If there are insufficient entries, then the remaining array elements default to their entire range. Extra entries cause a warning message to be produced, and are otherwise ignored.

Defining Functions

The LOOKER allows calling C library functions and user-written C functions. Functions require a `define` statement to explicitly declare their argument and return types, e.g.

```
define log(real*8) real*8
```

The above example lets the LOOKER know that the C library function `log` takes a single argument of type `real*8`, and the return value is also of type `real*8`. Any function must be defined before use, and must have a return type. The use of argument typing is optional but recommended: the LOOKER will convert any argument of `log` that is not of type `real*8` to `real*8` before calling the function.

Functions that reside in the *buildname.c* file can also be defined and used. When writing these functions, be warned that the arguments are *call by value*, not *call by reference* as in the user-written subroutines.

Arrays of Variables

You can also define arrays of most types of variables, as in the READER:

```
define calib[1000000] integer*2
```

When defining a variable, the number of elements within the array must appear between brackets after the name. This example allocates room for a million integers, each containing two bytes. It also implies that you can have an unlimited amount of space. There is indeed a limit, and you should remember that using too much space can contribute to slowing down the computer, which is not desirable if the analysis is on-line. A LOOKER that takes up too much space simply will not run.

Array subscripts always start at zero, just as in the READER. For example, the array in the previous example can have subscripts from 0 to 999999. All variables are guaranteed to have a zero value after an experiment is rebuilt; but they are *not* zeroed out automatically with every new experimental run.

Groups of Variables

LOOKER description files may come to contain hundreds of variables as an experiment grows, and it may become difficult to deal with such a large number. To accommodate such experiments, the LOOKER recognizes groups of variables which can be treated as a common unit. Such an aggregate of variables is easy to define, as shown in this example:

```
define myhistograms group
    define tdc0[8] hist from 0 to 128
    define tdc1[8] hist from 0 to 128
    define tdc2[8] hist from 0 to 128
endif
```

The histograms within the group can be used as they normally would. The name myhistogram can also be used in a limited fashion, and refers to all of the histograms within the group. You could assign 0 to myhistograms, thereby clearing all of the histograms in the group.

Histograms and simple variables can be mixed within a group, as shown here:

```
define allvars group
    define x, y, z int*4
    define elec,spec0[12] hist from 0 to 2048
    define adc2[12] previous data
endgroup
```

Simple assignment of a value to the group `allvars` will set all variables, all histogram elements, and the current values of the event data `adc2` to zero (assuming the zeroing is occurring in an event where `adc2` was previously saved).

Some operations cannot be performed on `allvars` however, such as incrementing it as a histogram. If all variables within the group *are* histograms, like `myhistograms` was declared, then the increment operation works as if you were repeating the operation on each of the histograms in turn. The rules for using group names are restricted in the same way as using any other names; if an operation would fail on *any* of the members of the group, then the operation will fail on the entire group.

You are allowed to nest group definitions within other groups. For example, the following definitions are acceptable:

```
define allvars group
     define x, y, z int*4
     define histonly group
          define elec,spec0[12] hist from 0 to 2048
          define tdcspec[96] hist
     endgroup
endgroup
```

All variables would be accessible by using `allvars`. Only the histogram variables would be accessed when `histonly` was used. Although unlikely to be useful, one could increment all histograms at once with the same variable:

```
increment histonly using x
```

Another convenient use of variable groups is in the `load` statement. Rather than reading a single variable at a time, or listing out all the variables to the `load` statement, a group name could be used.

Remember that all variables are still accessible by referring to their names, regardless of which groups they are contained in.

The EVENT Section

The event section of the LOOKER description file is usually much larger than the define section because it tells LUCID what to do when a particular event is found in the data stream.

Event Types

The section consists of one or more event specifications, one after another in the description file. Each specification may contain many statements, but the first line must contain an event name, similar to a subroutine name in a programming language. For example, the statement

```
event electron:
```

associates all subsequent statements (up to the next event specification) with the event name `electron`. The statements which follow in the file will be performed whenever an `electron` event is encountered. The following example shows a very simple LOOKER description file which recognizes 3 different event types in the data stream:

```
define e1,e2,e3 int*4

event photon:
         e1 = 1
         print "Found a Photon Event!"
event calib:
         e2 = 1
         print "Found a Calibration Event!"
event scaler:
         e3 = 1
         print "Found a Scaler Event!"
```

This is an overly simplistic example, meant to show the layout of the description file. Notice that the event section itself is quite similar in form to that in the READER description file; refer to The EVENT Section on page 3-12.

The rest of the statements in the event specification are similar to the statements in most programming languages, allowing you to access any variables that were defined in the first half of the file, or previously in the READER. Histograms may be built up on an event by event basis, since each event specification is repeated for every event of that type which is found in the data stream.

Keep in mind that variables which were defined in the READER are only accessible if they were *saved* in the READER. Furthermore, such variables will only be accessible within the specification for the event in which it was saved. For example, let's assume that the READER had events named electron and hadron, and also an array called scin_adc which was saved within the electron event. A LOOKER description file might look like this:

```
define elec[12] hist
define scin_adc[12] previous data int*2

event hadron:
         print "Found an X-arm event!"
event electron:
         increment elec using scin_adc
```

The scin_adc array is used every time an electron event is found, and is used to build up a histogram. If an attempt was made to use scin_adc when a hadron event was encountered, an error message would be generated. LUCID knows that the READER didn't save scin_adc within the hadron event, and that the array has no meaning when such an event is found in the data stream.

The LOOKER allows instructions to be performed at other times, not necessarily based on a particular type of event being encountered. For instance, an event name is not required to appear before a specification:

```
event:
          x += 1
          print "Found SOME type of event"
```

These statements will be performed when *any* type of event is encountered. Remember that you won't be able to access any READER variables, since the LOOKER must know beforehand that the variable will consistently appear for the given event. This can't be guaranteed when *any* event can trigger the statements. This type of event specification might be used if you want some statistics about all events, for example (see above) the number of events processed.

Processing Some of the Events

If a LOOKER performs a lot of computation when finding an event which is quite common in the data stream, the entire experiment will slow down. The READER won't process any more data until both the LOOKER or WRITER are ready to accept it. Fortunately, the LOOKER allows certain events to be processed only when there is enough time to do so. Consider the following example:

```
define spec hist

event hadron (some):
          increment spec using adc
```

When on-line, the LOOKER knows that it needs to process only some of the hadron events, and will not hold up the READER while it works. If other hadron events go through the data stream while its processing a previous event, the LOOKER will ignore them. By default, the LOOKER processes all events with the given name, but the keyword `(some)` will make sure that the READER won't be held up while processing goes on.

Similarly, if no event name is given and the statements are performed for all events, the `(some)` keyword may be used. For example:

```
event (some):
          print "An event has been seen!"
```

The print statement will be performed when any event is encountered, but doing so will not keep the data stream from flowing.

When off-line, the `some` keyword is ignored, as it is most important that the looker sees all events.

Actions at the Beginning or End of a Run

Statements may also be performed whenever the beginning of a new run is encountered in the data stream. Important values and messages could be printed when the analysis is ready to start, for example.

```
define numevents int*4

event:
         numevents = numevents + 1

beginrun:
         print numevents, "seen so far"
```

This will produce a diagnostic at the beginning of each run specifying how many events have been seen so far in the data stream. You could also use a beginrun section to automatically reset histograms or clear software scalers that are being kept.

Similarly, it is possible to perform some actions when the end of a Run is encountered in the data stream. Simply use "`endrun:`" instead of "`beginrun:`" in the first line of the specification. Keep in mind that variables saved in the READER are *not* accessible within either of these specifications.

User Specified Commands

To this point, the instructions within a LOOKER file can only be performed after recognition of something in the data stream, such as event types or a Beginning of Run record. The LOOKER also recognizes arbitrary commands from the User when the data stream is being processed, so that internal status information may be printed, or variables may be reset.

The following example shows a common use for this option:

```
define spect hist

event hadron:
         incr spec using adc

command "reset":
         spec = 0
```

A histogram is incremented every time a hadron event is encountered, and that same histogram is cleared every time the user issues a "reset" command. Again, variables saved in the READER are not accessible from such statements. The commands may contain any type of text, and follow the same restrictions as were placed by the READER. For more information, refer to Keyboard Triggers on page 3-10.

In summary, there are five different "causes" which are recognized to start performing statements in the LOOKER:

```
event pion:                event pion (some):
event:                     event (some):
beginrun:
endrun:
command "calibrate":
```

Using Variables

Variables in the LOOKER can be used in the same way as they would be in the READER. For more information about what values are acceptable and how constants may be specified, refer to the READER section Using Variables on page 3-12.

Some notable differences from READER variables should be realized here. First, double precision floating point variables are accepted in the LOOKER. The following example demonstrates:

```
define d_energy[6] real*8
```

Second, histograms and regions are variables which the READER doesn't understand. These rather specialized variables are known only in the LOOKER, but they are treated as variables in most respects. For a description of these variables, refer to LOOKER Histograms on page 4-26, and Histogram Regions on page 4-33.

Third, the LOOKER knows nothing about CAMAC variables. If the READER has saved values from CAMAC variables, the LOOKER will only see their values as data without caring about their origin.

Many other features of using variables are the same as they are in the READER. For instance, entire arrays of variables can be used as single entities by simply leaving out the subscript. The following segment would clear the first element within the named array:

```
define junk[100] int*4

beginrun:
         junk[0] = 0
```

The following code segment would clear the *entire* array at once:

```
define junk[100] int*4

beginrun:
         junk = 0
```

Built-in Variables

The LOOKER has a few built-in variables, allowing easy access to status information. The variables, and the information they make available, are:

`LUCIDrun`	current run number
`LUCIDonline`	non-zero if performing on-line acquisition
`LUCIDbytes`	number of data bytes seen this run
`LUCIDtime`	(off-line) timestamp of data record
`LUCIDsequence`	(off-line) sequence number of data record

The `LUCIDtime` and `LUCIDsequence` variables only work when running off-line. The timestamp is a standard UNIX clock value, as returned by time()[1]. The number of bytes is a real*4 value, and the other variables are integer values. These variables can be used anywhere.

Calculations and Assignment of Values

Arbitrary expressions are allowed in many places in LOOKER statements, the most common being the assignment statement.

```
define new_e[20] integer*4
define laststep[20] real*4
define spect hist

event energy:
         new_e[0] = 10 * (laststep[0] /
                     130.2 + 2) ** 2
```

This example shows several features: mixed mode arithmetic is allowed; parentheses can be used for grouping; and lines may be broken into pieces when they become too long. The list of symbols which the LOOKER recognizes as valid operators is shown in Figure 4.2.:

Histograms cannot be used in expressions requiring a single result. For instance, assume the assignment statement in the previous example was changed to:

```
new_e[0] = 10 * (spect / 130.2 + 2) ** 2
```

LUCID would print an error message because this expression is expected to yield a single result, and the spect histogram contains several values.

1. The UNIX timestamp is the number of seconds since 12:00 AM GMT, January 1, 1970. There are a number of standard library routines that convert timestamps into usable values. See especially the UNIX manual entry for ctime() and localtime().

Symbol	Function
**	exponentiation
*	multiplication
/	division
+	addition
-	subtraction
<<	bitwise left shift
>>	bitwise right shift
%	integer modulus (remainder from division)
&	bitwise “and”
|	bitwise “or”
^	bitwise exclusive “or”
$	summation (integration)
~	one’s complement
-	negation
!	boolean “not”

Figure 4.2: Valid Expression Operators

The integration operator for histograms yields a single result, and may be used to sum all channels in a histogram, or all valid channels in a region of a histogram. The following example sets the variable sum to the total count in all bins in the pion histogram:

```
define sum int*4
define pion hist from 0 to 512 (512 bins)

event moredata:
         sum = $pion
```

The summation operator may also be used on regular variables; the result will be the value of the variable itself. More useful however, is that the operator may be used on an array. The following example will add all of the elements within the array named junk:

```
define tot int*4
define junk[100] int*4

endrun:
         tot = $junk
```

Notice that the array has no subscript. If it did, the summation operator would evaluate to the value of that single element. The summation operator can be used wherever a single value is required.

An expression may be used to yield several results at once, when entire arrays are used at one time. As mentioned in a previous section, you may clear arrays with one statement; you may also assign entire arrays with only one statement. Using the previous example, we could add the following statement:

```
new_e = laststep * 1.05
```

This statement assigns each element of the laststep array, increased by 5 percent, to corresponding elements of the new_e array.

This seems to contradict a previous statement which said the an expression is expected to yield a single result. A better statement is that an expression is allowed to yield as many values as the left hand side of an assignment statement can use. In other words, an assignment statement which assigns a value to a single variable must contain an expression which also evaluates down to a single value. If one assigns something to an array, then the expression may yield an array of values.

Assigning Arrays of Different Sizes

If an array of values is used in an assignment to another *smaller* array, then the elements of the larger array which are beyond the smaller's size will not be used. Consider the following definitions:

```
define large[20] int*4
define small[9] int*4
```

The statement:

```
small = large
```

results in the last 11 elements of the large array being ignored, and the first 9 being assigned to the corresponding positions within small. The statement:

```
large = small
```

results in all elements within small being assigned to the corresponding elements within large, and the remaining 11 elements being assigned the value 0.

If an array is assigned to a larger array, the elements which are missing from the smaller one are assumed to be 0. Using the previous variable definitions, consider this example:

```
large = small * 2 + 5
```

The first 9 elements of large will be set to 5 more than twice the value of the corresponding elements of small. The remaining 11 elements will all be set to 5, since the corresponding element's values in small are defined to be 0.

Assignment Operators

The LOOKER understands several variations on the assignment statement, based on binary operators. For example, the following statement adds 5 to the variable i:

```
i += 5
```

This is exactly the same as using:

```
i = i + 5
```

Similarly, to increase a variable by 7 percent, you could use:

```
x *= 1.07
```

To shift the bits in an integer 3 places to the left, you could use:

```
j <<= 3
```

Using Character Strings

Character string constants cannot be involved in calculations, and may only be used in a simple assignment statement into a character variable or passed as an argument to a subroutine or function. For example:

```
define name char*40

beginrun:
         name = "Low Energy"
         call histdisplay("Histogram 1", hist1)
```

Character variables may used in print statements, read in from the user, assigned to other character variables, or used as arguments to functions or subroutines. Further manipulation of strings can be done using C *"string"* functions, although this approach should only be taken if you are already familiar with the routines and their limitations.

Incrementing Histograms

Histograms are used to count the frequency of values within a variable[1], and the increment statement is used to do this. Consider the following section of code:

```
define adc previous data
define freq hist from 0 to 1024

event adc_event:
         increment freq using adc
```

1. For a complete discussion of histograms, refer to LOOKER Histograms on page 4-26.

This demonstrates the most common use of the increment statement. A variable which was saved in an event within the data stream is checked, and its value is used to increment the appropriate bin of a histogram.

A more elaborate example can be seen using arrays of histograms. Consider another example which uses an array of tdc values from the data stream:

```
define tdc[32] previous data int*2
define tdchist[32] histogram from 0 to
         2048 (2048 bins)

event tdcfired:
         incr tdchist using tdc
```

This example shows that an entire array of histograms can be incremented using the corresponding values from an array of tdc data, in a single statement. All 32 TDC inputs will be used in separate histograms. For more information about histogram arrays, refer to page 4-29.

Corrections to Measured Data

The increment statement can use an arbitrary expression when histogramming. For instance, the following increment statement is acceptable to the LOOKER:

```
define n_adc previous data
define n_adchist hist (512 bins)

event adcfired:
         incr n_adchist using n_adc * 1.09
```

This type of expression could be used to make corrections to detector inputs or normalize measurements to account for unique geometry. Remember that the expression can be arbitrarily complex. If the resulting value is not within the acceptable limits of the histogram, that value is *silently* ignored; no diagnostic messages will be issued, and no bins will be incremented. For more information refer to page 4-28.

Decrement instead of Increment

The LOOKER also understands the decrement keyword, which simply decrements the appropriate histogram bins in the same way that the increment statement increments them. It might be used after incrementing a histogram and discovering a mistake. The same restrictions apply to decrement as apply to increment.

Assigning Histograms To and From Arrays

A histogram can be converted into an array by simply assigning it. For example, the following segment transfers the values from the bins of the spect histogram into elements within the junk array:

```
define junk[1024] int*4
define spect hist from 100 to 200

endrun:
         junk = spect
```

The rules for assigning an array to another one of different size apply to histogram assignments to arrays; see page 4-15 for details.

Similarly, the values within array elements may be assigned to histogram bins. The rules mentioned above still apply.

Using Regions With Histograms

The LOOKER allows certain bins within histograms to be grouped together and treated as a “sub-histogram”. Regions have a multitude of applications, such as disabling counts from broken wires in a wire chamber or integrating bins of a peak in an energy spectrum. A region is simply a mask which allows operations to be performed on certain histogram bins, but not on others. Probably the most useful feature of regions is that they may be redefined dynamically, as the experiment runs. For more information on regions, refer to Histogram Regions on page 4-33.

Consider the following example:

```
define adc previous data
define espect hist
define rspect region

event scint:
         increment espect region rspect using adc
```

This increments espect using the value in adc. The rspect region has not been given a value in its definition, and is therefore set to the entire range of the histogram. Recall that by default, histograms and regions both have 1024 bins. Consider, if we were to change the region’s definition:

```
define rspect region
         contains:  0 to 300
                    400 to 800
```

The histogram bins would still be incremented, but only if the value was greater than 0 and less than 300, or greater than 400 but less than 800.

If you want to use the bins of a histogram which are *outside* those accepted by a region, the LOOKER allows you to use the notregion keyword. The increment statement from the previous example could be changed to:

```
increment espect notregion rspect using adc
```

This will increment bins which are outside of the rspect region's bins.

The LOOKER also understands the at symbol (@) as a region specifier. For instance, the previous example of:

```
increment espect region rspect using adc
```

could be change to a shorter form:

```
incr rspect@espect using adc
```

Using Arrays of Regions

Arrays of regions can be used with histograms or arrays of histograms in a variety of ways. Consider the following definitions:

```
define adc[32] previous data int*2
define x hist from 0 to 100 (100 bins)
define y[32] hist from 0 to 100 (100 bins)
define rx region from 0 to 100 (100 bins)
define ry[32] region from 0 to 100 (100 bins)

event newdata:
          incr rx@x using adc[0]
```

This seems quite straightforward, and increments bins in x using the first value in the adc array, via the region rx.

Similarly, we could increment 32 histograms using the corresponding values from the adc array, using an array of 32 regions:

```
incr ry@y using adc
```

If we wanted to increment all 32 histograms from the 32 adc values, but use the *same* region in each case, we could use:

```
incr ry[0]@y using adc
```

or

```
incr rx@y using adc
```

In the first example, any subscript could be used to specify a single region. Using an array of regions is allowed *only* in conjunction with an array of histograms. The following example is incorrect, and would cause an error message to be printed:

```
incr ry@x using adc
```

Remember that the histogram and not the region is being incremented, and the region can only work in conjunction with a histogram.

An array of histograms can be incremented by an array of variables; each variable within the array will be used to increment the corresponding histogram. If a single variable is used, then all histograms in the array will be incremented in the same way[1].

If they are arrays, histograms and corresponding data must have the same array size.

Incrementing Two Dimensional Histograms

Two dimensional histograms allow analysis of two variables at once. For a description of two dimensional histograms, refer to Defining Two-Dimensional Histograms on page 4-36. Increment statements are used in the same way, but two values must be given:

```
define adc[12] previous data
define tdc[12] previous data
define t_vs_e hist[64] from 0 to 1024 (64 bins)
         and from 0 to 256 (64 bins)

event newdata:
          incr t_vs_e using adc,tdc
```

Notice that arrays of histograms and data have been used, in the same way as they were with one dimensional histograms; the same rules apply.

The bits keyword may also be used in two dimensional histograms, with the same functionality. In fact, one variable may increment the histogram with the bit keyword, and the other use regular value increments. In any case, remember to put the word bits immediately after the expression for the corresponding axis. See Histogramming Bit Positions on page 4-40 for a further description of how this works.

Using Two Dimensional Regions

Two dimensional regions follow the same guidelines as one dimensional regions, as described on page 4-18. Arrays of regions are allowed, but the restrictions on array size apply here as they do with one dimensional regions.

1. All histograms in an array have the same limits, binsize and number of bins, by definition.

The If-then-else Statement

The If-then-else statement allows the flow of execution to be altered within the LOOKER description file. There are two general forms of the statement.

Simple Comparisons

First, an if statement contains comparisons between variables and constants. For instance, the following statements check that a scaler value is below 30,000:

```
define scaler[6] previous data int*4

event checkdata:
         if scaler[0] .lt. 30000
         then
               print "Overflow"
         else
               print "Scaler O.K."
         endif
```

The then keyword is optional, but the endif keyword is required.The else keyword and the statements associated with it are optional, too. An unlimited number of statements may be included within the then or else part of the statement, including other if-then-else statements.

Note that the comparison allows FORTRAN style keywords such as .eq., .ge. and so forth. Standard comparison operators are also allowed, as found in C, such as <, >= and so forth. More complex comparisons are allowed, such as:

```
if x .eq. 10 .and. y .lt. 100
```

Similarly, the C language conventions can be used:

```
if( x == 10 && y < 100)
```

The LOOKER accepts a mixture of them as well. Parentheses are not required around the comparison.

Histogram Acceptance Tests

The second form of the if statement is used to test the acceptability of histogram data. As described on page 4-17, any data which is beyond the limits of a histogram is silently ignored; no error messages or diagnostics are printed. If it is important to know when a variable had an acceptable histogrammed value, the if statement can be used. Consider this example:

```
define tdc[32] int*4
define eres[32] hist from 0 to 1024

event read_data:
         if incr eres using tdc
               print "Acceptable value"
         else
               print "Unacceptable value"
         endif
```

The increment statement itself will be true or false depending upon whether the variable was within the accepted range of the histogram. Similarly, if a region is used and the value is not within the accepted bins of the region, the condition is considered false. In this particular example, an array of histograms and values are used; if *any one value* of the array is outside the accepted limits of the its histogram, the increment statement is considered false.

The LOOKER also allows checking data acceptance without actually incrementing the histogram. The following example shows how a value can be passively checked without the histogram being changed:

```
define tdc[32],adc[32] int*4
define teres[32] hist from 0 to 1024 (64 bins)
         and from 0 to 64 (64 bins)

event read_data:
         if teres contains adc,tdc
               print "Acceptable value"
               incr teres using adc,tdc
         else
               print "Unacceptable value"
         endif
```

This example shows two variables being checked at one time for acceptance in a two dimensional histogram. In fact, arrays of variables are being checked; if *any* of the values in either array are outside the limits of the teres histogram, the if statement will evaluate to false. Accordingly, a single variable can be tested by simply using:

```
if nhist contains nvariable
```

Similarly, if the word bits comes after the variable, the bits will be tested for acceptance, rather than the value of the variable.

Keep in mind that the variables which are being used in these examples can actually be complicated expressions. For example, consider this if statement which checks two variables for acceptance in a two dimensional histogram:

```
if xyhist contains tdc[0]/tdc[1],adc[0]*0.98/adc[1]
```

The Repeat Statement

The LOOKER allows an arbitrary number of statements to be repeated, according to certain criteria. The standard form of the repeat statement can be seen in the following example:

```
define x int*4

endrun:
         repeat x from 1 to 100 by 10
               print "You should see 10 messages"
         endrepeat
```

The repeat statement must contain a variable, optionally followed by a beginning value, and ending value and a step size. All of the statements up to the endrepeat will be repeated so long as the variable is within the specified range. This particular example goes from 0 to 100 in steps of 10, changing the value of x at every iteration.

The following example shows an infinite loop:

```
repeat x
         print "Another iteration..."
endrepeat
```

The beginning, end and step portions of the statement are optional, allowing many variations:

```
repeat x from 1000 by 100
repeat x to 10000
repeat x by 2
repeat x from 1
```

The variable which does the counting may be changed within the loop, to affect the control as you would expect. For instance, you might explicitly set it to the upper limit, so the loop will terminate. Repeat statements are allowed to be placed within other repeat statements.

The variable's initial value should be less than the loop's final value, and the increment must be positive. This is because the looker always tests that the variable be less than or equal to the final value.

Printing Values from the LOOKER

The print statement is quite general and allows the printing of messages to the user's terminal screen. The messages may contain values from variables of any type. The following example shows the general format:

```
define x int*4

beginrun:
         x += 1
         print "Now starting Run ",x
```

Each item in the print statement must be separated from the next item by a comma, and character strings, constants and variables can be printed in any order. Remember that using an array without a subscript in a print statement will print the entire array.

If the user is not accessing the experiment when the print statement is performed, it will not be displayed but saved in the experiment's log file.

The print statement should be used with caution, since each event specification will be performed when the appropriate event is encountered in the data stream. There could conceivably be thousands of events processed per second, which implies thousands of messages printed per second!

Messages can also be printed directly to a file or to a user program:

```
define time(int*4) int*4
print "Run ", LUCIDrun, " at ", time(0) to "runlog"
print LUCIDrun to program "/home/exp000/bin/exparam"
```

Histograms can be saved to files using the write statement. There is little different between the write and the print statement: write requires that the output be directed to a file or program, and closes that file immediately after writing; variables saved by write are in a fixed predefined format; print allows constants and expressions to be saved; write only allows one variable per statement to be written.

The syntax of the write statement is:

```
write [compressed] [binary|ascii] variable to
[program] "destination"
```

Loading Histograms from Files or Programs

The reverse of the write statement is the load statement, which allows reading previously saved or software generated histograms into the LOOKER. The load statement expects input in the same format as that output by the write statement.

Setting Looker Variables Interactively

Users can set simple variables directly by using the read statement. A read statement looks like:

```
read variable [,variable ...] "prompt"
```

The user is prompted for input with the string *prompt* for each of the variables in the read statement. The value input from by the user takes effect immediately. Be warned that analysis stops while the LOOKER is waiting for input, and this can cause the entire experiment to pause waiting for a response. This is best used in a beginrun event for log information.

Using Your Own Subroutines

One important aspect of the LOOKER is the capability to perform your own subroutines whenever an event is encountered. Arguments may be passed to the subroutine, and may be variables, arrays or histograms. The following example shows how this is typically done:

```
define tdc[32] int*4
define adc[32] previous data

event wirechamber:
         call tracking( tdc, adc)
```

The word call must appear immediately before your subroutine's name. The parentheses *must* appear afterwards, whether variables are being passed to it or not. If arguments are given, they must be separated by commas.

This example shows a subroutine with two arrays being passed as arguments. The LOOKER currently understands subroutines written in C or FORTRAN, and will call them with the appropriate arguments. Histograms may also be passed as arguments, the size of the array will be equal to the number of bins in the original histogram. All arguments are call by reference, which is the normal method of passing arguments for FORTRAN. In C, all arguments must be pointers.

Subroutines may be called at any point in the LOOKER description file. If it prints something using a write statement (or printf in C), the message will be displayed on the user's display screen, or in the experiment's log file, as described under Printing Values from the LOOKER on page 4-24.

Your subroutines must exist in a file with the same name as your description file, but having the appropriate suffix. For example, if your description files were contained within demo.r and/or demo.l demo.w, your C language subroutines would be contained in a file named demo.c. Similarly, FORTRAN subroutines would be contained within demo.f. You are allowed to use both C and FORTRAN subroutines at the same time.

Unfortunately, errors in your subroutine are not handled gracefully in this version of LUCID, and the entire *experiment* will stop if your subroutine performs an illegal operation, such as an array subscript out of bounds, or an attempt to divide by zero.

The LOOKER has three user-callable routines available: Lucidsavehist(), Lucidloadregion(), and Lucidloadvars(). All these routines must be declared as *integer*4* within any looker program that calls them.

Lucidsavehist("*varname*", "*filename*", *flags*) is similar to the write statement. The main differences are: strings variables can be used to specify the variable name or the file name; selection of type of save (compressed, binary vs. ascii, save to a program, bin vs. value numbering) can be done at runtime; saving a mid-point value is optional. The variable name is passed as a string (either a string variable or a string constant) so the function can obtain all necessary information for saving the histogram.

Lucidloadregion("*regionname*", "*filename*") is like a load statement for a region. The input format looks like a region definition within the looker.

Lucidloadvars("*filename*") is similar to a namelist read in FORTRAN. Each line in the input file is expected to have the name of a variable declared in the looker, an optional subscript (which must be a constant), an equals sign, and a replacement value. The replacement values are converted to the type of the variable. String variables expect their values to be placed within double quote marks.

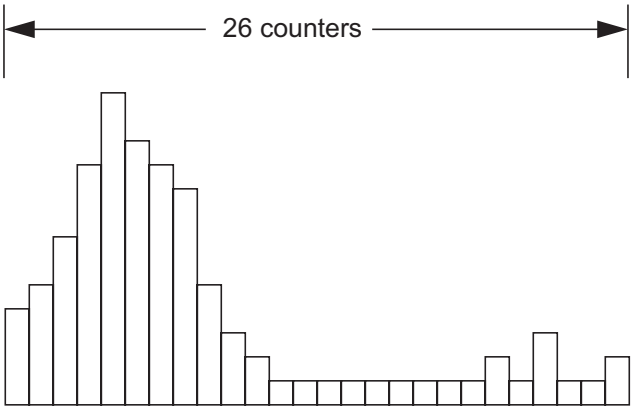
LOOKER Histograms

Probably the most common practice followed when analyzing data is to count the number of times that specific values are found in a variable. For instance, it might be important to know how many times a certain variable contained the value 103, or how many times the fourth bit position in a variable was set to a 1.

Histogram Definition

Normally, we have a separate counter for every possible value that a variable might contain; every time the variable changes, the counter which corresponds to its value is incremented by one. Keep in mind that a variable which contains N bits will need 2^N counters. For example, an 8 bit variable would require 256 counters. Similarly, a common `integer*4` variable would require more than 4 billion counters! Unfortunately, the computer can't manage 4 billion counters, so a compromise us usually made where several different values share the same counter.

Regardless of the number of counters, we can visualize this process by thinking of a "bar graph", with one bar representing each counter; the bars will increase in height whenever the variable is found to contain the corresponding values. For example, we can visualize a set of twenty-six counters like this:



As previously suggested, it is sometimes important to count the times when certain bits were set in a variable. The total value of the variable isn't interesting, because each individual bit might correspond to something different, such as a bank of detectors firing. When counting bits instead of variables, a 32 bit variable (`integer*4`) would have only 32 counters associated with it. Note that several counters might be incremented at once when a variable is checked; the value of 5 (binary 101) would cause 2 different counters to count at the same time.

In any case, a collection of counters is called a *histogram*, and LUCID lets you define and use them as you would regular variables. A histogram can count the occurrence of any value or count the changes to any bit position within a variable. Ideally, a histogram will count these different values whenever the variable changes.

This action of determining the frequency at which different values occur is a fundamental feature of the LOOKER.

Bins

Each counter within a histogram is often referred to as a *bin*. Usually, a histogram won't contain more than a couple of thousand bins; LUCID lets you decide how many bins a histogram will contain, and therefore what range of values will be counted by each bin.

The rule to remember is that all bins in a LUCID histogram have the same width.

Each individual bin must therefore have the same range of values, but the entire histogram is allowed to count values in any range that can be contained in the variable. The *resolution* of any histogram is defined to be the width of a single bin, or the range of different values a bin will accept.

Defining Histograms

The following example shows how histograms can be defined:

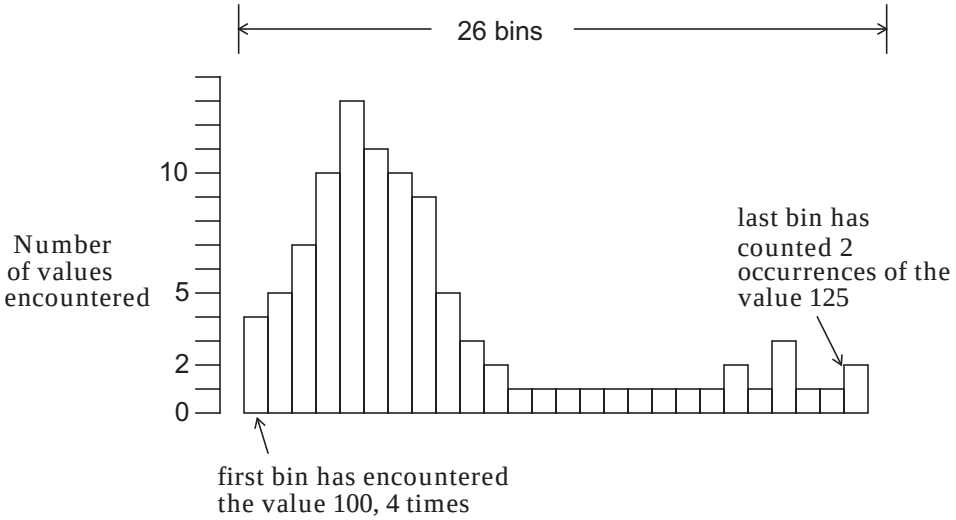
```
define e1,e2,e3 histogram from 0 to 1024
```

This allocates room for three histograms whose names are `e1`, `e2` and `e3`. Each one can count values from 0 up to *but not including* 1024. If one of these histograms were given a variable with a value of 1029 for example, it would simply ignore it. By default, LUCID gives every histogram 1024 different bins. Therefore the default resolution is $1/1024^{th}$ of the histogram's acceptable range. The keywords `from` and `to` name the limits of values accepted by the histogram. Remember that the value of 1024 will *never* be counted in this histogram. For now, we will assume that only integer variables are being counted, but real variables (floating point) can also be used, as described on page 4-31.

Let's look at another example:

```
define pions histogram from 100 to 126 (26 bins)
```

Imagine how this histogram would look after it has monitored some variable for awhile. The height of each vertical bar corresponds to the number of counts in each bin:



The definition says that only the values between 100 and 125 inclusive are important, and values greater than *or equal to* 126 will not be accepted. It seems apparent that *most* of the values that were checked so far were closer to 100 than 126, because those bins have higher counts. We can also tell that 98 acceptable values were encountered, because that is the *total* number of counts in *all* bins.

Number of Counts in each Bin

Bins can count as high as 4 billion before they "wrap around" to 0. If you think that your experiment might be very fast or lengthy, such that it will check billions of values, then you may redefine your histograms to count to a higher range. Typically, a bin is implemented as a 32 bit counter, but the LOOKER also allows the bins to be real numbers. The following example shows how this is done:

```
define pions real*8 histogram from 100
         to 126 (26 bins)
```

Keep in mind that the most efficient counter is the default[1] which the LOOKER provides. Any other type of bin is *rarely* needed.

Histogram Arrays

You're also allowed to have arrays of histograms.

```
define d1[10], d2[10] hist from 0 to 2048 (512 bins)
```

1. The default bin is of type int*4.

This example allocates two arrays of histograms, which will count values only when between 0 and 2047 inclusive. There are actually 20 separate histograms defined in this statement, making up two separate arrays. Remember that arrays of variables are accessed with subscripts that begin at zero, and arrays of histograms are treated similarly.

In this same example, each histogram will contain only 512 bins, as specified with the keyword `bins`. Therefore, every bin will count the occurrence of four consecutive values. For instance, the first bin (bin 0) would be incremented when a value of 0, 1, 2 or 3 is found. Accordingly, the last bin will be incremented when a value of 2044, 2045, 2046 or 2047 is encountered. Also notice that the word `histogram` can be abbreviated to be `hist`.

In the following example, we define a histogram with 32 bins, counting values from 0 to 31 inclusive; This would probably be used to count changes in bit positions.

```
define pattern hist from 0 to 32 (32 bins)
```

Whether counting bit positions or different values within a variable, the histogram is defined in the same way; refer to page 4-40 for a description of counting bit positions.

Specifying Histogram Bins Instead of Limits

For convenience, LUCID also recognizes definitions based on the first and last bins used in a histogram, rather than the limits of variable values accepted. This is better shown in the following example, where the keywords `first` and `last` are used:

```
define spectrum first 0 last 1023 (1024 bins)
```

This describes a histogram in which the first bin starts at 0, and the last bin *starts* at 1023. Defining a histogram in this way ensures that the upper "limit" is included in the counting procedure. The following definition would result in exactly the same histogram:

```
define spectrum from 0 to 1024 (1024 bins)
```

The keywords `from` and `to` may seem more natural, but might seem ambiguous in some cases, since the upper limit is never really included in the counting. Using the `first` and `last` keywords introduce other conceptual problems however, such as the acceptance of values within each bin. This is best shown by introducing another keyword option, described in the next paragraphs.

Specifying Bin Widths

In many cases, you aren't concerned with the number of bins in a histogram, but rather with the *acceptance* of each bin. In other words, you'd like to be able to say that each bin will count values in a certain range. The following example shows how this can be specified:

```
define spec hist first 0 last 200 (binsize 2)
```

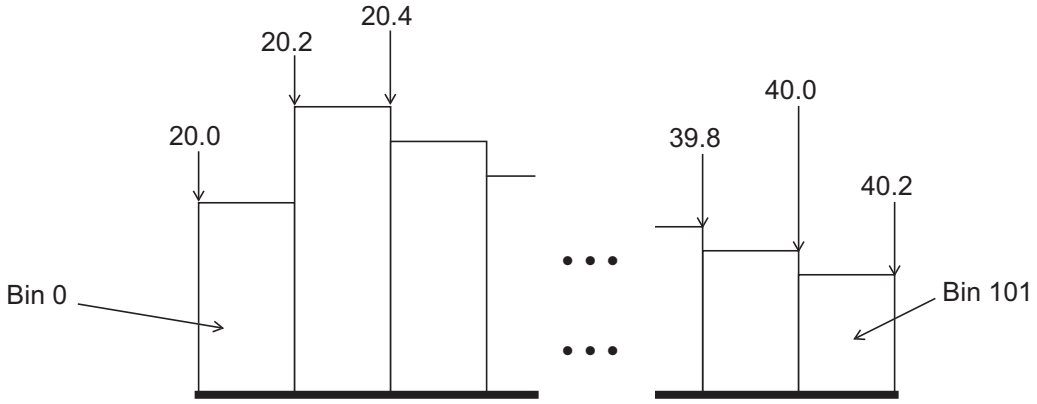
It may not seem obvious at first, but this histogram actually has 101 bins. It will accept values from 0 to 200 inclusive, with the first bin counting values from 0 up to but not including 2. Therefore 101 bins are set up because the range of 0 to 200 would represent 100 intervals, and 200 marks the start of the *last* bin (which counts the values 200 and 201), making 101 bins. The next example further explains this idea.

Counting `real` values

In all of the examples so far it is assumed that only integer values will ever be checked, but it is also possible to count values contained in a `real` variable; LUCID makes no distinction as to what type of variable it is counting. In many cases, the range of acceptable values in a histogram could be a very large or small interval. For example:

```
define de[5],energy hist
                     first 20
                     last  40 (binsize 0.2)
```

Here we have an array and a variable in which only values between 20 and 40 inclusive will be counted, and which contain bins which are 0.2 counts wide. In other words, any value between 20.0 and 20.2 will be counted in bin 0.



Strictly speaking, the value would have to be *less* than 20.2 to be counted in bin 0, since 20.2 marks the first value accepted in bin 1. Due to the way the LOOKER calculates bins for incrementing, `Bin 101` will contain counts of values >= 40 and <40.2, which goes against the "intuitive" counting of only the value 40!

How Histograms Count

This brings up a very important point in the LOOKER. A variable's value is always included in the bin having a starting value which is lower than the variable's value. In other words, values are always rounded down to the next lowest bin's starting value[1].

In reconsidering the earlier example of

```
define spec hist first 0 last 200 (binsize 2),
```

bin 1 counts any values from 2 up to (but not including) 4, but the *last* bin would actually *start* at 200. Using the `first` and `last` keywords, the histograms could actually count certain values which are *larger* than 200; a value greater than 200 and less than (but not including) 202 will be counted in the last bin.

Counting Negative Values

Negative numbers can be used to describe the range of acceptable values. For example:

```
define posn histogram first -1000 last 1000
```

This defines a histogram which will count values between -1000 and 1000 inclusive, thereby containing 2001 bins. The only restriction on the limits for acceptable values is that the `first` value is numerically less than the `last` value. LUCID will warn you if the limits are backwards.

Default `first` and `last` histogram values

Recall that by default, LUCID will allocate 1024 bins in a histogram, if no bin or binsize values are given. You may also omit the `from` or `to` keywords if you want. The default values of limits for the first and last channels are 0 and 1023 respectively. The following example will therefore define a histogram with 1024 bins, the first one being 0 and the last being 1023.

```
define e1 hist
```

Restrictions

Some other restrictions should be mentioned here. The binsize value can't be negative, and must divide the entire histogram into an *exact* number of bins. In other words, the difference between the last and first bins must be a multiple of the binsize. LUCID doesn't care if you have a histogram with only 1 bin, but it will complain if the bins don't divide the total histogram into equal pieces. For example:

```
define p1 hist first 0 last 99 (binsize 8)
```

1. To have the values *rounded* to the nearest bin boundary (higher or lower), you can add one half of the bin width (half the *resolution*) to the variable before histogramming.

This implies that 12 bins have a width of 8, and the last one has a width of 4. Since all bins must be the same width, LUCID would print an error message for this definition. Probably the best fix would be to use:

```
define p1 hist first 0 last 99 (13 bins)
```

This histogram would have 13 bins between 0 and 99 inclusive. Since the value 99 names the last bin, there are actually only 12 bins which are less than 99; Each bin is therefore 8.25 units wide. The last bin will accept values from 99 up to (but not including) 107.25.

This example also shows some of the shortcomings of using the `first` and `last` keywords; most of the time they aren't intuitive and don't fit the concept that you may have of the histogram. Using the `from` and `to` keywords are recommended most of the time.

One-dimensional histograms default to 1024 bins. The maximum number of bins in a one-dimensional histogram is 4096. Two-dimensional histograms default to 128 bins per axis. The maximum number of bins in a two-dimensional histogram are 256 bins per axis. These limits attempt to keep the virtual memory requirements of LOOKER programs from becoming excessive. When you need to look at an area of a histogram in greater detail, try restricting the X or Y range to just the area of interest, or define multiple histograms with adjacent ranges incremented from the same variables.

Using Histograms

In the `event` section of the LOOKER description file, histograms can be used in expressions just as other variables would. Let's skip ahead for a moment and see how you can actually get a histogram to count. Using the histogram from the previous `define` example, we could say:

```
increment p1 using adcval
```

This tells the LOOKER to check the value of the `adcval` variable, and increment the corresponding bin within the `p1` histogram. For a complete description of the increment statement, refer to Incrementing Histograms on page 4-16.

Histogram Regions

LUCID strictly enforces the rule that all bins in the same histogram must be the same resolution. Sometimes it is important to deal with a range of values which is *larger* than a single bin's acceptance, but still keep the resolution of a single bin. In other words, we need the ability to deal with a group of bins which

could make up just a small piece of the total histogram. To do this, LUCID allows you to define *regions* of a histogram. These regions can be treated as histograms unto themselves, but actually use the bins of other histograms. Regions cannot have any values on their own, and can only be used in conjunction with a regular histogram. It is best to think of a LUCID region as a *mask* through which we can see certain bins of histograms. This will be further discussed, but we'll first consider how regions can be defined.

Region Definition

The following example shows how to define a region:

```
define tdc,rtdc hist from 0 to 1024
define tval1, tval2 region from 0 to 1024
```

The first line defines two regular histograms, but the following line describes two regions; the keyword `region` is simply used instead of `histogram`. All other options to define histograms can also be used to define regions, and as mentioned under Defining Histograms on page 4-28, the default number of bins is 1024 if no other indication is given.

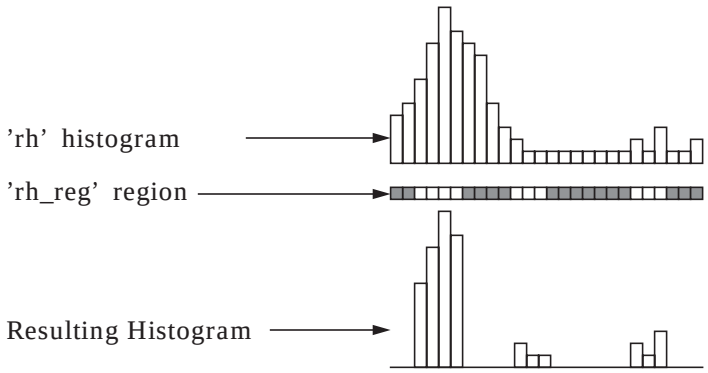
Consider another example with fewer bins:

```
define rh from 0 to 26 (26 bins)
```

Let's assume that only a few bins should be considered acceptable; bins 2 through 5, 10 through 12 and 20 through 22. We would define a region to mask out the unacceptable bins:

```
define rh_reg region from 0 to 26 (26 bins)
                     contains: 2 to 6
                               10 to 13
                               20 to 23
```

The following diagram shows how we can best visualize this region and its relationship to the histogram named `rh`:



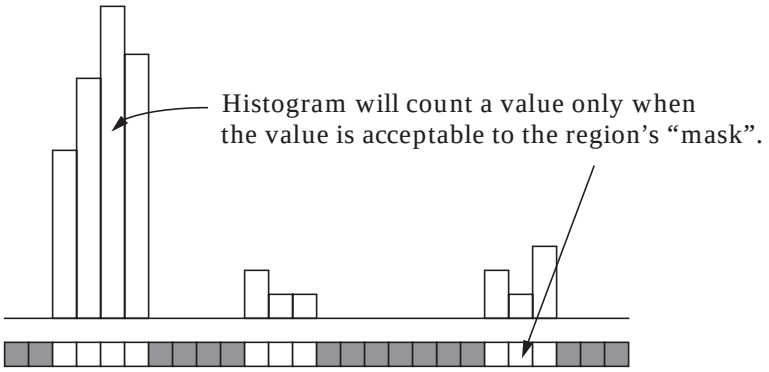
The dark areas in the region represent masks through which bin values will not pass[1]. Using this region with the `rh` histogram produces a second histogram containing only certain bins of interest.

Using Regions

A region can be used wherever a histogram is used, by simply giving the original histogram name, the `region` keyword, and the region's name. Recall the example with the `increment` statement on page 4-33; we can give a similar example using a region:

```
increment rh region rh_reg using adc
```

This statement tells the LOOKER to check the value within the `adc` variable and increment the correct bin in the `rh_reg` region of the `rh` histogram. In other words, the `rh` histogram is incremented as it would normally be, but only if the `adc` value is within the range of acceptable bins in `rh_reg`.



For a complete description on how to use the increment statement, refer to Incrementing Histograms on page 4-16; for more information on how to use regions, refer to Using Regions With Histograms on page 4-18.

Region Arrays

Arrays of regions may be defined, just as arrays of histograms. A region cannot be used by itself however, and an array of regions is usually accompanied by an array of histograms. Assume the following definitions:

```
define tspectrum[12] histogram
define tvalid[12] region
```

1. Remember that bin numbers start at zero.

Regions and histograms both have 1024 bins by default. If we assume that an array of 12 `tdc` values exist, we could increment all 12 histograms using the 12 corresponding tdc values, but only if they are acceptable to the corresponding 12 regions. This rather elaborate process is described in detail later, but keep in mind that only one statement is needed to do all the work:

```
increment tvalid@tspectrum using tdc
```

Note the alternate method of naming a region and histogram pair; using `tvalid@tspectrum` is the same as using `tspectrum region tvalid`.

Basically, regions make it very easy to determine if variables contain acceptable values, and can therefore let you *cut* unacceptable events from the data stream, or increment histograms based on ranges of values.

Region Restrictions

Some restrictions should be mentioned at this point. In particular, a region must have the same number of bins as any histogram with which it will be used. The notion of `binsize` has no meaning to a region, since it may be used to mask out histogram bins of arbitrary resolution; it is only important that the region and histogram have the same number of bins. Actually, bins can be defined without limits, and can therefore be defined like this:

```
define validbins region (26 bins).
```

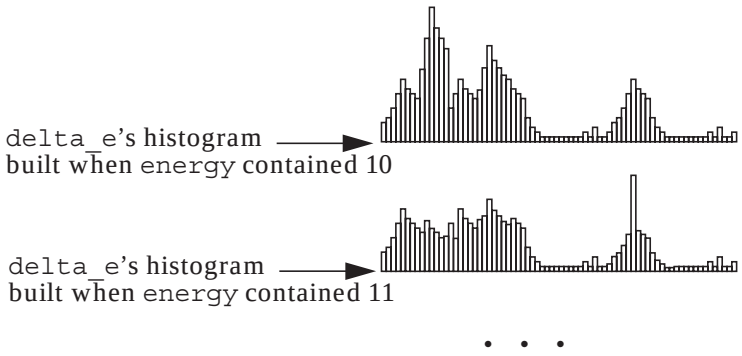
This `validbins` region can be used with any histogram having 26 bins. Many programmers include limits in region definitions, specifically the same limits as are used in their associated histograms. This makes it easier to see which will be used together.

Defining Two-Dimensional Histograms

Histograms can be very useful when analyzing data. They are quite simple and essentially describe all of the changes made to a variable. We often need to know more information about a variable, particularly with respect to other variables. For example, we might want to count the occurrence of different values only when some other variable has a value between say, 10 and 20. In general terms, we want to count the occurrence of different values in two different variables at the same time.

Example

As an example, let's consider a variable called `energy` which can contain values between 10 and 19 inclusive. A histogram of this variable would be very simple to define, since it would contain only 10 counters. Now assume that every time this variable changes to a new value, a second variable `delta_e` also changes, and it can contain values from 0 to 2047. Ideally, we would like to keep 10 different histograms for `delta_e`; for every value that `energy` contains, we want to see how the `delta_e` values change.

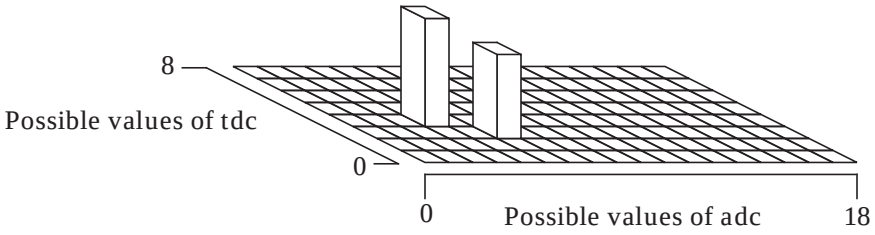


Eight other histograms, corresponding to the remaining possible values of `energy`

One solution is to simply define an array of 10 histograms which would count `delta_e` values, and update the appropriate histogram by using the `energy` value as a subscript. Of course, we would have to subtract 10 from the `energy` value to get the correct subscript, but it would work fine. The ultimate goal from such a procedure is to determine how `delta_e` values relate to `energy` values, if at all. For instance, we might be able to determine that `delta_e` values are higher whenever `energy` has a relatively low value.

LUCID allows an easier method of relating one variable's values to those of another. A two-dimensional histogram can be defined which allows two variables to be considered at the same time. It is assumed that both variables will change at the same time, because usually they are both part of the same event from the READER. Functionally, a two-dimensional histogram will do exactly what the array of histograms did in the previous example, and we can visualize it as a matrix of bins, rather than a simple list of them.

For example, consider having to count values ranging from 0 to 18 in one variable named `adc`, and a second variable named `tdc` having values from 0 to 8. A matrix of bins could be visualized like this:



In this particular representation, the histogram counted several instances where `adc` contained the value 5 when `tdc` contained a 2. Even more counts were made when `adc` and `tdc` both contained the value 3.

One advantage of using a two-dimensional histogram is that both variables can have ranges of arbitrary values, as a regular histogram does. The following example defines this type of histogram:

```
define ta_common hist from 0 to 18 and 0 to 8
```

The two ranges of values are separated by the keyword `and`, which tells LUCID that `ta_common` is a two-dimensional histogram. This particular example shows an alternative to using the array of histograms for `delta_e` in the previous discussion, which used `energy` values as subscripts. Visually, we can think of this histogram as a graph with an X-axis from 0 to 18, and a Y-axis from 0 to 8. Technically, when new `adc` and `tdc` values are found, the corresponding bins in each axis are determined, and the intersecting counter is incremented.

Two-Dimensional Histogram Arrays

Arrays of two dimensional histograms are allowed, just as with any other variable, but one common problem to avoid is using too much memory. If we use the example of a 1024 bin histogram[1] on each axis of a two-histogram, the result is over a million counters! Usually, such histograms should have very few bins in each axis to conserve memory space.

The next example shows an array of 2D histograms, using different bin sizes for each axis:

```
define xsect[10] hist from 103.4 to 110.6 (64 bins)
            and from 150 to 250 (binsize 1)
```

1. The LOOKER does not allow two-dimensional histograms to have more than 256 bins per axis. See page 4-33for further discussion on this restriction.

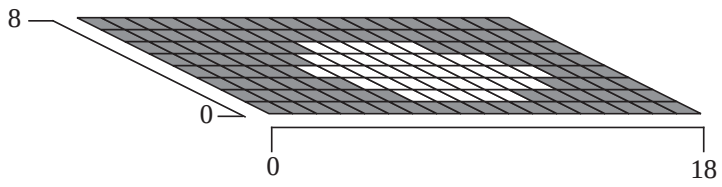
We see that the definition can become quite long. Remember that the lines are free format, so you can wrap long statements onto the following lines in your description file. This example defines an array of 10 two-dimensional histograms, and has the first range of values divided into 64 bins, the second divided into 100. The definition of a 2D histogram must have both ranges of values defined. The 'bin' and 'binsize' keywords are optional, and the default is 128 bins if not otherwise specified.

Defining Two-Dimensional Regions

Two-dimensional histograms may also be used with two-dimensional regions, as regular histograms can. They're a bit more difficult to describe, because for every "region" on one axis, there must be a second region specified for the other axis. Let's define a region which could be used with the `ta_common` histogram from the previous example:

```
define protons region from 0 to 18 (18 bins)
                        and 0 to 8 (8 bins)
            contains: (7 to 12) 1 to 6
                      (6 to 14) 2 to 5
                      (5 to 15) 3 to 4
```

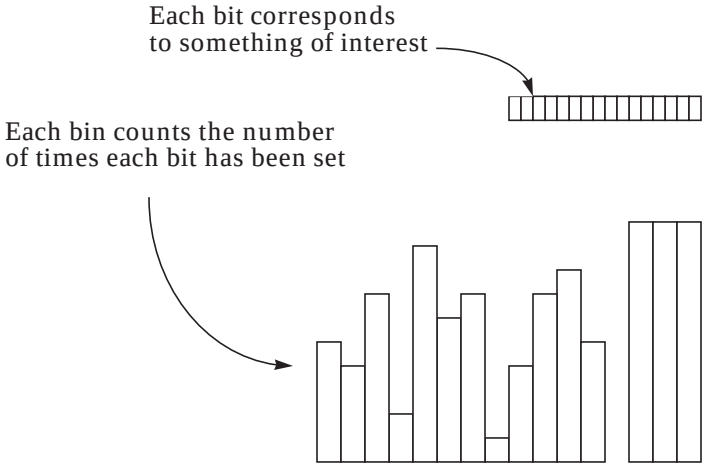
The following diagram shows how we can imagine this region with respect to the `ta_common` histogram from page 4-38:



The region has the same matrix structure of the histogram, but in this diagram the white bins represent the acceptable bins from its definition. If this region was used with the `ta_common` histogram from page 4-38, then only the histogram bins corresponding to the white bins of the region would be accepted. In short, such regions will allow more elaborate cuts to be made to your data. Analysis can thus be limited to an area of particular interest.

Histogramming Bit Positions

Histograms typically count the occurrence of different values within a variable. The LOOKER allows histograms to count the frequency of bit values within an integer. For example, an experiment might contain an input register which has 16 bits in a word corresponding to 16 physical actions taking place. A common practice is to count the times each particular bit is being set:



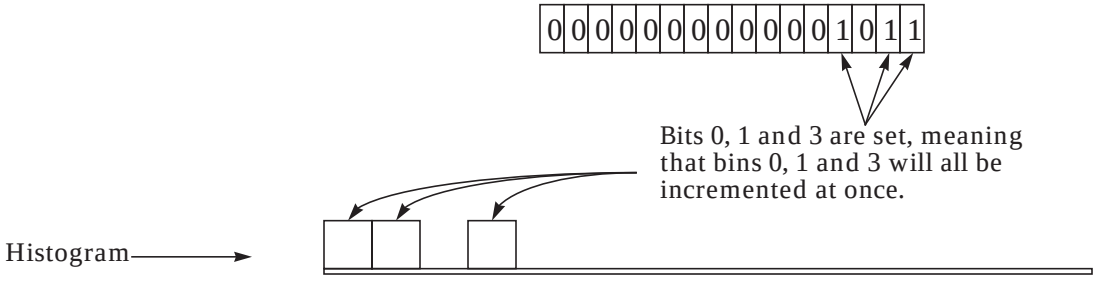
For example, the least significant bit (rightmost in the diagram) in the integer might represent the occurrence of a special hardware trigger; if the bit is set[1], the trigger has occurred. The second bit could be set if a calibration signal was present somewhere. In any case, the LOOKER allows you to count the occurrences of these bits being set by using the bits keyword:

```
define inreg previous data int*2
define bitcount hist from 0 to 32 (32 bins)

event readbits:
         incr bitcount using inreg bits
```

1. If a bit is set, it contains a 1. If cleared, it contains a 0.

The definition of the histogram is the same as any other histogram, but the increment statement is slightly different. The bits keyword tells the LOOKER to increment bins using the bit positions which are set in inreg. Notice that several bins could be incremented at once, since a particular value within inreg might have several bits set. For instance, if the integer contained the value 11, the binary representation would be 001011, and three bins would be incremented at once:



The least significant bit in the integer will cause bin 0 to be incremented. The second least significant bit will increment bin 1, and so forth.

LOOKER Keywords

There are a number of keywords reserved by the looker. These must not be used for variable names, or great confusion (and an unsuccessful LOOKER build) will result. The LOOKER distinguishes between upper and lower case characters in keywords and variable names, so f is a valid variable name, whereas F causes an error as a reserved keyword. Not all the keywords reserved by the LOOKER are actually *used* by the LOOKER; some of these keywords are part of the READER language. Keywords marked (*) are reserved words in other parts of LUCID, and should be avoided. Keywords marked (^{X}) are keyword abbreviations, and would best be avoided as they may disappear in a later version of LUCID.

A *
B *
C *
F *
N *
all
and
ascii
beginrun
bin
binary
bins
binsize
bits
bool X
boolean
by
call
camac *
char
character *
clear *
command
compress X
compressed
constantly *
contains
data
dataway *
decr X
decrement
def X
define
discard *
display *
elif *
else
endgroup
endif
endloop
endrepeat
endrun
event
events
every *
first
from
group
hist X
histogram
if
incr
increment
inhibit *
initialize *
int X
integer
integrate
into *
label
lam * X
last
load
microseconds *
milliseconds *
minutes *
notregion
or
previous
print
program
read
ready *
real
rebin
region
reject
repea t
resumerun *
save *
seconds *
sizeof
some
suspendrun *
then
timeout *
to
trigger *
uninhibit *
using
wait *
when *
write

Chapter

5 The Writer

The third part of LUCID is called the WRITER, and its sole function is to record the data produced by the READER and LOOKER. LUCID runs the WRITER when an experiment is built, and the user is responsible for informing the writer of the destination of the data.

Although the WRITER is only concerned with saving data, it is capable of saving it in a variety of ways. For instance, you can tell the WRITER to write certain events on a tape using LUCID, or to write only as many of certain event types as will keeping the acquisition from blocking.

Unlike the READER and LOOKER, there is no WRITER description file. This allows changing writer destination information without requiring a rebuild of the experiment. However, destinations can only be changed when data acquisition is in a STOPPED state. This helps to ensure the integrity of a run's data.

Data is saved only is a LUCID-specific format. There are functions available to read this data into user-written C or FORTRAN programs, see Reading LUCID Data Files on page 7-10. LUCID also reads this data when running an off-line experiment, which is the easiest way to do more intricate calculations on your data.

The rest of this chapter assumes familiarity with the *Build Writer* window described on page 2-11, or with the Modify writer list option of the terminal-oriented lucid command, described briefly in The lucid Command on page 7-8.

Saving Event Data[1]

Saving ALL Events

Although it is very easy to save different events in different places, the most common requirement of the WRITER is to write *every* event that it finds in the data stream. *Every* event that the READER can produce is saved, thereby writing *all* data to tape.

You're allowed to mix events and destinations in any fashion. For example, all events could be written to tape, but certain scaler events could *also* written into a disk file.

Critical Data Destinations

The WRITER allows data output files to be flagged as *critical destinations*, as set by the Fail on Error setting in the writer window. When this is set for a data destination, the LUCID experiment will stop when a write error occurs on output. This could be due to filling a tape or disk partition, to an error on the output media, or an output program terminating. After such a failure, the experiment must be restarted. The resulting output files are possibly difficult to use: end-of-run summary records are missing, as are the LUCID-supplied end-of-run marks.

When there is only one output device, Fail on Error should be set. With multiple devices, there are trade-offs. Fail on Error can be set according to the results required.

Data Shadowing

The idea behind data shadowing is to write the same data to two or more destinations. All destinations are considered non-critical. If one fails, the user can stop the experiment normally, and all other destinations will have proper end-run records and appropriate file marks. The drawback of the approach is the requirement for *constant* vigilance. The experimenter must monitor the message display window continuously, watching for an error message from any of the writers. The risk is that all the messages could be missed, and once all writers have failed, the experiment still continues merrily along, sending all the data to the bit bucket!

One Fail-on-error

The safest approach when acquiring data is to have only one device selected to fail on error. This device should be writing all occurrences of all events. When this device fails, the acquisition will shut down and demand attention from the experimenter.

---

1. At the time this section was written, the WRITER software that distinguishes individual event requirements was not completed. It is hoped that the delay between documenting this software and writing it is minimal.

Efficiency

The WRITER will work most efficiently when writing every event to the same place. When it has to look at an event and decide where it should be written, the entire data stream will be forced to slow down. It is best to save *all types* of acquisition events to tape or disk, and then separate events later with a second off-line LUCID data stream.

The advantage of using the LUCID data file format is that your data file will be completely self-contained. Variable names, event descriptions and your original description files are saved to the data stream whenever an experiment is started and then every time you change a description file and rebuild. This means that you could tell LUCID to generate software to read an old data tape by simply telling it which tape drive to use! It will check the tape, extract the original description information, then write the software to read the rest of the data.

Examples

The most common interaction with the WRITER is changing tapes in the destination tape drive. This is accomplished in the following steps:

Changing Tapes

- Stop the current experiment (but don't shut down)
- Terminate the tape drive writer. In XLUCID, toggling the SAVE state of the destination and APPLYing the change is sufficient. In lucid, the tape drive must be deleted from the list and the list sent to the manager.
- After the writer process for the tape drive exits (this is indicated by a diagnostic message), remove the tape from the drive and insert a blank tape.
- Add the tape drive back to the list of destinations. In XLUCID, toggling the SAVE state on and APPLYing is sufficient. In lucid, the tape drive must be re-added to the list of output devices.

Multiple Tapes

The most common use for multiple tapes is Data Shadowing. It is only necessary to specify two output tape destinations that accept all occurrences of all events. Both destinations should have identical Fail-on-error settings. Having both fail-on-error increases the chances of a tape write error aborting a run. If both silently exit, there is a remote chance that both destinations

could stop writing with only a message to the view window. If the experimenter misses both termination messages, the experiment would continue running but would not record any data. The only other feedback is the *Writer: Bytes written this run* box, which would stop updating the number of bytes acquired.

Chapter

6 Conceptual Overview

How LUCID Works

When you tell LUCID to `make` an experiment, it takes your description files and writes several computer programs. Each of the READER, LOOKER and WRITER are separate programs which LUCID may produce, along with a separate program to access CAMAC if that was specified. These programs are compiled, linked and executed automatically when you say `make`, but only if it's necessary. For example, if you change your READER description file slightly, then only *part* of the READER program will be rewritten, recompiled and fit into the system. In any case, the programs are started and after setting up communications with one another, will wait for your command.

The data stream is on-line in this case, and the data is placed in *shared memory*. The experimenter may place certain restrictions on analysis in this case; for instance, he is allowed to have the LOOKER ignore some of the data in the interest of speed.

The Play button tells the READER to replay data which has been acquired earlier. The data stream is off-line in this case, and the READER will pass the data through a "pipeline", from which the LOOKER and / or WRITER can examine all of the data.

When you ask LUCID to Make the software, it determines if the data stream is on-line or off-line, and will only allow you to use one of either the Record or Play buttons.

Looker

Whether on-line or not, the LOOKER will build up histograms and perform other tests and calculations when new data arrives, possibly calling external subroutines that you've written beforehand. You can interact with the LOOKER via the `lucid` program, which allows you to view any or all data, and change the LOOKER's data tests dynamically. New data, such as histograms, can be included in the data stream for the WRITER to save.

Writer

The WRITER records the data after everything else has finished. Data can be converted to an alternate format and saved to a disk file, magnetic tape, or passed as input to another computer pro-

gram, or any combination of places at the same time. For instance, data can be written to a tape in LUCID format.

When data is saved by the WRITER using its default format, the names of all LUCID variables and data types are also saved.

In other words, the data which you save with LUCID is completely self contained; LUCID can rewrite your experimental software based on nothing more than the saved data.

Reader

The Record button in the Control window tells the READER to acquire data, and make it available to the rest of the data stream. This is only true when the WRITER uses its own data format; it is *not* possible with Q's data format, for example.

A Conceptual View of LUCID

A READER must *always* be present for LUCID to do an experiment or analyze data, but the LOOKER and WRITER need not exist. The *source* of data coming into the READER solely determines whether it's "on-line" or not. If it's getting data from a tape, that data must have been written by a previous LUCID data stream. There is no limit as to the number of data streams through which the data may eventually flow, but only *one* data stream is allowed to exist at one time while an experiment is running on-line.

A more comprehensive picture of LUCID is shown in the following figure. The pieces that are *required* to exist have been highlighted.

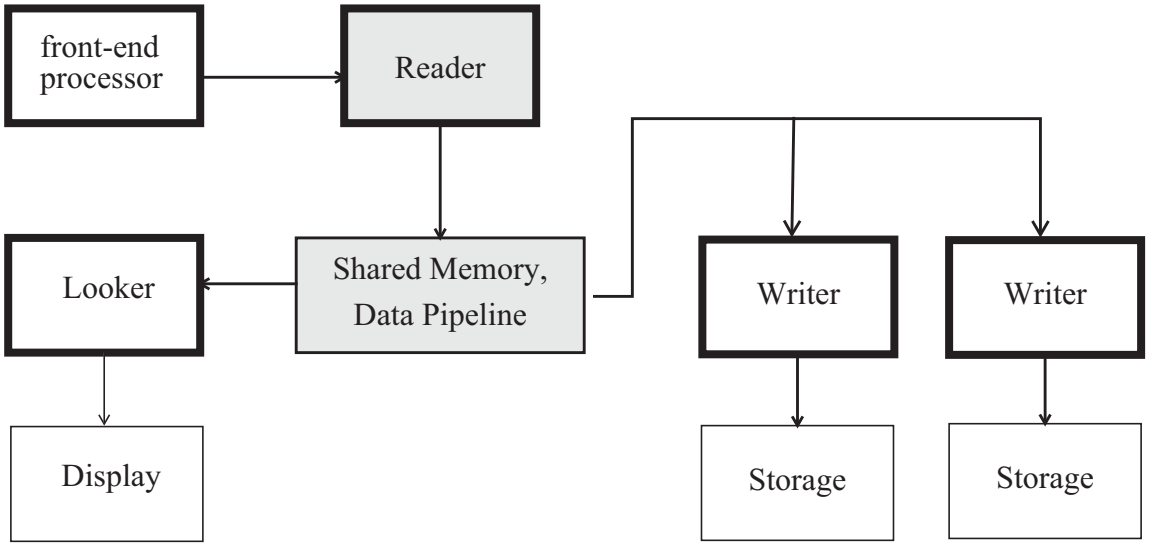


Figure 6.1. Conceptual Data Flow in LUCID

The READER may be used on its own when you're testing electronics. The LOOKER may be added when you're ready to calibrate your detectors or the electronics, and need to see what the data looks like. You can incorporate the WRITER into the data stream when you're ready to record the data. If you're running an experiment with a high count rate, you might also want to leave the LOOKER out of the data stream, after checking the initial setup. It can be easily restarted between runs.

Data Communications in LUCID

The inter-process communication in LUCID can involve up to three separate computers[1] and at least five different processes for an on-line acquisition. Figure 6.2. on page 6-4 shows a typical on-line setup of processes and communications.

1. The Front-end computer is considered a single computer, although it may contain multiple processors.

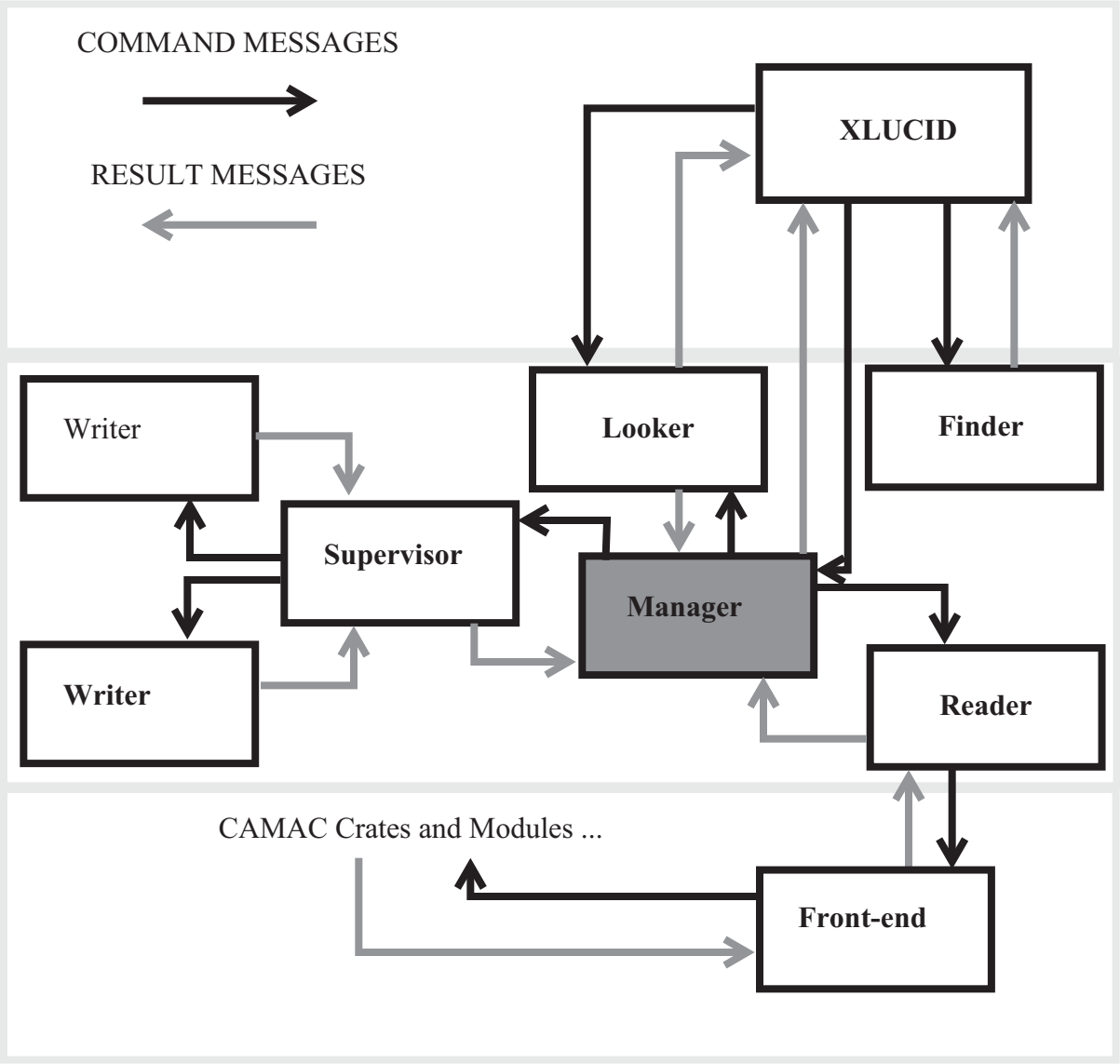


Figure 6.2. Processes and Inter-Process Communication

The LUCID Experiment Directory

For each experiment run on each computer, LUCID establishes a unique directory named *~lucid/experiments/experiment-name*. Within this directory, LUCID places copies of the *prefix.r* and *prefix.l* files. In addition, a number of other files exist. We'll look at the list of files assuming that the experiment prefix *prefix* was used when building the experiment.

Lucid.socket – a special file that allows the display programs to communicate with already running experiments.

prefix.l – the LOOKER program file, which was copied from the user's directory when the experiment was built.

prefix.lsymtab – the looker symbol table, created when the experiment looker was built.

prefix.r the reader program file, which was copied from the user's directory when the experiment reader was built.

prefix.rsymtab– the reader symbol table, created when the experiment reader was built.

prefix.offlooker – the executable version of the off-line analysis looker.

prefix.offlooker.c – the C source generated automatically from the prefix.l file when the user specified off-line analysis.

prefix.offlooker.o – an intermediate file created when the prefix.-offlooker is generated.

logfile – where many of the LUCID messages go. When an obscure message is displayed elsewhere, it's a good idea to look here for more information.

runno – the next run number to be used with data acquisition for this experiment.

seqno – a summary of the number of LUCID records of each record type.

prefix.frontend.c - the C source generated automatically from the *prefix*.r file when the user specified on-line acquisition from an VME front-end processor.

prefix.frontend.h - the C source generated automatically from the *prefix*.r file when the user specified on-line acquisition from a VME front-end processor. This file includes structure definitions for all the LUCID events, declarations for all the READER variables requested by the user, defines for the event numbers, and the maximum event size.

prefix.m167- the executable version of the VME on-line acquisition reader. This program is downloaded to the VME front-end computer.

prefix.m167.o - Intermediate file created when the *prefix*.m167 program is generated.

prefix.c – a copy of the LOOKER C routines written by the user.

prefix.f – a copy of the LOOKER FORTRAN routines written by the user.

*prefix*_C.o – an intermediate object file generated from the prefix.c file.

*prefix*_F.o – an intermediate object file generated from the prefix.f file.

user – a file listing the account names of the people allowed to access the experiment on the current computer. The first name in the list is the *experiment administrator*. If any of the other names is *anybody*, then any user with an account on the acquisition computer can connect to the experiment and control it.

Programming Languages Used

LUCID was written with the C programming language, and it produces C programs to do your experiments. A compiler writing program called YACC was used to understand the description files, and generate the corresponding programs. YACC is also written in C. The LOOKER allows subroutines written in C or FORTRAN to be called while it is running.

Chapter

7

Utility Programs and Subroutines

Utility Programs

There are a number of programs available for working with LUCID data. A few of these programs are system administrator programs, the rest are general use. In the following list, the programs for system administrators are flagged with (system).

addclient (system)

The addclient program sets up the system files on the current computer so LUCID can run. This program can only be run by the super-user. No arguments are required, and there are no harmful effects caused by running the command on a computer that had previously been set up.

copyrun

Copyrun copies runs (surprised?) from a LUCID data source to a destination. Which runs should be copied are determined by the command line arguments. The command synopsis is:

```
copyrun [-o outputfile] [-i inputfile] -s session
run_number ... [ -s session run_number ... ] ...
```

The *inputfile* and the *outputfile* are LUCID data files; either may be a tape drive. The *session* is the session number as shown by lucidview. The *run_number*s listed after a session number are the runs that are to by copied. Runs from multiple sessions can be copied by adding additional session and run number specifications.

Copyrun supports the idea of *remote tape drives*. Although not a regular feature of SunOS, remote tape drives can be accessed by using `hostname:/dev/tape` to access the device `tape` on machine `hostname`, e.g. `vega:/dev/nrst1` accesses the *no-rewind raw scsi tape unit 1* on computer *vega* from any other computer.

demolucid

The demolucid script installs in the current directory the necessary files for running the demo lucid experiment. Once the files are installed, the xlucid program starts with the demo experiment. See Chapter 1 for more information on running the LUCID demo.

extractrun

This program extracts a single run from the specified input and writes it to the specified output. Usage is

```
extractrun [ -o outputfile ] inputfile run_number
```

If *outputfile* is not specified, then the run is written to the standard output. Inputfile must be a valid LUCID data file, and run_number should be a valid run number within that file.

It is recommended that copyrun be used instead, as it allows the same functionality with greater flexibility. For this very reason, extractrun may be removed from the LUCID system in the near future.

findeot

The findeot command positions the tape in the tape drive specified as the only command line argument between the two EOF marks at the end of the data. If the name specified as a no-rewind tape drive, more data can then be added to the tape.

The mt eom command should be equivalent to the findeot command. However, early problems with eom on Exabyte drives attached to Sun workstations brought about the findeot command. Findeot will disappear when there is complete confidence that the majority of the free world has access to a properly functioning mt eom command.

generate

Generate takes as input a LUCID data stream, and produces a C include file from all the LREC_SYMBOL records found. This include file can subsequently be used when writing custom code. For an existing on-line experiment, look for the already created *buildname*.frontend.h in the experiment directory.

hv1440

The hv1440 command communicates with the LeCroy 1440 high voltage mainframe. Please refer to the manual page (*man hv1440*) for complete details.

hv4032

The hv4032 command communicates with the LeCroy 4032 high voltage mainframe. Please refer to the manual page (*man hv4032*) for complete details.

intape

An early program to copy a LUCID data tape to disk. This is now obsolete and selected for removal. Use the copyrun program instead.

lucid

The lucid command is a terminal-based control program for a LUCID experiment. This program pre-dates xlucid, and has not been kept up-to-date. It does allow simple access for monitoring a running experiment. Further discussion of this command can be found in The lucid Command on page 7-8.

lucid_to_q

The lucid_to_q command attempts to translate LUCID data files into Q-format data. This program has not been used to any great extent, and the documentation the program was based on may not be current.

Command syntax is:

```
lucid_to_q [ -ooutputfile ] inputfile ...
```

Where *outputfile* is the destination Q file, and the rest of the arguments are LUCID data files. If the output file is not specified, data is written to the standard output.

There is no current documentation for this command.

lucidlog

The command syntax is:

```
lucidlog [-N] experiment[@machine_name]
```

Lucidlog displays the last N lines of the logfile for the named experiment. If *machine_name* is not specified, the log file on the current computer is used.

lucidman

The lucidman command prints out the LUCID manual pages on systems lacking the MANPATH environment variable (e.g., early ULTRIX systems). It accepts the same arguments as the *man* command on the current machine.

lucidview

Lucidview displays the contents of the named files in human-readable form. Lucidview reads from the standard input if no file arguments are specified. The flag arguments control the amount and style of output produced. By default lucidview displays everything.

The flag arguments, and their effects, are:

-c	Display headers and contents of comment records.
-d	Display event data in data records (implies -e).
-e	Display data record headers and event headers within the data records.
-h	Display LUCID record headers.
-l	Display headers and contents of ‘new LUCID session' records.
-n	At every ‘end-run' record and at the end of each file display the number of each type of event.
-r	Display headers and contents of reader description records.
-s	Display headers and contents of event symbol table records.
-t	Table of contents, showing run numbers and number of megabytes in each run.
-v	Display headers and contents of volume label records.
-wwidth	Sets the maximum width of displayed event data to width characters (default 80).

netcamac

Netcamac sends CAMAC commands to a remote acquisition computer for running on an attached CAMAC highway. The command synopsis is:

```
netcamac [ -d level -D level ] frontend [ branch [
crate [ module [ address [ function [ data ] ] ] ] ] ]
```

The -d and -D flags set the debugging level of the front end host and the netcamac process, respectively. The `frontend` is the name of an acquisition computer. The branch, crate, module, address, and function specify the operation to be performed. Operations which require data input need a data argument.

If any of the arguments are missing, netcamac prompts for input. When all arguments are supplied, the program exits after the command is run. Otherwise, netcamac repeats the prompts to allow running another command.

netcrateinit

Netcrateinit takes an acquisition CPU name, branch, and crate number as arguments. The crate is initialized.

nethv

The nethv program communicates with a LeCroy 2132 across the network. The command takes the form:

```
nethv [ -i ] [ -ccontroller ] [ -pprompt ] [ file ...
]
```

The controller should be specified as bNcNnNtN@host. Host must be a valid acquisition computer name. N is a number representing the branch, crate, station, and type of high-voltage mainframe (either 1440 or 4032). Nethv commands are read from the files named on the command line. The -i option tells nethv to read from the standard input after reading commands from the named files.

help — Produce a description of the rest of the commands.

mainframe N
: Make mainframe N the target of subsequent commands. The default 1440 mainframe address is 1 and the default 4032 mainframe address is 16.

hvon — Turn high voltage on.

hvoff — Turn high voltage off.

set N V

set N-M V Set the demand value for channel N (or N through M) to the value V. The set keyword can be omitted. Channel addresses range from 0 to 255 for 1440 mainframes and from 0 to 31 for 4032 mainframes.

For 1440 mainframes the sign of non-zero values must agree with the polarity of the channel card. For 4032 mainframes the sign of the value is ignored.

Channel ranges are handled differently for 7 kV pods in 4032 mainframes. For these pods, a range affects every other channel. For example, assuming that two 7 kV pods are installed in the first sockets, the command

```
set 0-6 5000
```

will set voltage channels 0, 2, 4 and 6 to 5000 volts. The command

```
set 1-7 300
```

will set the corresponding current limit channels (0c, 2c, 4c, and 6c) to 300 microamps.

display backup N-M

display demand N-M

display measured N-M

Display the backup, demand or measured value of channel N through M (a single channel number N is also acceptable). The display keyword can be omitted. The demand and measured keywords can be abbreviated to their first character. If no channel argument is specified the value (or values) from the previous display command will be used.

ilimit

ilimit value If no value is specified this command displays the positive and negative current limit settings. If a value is specified the command set the appropriate current limit (according to the sign of the value). The current limit values are in units of approximately 10 microamps. This command has no effect on 4032 mainframes.

save filename

Save the demand and current limit values in the specified filename. The name from the previous save command is used if no filename is specified. The default file name is *hv.save*.

quit — Stop processing this file. Forces termination if the end of the file arguments has been reached or if the program is in interactive mode.

qview

Qview is similar to lucidview, but instead works with Q-format data files. Usage is

```
qview [ -cehd ] [ qfile ... ]
```

The flag arguments, and their effects are:

-c Display comments in begin-run and comment records.

-e Display event counters in end-run records.

-h Display event headers in data records.

-d Display event data in data records (implies -h).

If no files are specified on the command line, Q data is expected from the standard input.

readsara

Readsara communicates with the accelerator control system to retrieve and display values of named signals. The common use is to use readsara in a READER startup event or a timed status event to obtain values of accelerator settings and save them. Any number of signal names can be given on the command line, and readsara prints out values in the same order.

Listing the approximately 2000 different sara signals is beyond the scope of this manual.

vme_console

The vme_console command is used to connect to the console port of the acquisition computer. This is useful for debugging purposes, and for resetting the acquisition computer. Usage is simply:

```
vme_console machine_name
```

where *machine_name* is the name of the acquisition computer. If a connection cannot be made from the current machine, vme_console gives a diagnostic message with the name of the computer to connect from.

The *lucid* Command

The xlucid command only allows an experiment to be controlled from a workstation running X11. There are occasions during experiment setup and extended running when a workstation is not at hand. The lucid command provides simple access when only a terminal available.

Options available to the lucid program are:

-d N Set the debugging level to N. The default (no debugging) is 10, the maximum debug level is 0.

-x The current experiment is *off-line*. Default is *on-line.*

-i FILE The current experiment is *off-line,* with input coming from FILE.

-m FILE The file containing module descriptions (as used when building the front-end reader) is set to FILE.

-s TYPE@NAME
The acquisition computer to be used is set to NAME, and it is treated as a TYPE computer. Valid types are *m147* and *m167*. (The -s stands for “starburst”, when it was believed that was the only front-end this program would ever deal with.)

```
lucid -s m167@hubble demo_glen@virgo
```

starts up (connects to) the acquisition experiment demo_glen on the machine virgo, using the front-end computer `hubble`.The menu options shown in Figure 7-1. on page 7-8 are for an *on-line* acquisition. The commands change slightly for an off-line acquisition. Familiarity with using xlucid for controlling an experiment makes many of the command here straightforward.

```
MANAGER: glen@skatter is now in charge of the experiment.

MANAGER: Using directory /users3/wright/Lucid/xlucid

 1. Build/Run experiment   8. Show user-defined commands  15. -<Offline Only>
 2. Start run              9. Start viewer program        16. -<Offline Only>
 3. Stop run              10. Enter comment record        17. -<Offline Only>
 4. Suspend run           11. Create user list            18. -<Offline Only>
 5. Resume run            12. Set debug level             19. -<Offline Only>
 6. Change buffer size    13. Terminate                 20. Modify writer list
 7. Show status            14. -<Offline Only>

Command:
Experiment name: demo_glen@virgo.
- No READER currently active.
- No LOOKER currently active.
- No WRITER currently active.
```

Figure 7-1. LUCID Menu

1. Builds an experiment, prompting for build name (to find source files), and whether to build a reader and / or a looker.
2. (ON-LINE) After an experiment has been built, this starts the acquisition.
3. (ON-LINE) This stops the acquisition for the current run.
4. (ON-LINE) The current run is paused.
5. (ON-LINE) A paused current run is resumed.
6. (ON-LINE) The buffer size used in communicating with the front-end processor is changed. It should not be necessary to use this, as a save data statement in a user-triggered reader command will flush the current buffer no matter the contents.
7. Prints the current status of the READER, LOOKER, and WRITER.
8. Prints the list of user-defined commands in the reader and looker.
9. Starts the *Sunview*-oriented viewer. Because of conflict between this and xlucid, this feature should be ignored. Do not use.
10. Creates a comment record and embeds it in the acquisition stream. At the time of writing, this is the only way to create a comment record.
11. Modifies the user list of people who can access the experiment. This does not handle the anybody user that matches all users.
12. Controls setting the debug level for the various parts of the LUCID system.
13. Shuts down the experiment and exits the lucid program.
14. (OFF-LINE) Opens an alternate data file for input.
15. (OFF-LINE) Starts the playback of the current data file.
16. (OFF-LINE) Stops the playback of the current data file.
17. (OFF-LINE) Rewinds the input data file.
18. (OFF-LINE) Skips forward to a specified run number.
19. (OFF-LINE) Closes the input data file.

20. Allows setting up writer devices and event lists for writer devices. A separate menu is shown for this option which uses

```
Command: 20
r/g/s/p/d/a/e/?: ?

r - read event name list
g - get destination names
s - save destination names
p - print destination info
P - print event names
d - delete destination
a - add destination
e - edit destination info
```

Figure 7-2.Writer Destination Menu

a character menu rather than a numeric menu.

^C Quits the lucid program without shutting down the experiment. This can be entered at any time.

LUCID Functions and Subroutines

Accessing Networked Directories

The *awd()* and *dwa()* routines simplify the mapping of directory paths on different computers. awd(pathname) fills in pathname with the name of the computer and the path on that computer required to get to the current working directory. dwa(rempath, pathname) takes a remote path specification (as returned by awd()) and changes it into the proper path name for the current machine to reach the original path.

Awd stands for Absolute Working Directory, and dwa stands for Directory from Working Absolute. The fact that one name is the reverse of the other is entirely coincidence.

Reading LUCID Data Files

On occasion, it may become necessary to write data manipulation software that reads a LUCID data file. The following routines simplify this.

Lucidopen(file) Opens the named file for reading. An error is returned if the file name is not valid, but no checking is done to ensure that the file contains valid data.

Lucidfdopen(fd) Initializes for LUCID I/O on the argument file descriptor. No checking is done to ensure that the file descriptor references a LUCID data file, or that the file is positioned at a proper location.

Lucidread() Returns a pointer to the next LUCID record. On an error, a record of type LUCIDerr is returned. On end-of-file, a NULL pointer is returned.

Lucidskipdata() Returns the next non-data LUCID record. Error and EOF conditions are the same as above.

Lucidclose() Closes the LUCID data stream. If the data comes from a pipe which was initiated by Lucidfdopen(), Lucidclose() must be called before pclose().

Writing LUCID Data Records

This group of routines provide a low-level method of writing LUCID data files.

Lucidwopen(file) The named file is opened for output. If the file already exists, output will be appended to the end of the file.

Lucidwrite(data, size) The LUCID record referred to by data, containing size bytes, is added to the current LUCID data block. If this fills the data buffer, the buffer is written.

Lucidpad() The current buffer is filled with an LREC_NOTHING record, and written out.

Lucidweot() Two EOF marks are written to the output tape device, and the write head is then positioned between these marks.

Lucidwclose() The output file is closed.

Unfortunately, the LF_ routines (below) do not make use of the Lucidwrite routines. Until this changes, `Lucidwrite()` and `LF_event()` cannot be combined in the same program for producing a single output data stream.

Generating LUCID Data Files

The LF_ (Lucid Format) set of subroutines also write out LUCID data files, but provide a higher level of access to the user. Unlike Lucidwrite(), which expects all the proper LUCID records in the correct order, here records can be built up an element at a time.

LF_rsymtab(filename) The filename can either be NULL, indicating that there is no rsymtab file and the user may define the events manually; or the name of a LUCID data file containing a valid symbol table. This symbol table gets copied to the output data stream.

LF_define(eventno, eventname, varlist, count)

LF_define() creates a symbol table entry for the event named eventname, numbered eventno. Varlist is a pointer to an array of count Lucidvar variables that will be entered into the event description record.

LF_open(filename, experiment, description)

filename is opened for writing LUCID events. The output starts with a NEWSESSION record and a LUCIDFORMAT record. If an rsymtab file was given to the LF_rsymtab filename, then the symbol table is copied from that file to the output file. If LF_rsymtab was not called, then the experiment character string is used to find a computer and experiment directory with an rsymtab file for buildname description. An example of looking for a description file is:

```
LF_open( "testdata", "demo_glen@virgo", "demo");
```

which opens the file `testdata` for output, and looks in the directory `/home/lucid/experiments/demo_glen` on the computer `virgo` for the file `demo.rsymtab`.

The description name and experiment name are used as part of the start session and start run entries. Run numbering starts at 1.

LF_event(eventno, buffer, nbytes)

An event record is written out for type eventno, and is copied from buffer and is expected to be nbytes in size. Nbytes is the unusual size of *number of bytes rounded up to LUCIDALIGN boundary - 2*. The two bytes subtracted at the end are filled in with the event number. The easiest way to calculate this number for a non-compressed event is to define the structure for the event, either using the generate program (page 7-2) or by including the on-line reader include file, and use the C operator sizeof to get the size of the event, then subtract two.

LF_close() The file opened for writing by LF_open is closed.

Low Level CAMAC Access

These routines allow camac operations to be remotely executed on a vme-based cpu. To call these routines it is necessary to include *-llucid* on the compile command line.

NetCamacOpen(), NetCamacPing(), NetCamacSetDebuglev(), NetCamac()

NetCamacOpen (char *name)To perform a camac operation first open a communications channel (a socket) using *NetCamacOpen*. The argument to *NetCamacOpen* is the host name of the target cpu (e.g., palomar or hubble). Only one connection can be operational at any given time. Opening a second connection will cause the first to be closed.

NetCamacPing() can be used to determine if the connection is still active. It is called by *NetCamacOpen* when the connection is established.

NetCamac (int b, int c, int n, int a, int f, long *data)Camac operations are performed by *NetCamac* where:

- b is the branch (an integer between 0 and 7),
- c is the crate (an integer between 1 and 1),
- n is the station (an integer between 1 and 23),
- a is the address (an integer between 0 and 15),
- f is the function (an integer between 0 and 31),
- data is the address of a long integer, which is either the source or destination of the data for the operation.

NetCamacSetDebugLev(int level) sets the debugging level for CAMAC operations on the front-end computer.

DIAGNOSTICS: On error all routines send a message to stderr and return -1. If successful *NetCamacOpen* and *NetCamacPing* return 0, *NetCamac* returns the status bits X (bit 0) and Q (bit 1).

High Voltage Access

The *NetHv* routines provide network access to the 4032 and 1440 high voltage controllers. These routines are a standard part of the LUCID library.

NetHvOpen(char *name)*name* is a string specifying the high voltage controller to connect to. This string is of the format “bBRANCHcCRATEnSLOTtTYPE@MACHINE”, where MACHINE is the front-end computer, BRANCH is the CAMAC branch to use, CRATE is the crate in the branch, SLOT is the module address, and TYPE is the type of module (either 4032 or 1440) that is in that slot. a *daemon* process is started on the front-end computer, and it is this process that the NetHv routines communicates with. 0 is returned on success, -1 on error.

NetHvFrame(int frame)Sets the frame entry for subsequent *NetHv* calls. *frame* must be between 1 and 16, inclusive.

NetHvOnoff(int onoff)If *onoff* is 1, the power supply is turned on. A value of 0 turns the power off. Any other value results in an error return.

NetHvSetRampRate(int ramprate)Sets the ramp rate for the power supply. *ramprate* must be positive.

NetHvSetDemand(int first, int nchan, int value)
Sets the *demand* to the specified *value* for the channels starting at *first* for the number of channels *nchan*.

NetHvGetDemand(int first, int nchan, int *value)
Finds the *demand* for the specified range of channels and places the values in the array *value*.

NetHvGetMeasured(int first, int nchan, int *value)
Reads the voltage levels for the specified range of channels, and places the result in the array pointed to by *value*.

NetHvSetIlimit(int ilimit)Sets the current limit to *ilimit*. If the limit is negative, the negative limit is set, otherwise the positive limit is set.

NetHvGetIlimit(int *pos, int *neg)
Finds the current limits and places them in *pos* and *neg*.

NetHvClose() Removes the connection to the high voltage controller.

NetHvSetDebuglev(int level)Sets the debugging level for the net daemon software.

Reading Q-format Data

The Qread routines provide access to events in files written by the Q software package. A C definition for the Q structures used is found in */home/lucid/include/qformat.h*. This software has not been extensively tested.

Qopen(char *name) The file referenced by *name* is opened to be read by the Qread() routine. 0 is returned on success, -1 on error.

Qread() The next Q record is read and returned as a pointer to the *Qtape-header* structure. This pointer can be cast to point to the appropriate structure type. *NULL* is returned on error or end-of-file.

Qclose() The Q data stream is closed.

Writing Q-format Data

These routines were originally intended to allow Q data to be written directly from the writer. Because the Q information is a subset of what LUCID records, experimenters must now record their data in LUCID format, and then use the lucid_to_q program to convert their data. This allows complete access to the experiment data for independent analysis by different groups.

Qwopen(char *name) The file referenced by *name* is opened for writing of Q-format data.

Qwrite(void * lucidp) The lucid record pointed to by *lucidp* is converted to the proper type of Q record and then written. 0 is returned on success, -1 is returned on error.

Qwclose() The output file is closed.

LUCID Data File Format

This section describes the format of LUCID data. Data definitions are shown using C syntax, and can be found in the LUCID include file `lucidformat.h`.

LUCID Data Record Format Declarations

Output is always fixed length blocks (LUCIDBLKSIZE). Within these blocks are one or more *records* or parts of *records*. Routines are provided to read and return the records to the calling routine. The calling routine never sees the underlying block structure.

```
#define LUCIDBLKSIZE                    (32*1024)
```

Alignment restrictions for specified types: must be powers of 2 since (LUCIDALIGN_XXX - 1) is used as a bit mask.

```
#define LUCIDALIGN_RECORD               4
#define LUCIDALIGN_SHORT                2
#define LUCIDALIGN_LONG                 4
#define LUCIDALIGN_FLOAT                4
#define LUCIDALIGN_DOUBLE               8
```

Number of bytes in variable names, user names, etc. All names are null-terminated, so the maximum number of characters in a name is LUCIDNAMESIZE - 1.

```
#define LUCIDNAMESIZE                   32
```

This header is contained within every record structure given below. The *magic* number ALWAYS comes first in EVERY record, and will make the byte ordering of the writing machine apparent to the reader.

```
struct Lucidheader {
      u_long lh_magic;  identifies type of record
      u_long lh_time;   UNIX style time stamp
      u_long lh_nbytes; number of bytes AFTER header
      u_long lh_sequence;seq. number of this record
                        type
};
```

The following is a list of values for the 'lh_magic' field in the header structure. This identifies the type of record that follows:

LREC_FORMAT declares writing machine's binary representation.

LREC_SYMBOL defines layout and variables used in an event's data record. See *struct Lucidsym* below. This record will contain Lucidvar structures to describe individual variables.

LREC_READER written when user changes experiment's reader description file. Essentially contains copy of the file itself. See *struct Luciddescribe* below. Padded with newlines to a 0 mod LUCIDALIGN_RECORD boundary.

LREC_LOOKER written when the Looker description file changes, similar to LREC_READER, above. See *struct Luciddescribe* below. Padded with newlines to a 0 mod LUCIDALIGN_RECORD boundary.

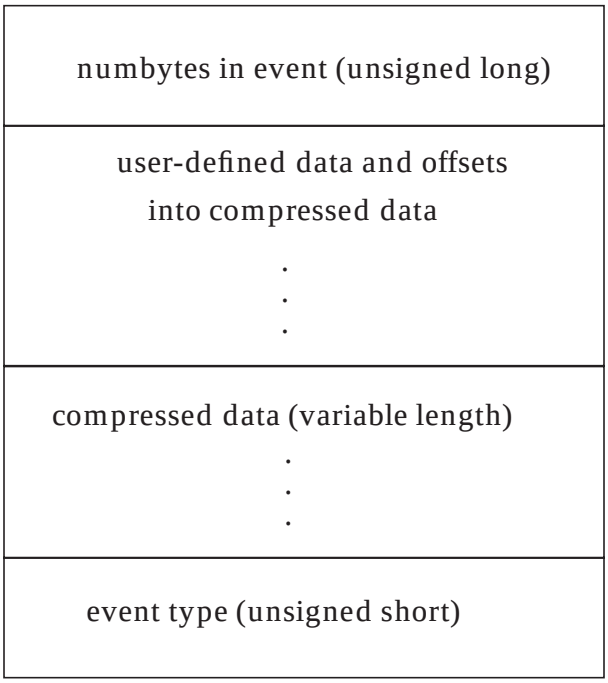
LREC_WRITER written when the Writer description file changes, similar to LREC_READER, above. See *struct Luciddescribe* below. Padded with newlines to a 0 mod LUCIDALIGN_RECORD boundary.

LREC_NEWVOLUME
marks the transition from one file to another (typically a change of tapes in a stacker). See *struct Lucidvolume* below.

LREC_NEWSESSION
written at the beginning of each session. See *struct Lucidsession* below.

LREC_BEGINRUN written at the beginning of each distinct run, whenever the *begin-run* trigger is processed. See *struct Lucidbegin* below.

LREC_DATA Contains zero or more event records, as defined by the user's reader file definition of each event. Each event has a similar overall structure:



The number of bytes in the event does not include itself, only the remainder of the event structure. For a given event type this size will differ from one record to another only if data compression is used. See *struct Luciddata* below.

LREC_ENDRUN written when an *endrun* trigger is processed. It is useful to include this record for time information, rather than depending on the following *beginrun* record, especially for the last run of the experiment, which won't have one! See *struct Lucidend* below.

LREC_COMMENT Written whenever a privileged user wants to write a comment to the data stream. Usually included by default at the start of an experiment, and at the beginning of every run. See *struct Lucidcomment* below. Padded with newlines to a 0 mod LUCIDALIGN_RECORD boundary.

LREC_NOTHING Used for padding. Specifically, when an experiment ends or the output block must be flushed for some other reason, the unused remainder of the block is filled with null bytes contained in this record. See *struct Lucidnull* below.

LREC_ERROR — Used to indicate error reading data. Only the lh_magic structure element is valid. All other structure entries are zero.

```
#define LREC_FORMAT     0x553001AAL
#define LREC_SYMBOL     0x553002AAL
#define LREC_READER     0x553003AAL
#define LREC_LOOKER     0x553004AAL
#define LREC_WRITER     0x553005AAL
#define LREC_NEWVOLUME  0x553006AAL
#define LREC_NEWSESSION 0x553007AAL
#define LREC_BEGINRUN   0x553008AAL
#define LREC_DATA       0x553009AAL
#define LREC_ENDRUN     0x55300AAAL
#define LREC_COMMENT    0x55300BAAL
#define LREC_NOTHING    0x55300CAAL
#define LREC_ERROR      0x55300DAAL
```

Event source values:

```
#define LSRC_UNKNOWN    0
#define LSRC_READER     1
#define LSRC_LOOKER     2
```

The following structure is written near the beginning of a volume to allow software to determine the architecture of the writing machine. It includes a constant in single and double precision format.

```
#define LUCID_FLOATCHECK (1.25)
struct Lucidformat {
      struct Lucidheader lf_head; standard record
                                  header
      double lf_double; double precision format of
                                  source
      float lf_float;   floating format of source
};
```

Event definitions consist of the following Lucidsym structure which will contain zero or more variable definitions. The order of the definitions dictates their order within the data record. Zero variables indicates no data saved.

Different types of variables that can be saved.

```
#define LVAR_UNKNOWN    0
#define LVAR_BOOL       1
#define LVAR_CHAR       2
#define LVAR_SHORT      3
#define LVAR_LONG       4
#define LVAR_FLOAT      5
#define LVAR_DOUBLE     6
#define LVAR_TYPEMASK   077   mask for basic type
```

```
#define LVAR_PADNAME    "<<pad>>"
                        name for all pad bytes
#define LVAR_HISTFLAG   0100
                        histogram bit or'd in
#define LHIST_MAXDIMS   2
      maximum histogram dimensions

#define MAXHISTBINS     4096
      maximum 1 dimensional hist bins
#define MAXSCATBINS     256
      maximum 2 dimensional hist bins
```

Data compression variations

```
#define LCMP_SHORTSHORT 1
      Short channel:short data
#define LCMP_LONGSHORT  2
      Long channel:short data:short pad
#define LCMP_LONGLONG   3
      Long channel:long/float data
#define LCMP_LONGDOUBLE 4
      Long channel:long pad:double data
#define LCMP_HISTOGRAM  5
      Short X channel:short Y channel:long/float
data
#define LCMP_DOUBLEHISTOGRAM     6
      Short X channel:short Y channel:long
pad:double data
#define LCMP_VARIABLE   7
      New-style compression
```

A variable stored using LCMP_VARIABLE data compression appears in the data stream as a variable-length and fixed-length component. The variable-length components appears after all fixed-length data. Indices into the variable-length data are stored among the fixed-length variables. The offsets are the offsets (from the beginning of the event) of the first and just-past-the-last byte of the associated variable.

For example:

```
event e1:
read and save                   v1
read compress and save          v2
read and save                   v3
read compress and save          v4
read and save                   v5
```

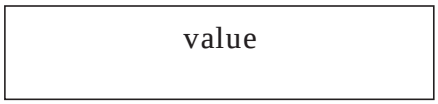
Would produce the data record

```
event size in bytes (padded to modulus 4)
v1
offsets for v2
v3
offsets for v4
v5
variable-length v2 data
variable-length v4 data
event number
```

There may be padding between any of the entries to enforce LUCID_ALIGN alignment.

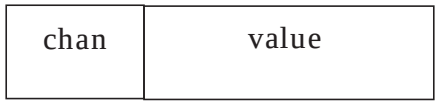
Information on the form of the compressed data shares space with the histogram info in the Lucid symbol table structure. The information includes the bit offsets and lengths of the ‘channel' and ‘value' components of the compressed data. Bit offsets are little-endian after the variable has been converted to native format. A bit offset plus a bit length can not exceed the number of bits in the underlying data type.

The following examples show the format in which compressed data are stored, the lv_type of the symbol table entry, and the compression information:



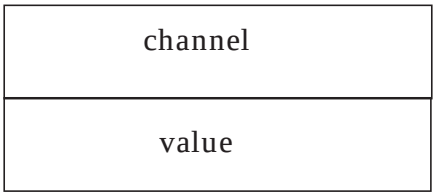
LVAR_SHORT
channel bit offset 0
channel bit length 0
value bit offset 0
value bit length 16

For a module such as a waveform digitizer, or completely user-defined data:



LVAR_SHORT
channel bit offset 12
channel bit length 4
value bit offset 0
value bit length 12

This is similar to the old-style LCMP_SHORTSHORT compression:

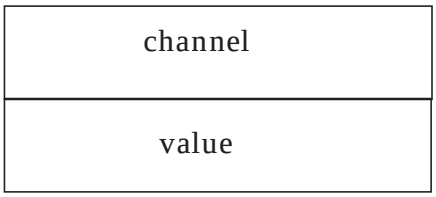


LVAR_SHORT
channel bit offset 0
channel bit length 16
value bit offset 16
value bit length 16

The data storage requirements of this long-style compression is the same as the previous example, but the extracted value is treated as a long when doing data calculations:

LVAR_LONG
channel bit offset 16
channel bit length 16
value bit offset 0
value bit length 16

This is similar to the old-style LCMP_LONGLONG compression:



LVAR_LONG
channel bit offset 0
channel bit length 32
value bit offset 32
value bit length 32

This example shows an extra bit, which is treated as part of the value:

LVAR_LONG
channel bit offset 17
channel bit length 15
value bit offset 0
value bit length 17

```
struct Lucidcompressoffset {
      long lco_begin;   Offset of first byte
      long lco_end;     Offset of just past
                        last byte
};
```

Extra information stored in symbol table for compressed variables.

```
struct Lucidcompressinfo {
      u_char lci_channel_offset;
                        Bit offset of channel
      u_char lci_channel_length;
                        Bit length of channel
      u_char lci_value_offset;
                        Bit offset of value
      u_char lci_value_length;
                        Bit length of value
};
```

Extra information stored in symbol table for histogram variables.

```
struct Lucidhistinfo {
      u_short lh_nbins[LHIST_MAXDIMS];
               number of bins per dimension
      float lh_lolimit[LHIST_MAXDIMS];
               lower limit values per dimension
      float lh_hilimit[LHIST_MAXDIMS];
               upper limit values per dimension
};
```

Symbol table entry for variables.

```
struct Lucidvar {
      char lv_name[LUCIDNAMESIZE];name of variable
      u_long lv_asize;  array size
      u_char lv_type;   type of variable
      u_char lv_compress;non zero if data compressed
      u_char lv_strlen; size of string variable
      u_char lv_ndims;  histogram dimensions used
      union {
        struct Lucidhistinfo lv_lh;
                        Histogram information
        struct Lucidcompressinfo lv_lci;
                        LCMP_VARIABLE information
      } lv_un;
# define lv_nbins       lv_un.lv_lh.lh_nbins
# define lv_lolimit     lv_un.lv_lh.lh_lolimit
# define lv_hilimit     lv_un.lv_lh.lh_hilimit
```

Symbol table entry for events.

```
struct Lucidsym {
      struct Lucidheader ls_head;
               standard record header
      char ls_name[LUCIDNAMESIZE];name of this event
      u_short ls_eventtype;event I.D. code
      u_short ls_numvars;
               Number of ls_var to follow
      u_char ls_source;
               Event source(reader/looker)
      u_char ls_internal;
               Internal flags for Lucidread
      u_char ls_pad[2]; Pad to 0 mod 4
      struct Lucidvar ls_var[1];
               variables saved in this event
};
```

Definition of Reader / Looker / Writer description record

```
struct Luciddescribe {
      struct Lucidheader ld_head;
                standard record header
      char ld_name[LUCIDNAMESIZE];
                 user who installed this change
      char ld_text[4];  text of description file
};
```

Definition of new volume (i.e. tape) record. The overall volume number is recorded in the header.

```
struct Lucidvolume {
      struct Lucidheader lv_head;
                  standard record header
      u_char lv_version;
                  LUCID version number (i.e. X.11)
      u_char lv_release;
                  LUCID release number (i.e. 1.XX)
      char lv_pad[2];   Pad to 0 mod 4
      char lv_volname[LUCIDNAMESIZE]; volume name
};
```

Definition of new session record

```
struct Lucidsession {
      struct Lucidheader ls_head;
                        standard record header
      char ls_expname[LUCIDNAMESIZE];
                        name of experiment
      char ls_dscname[LUCIDNAMESIZE];
                        description file base name
      char ls_name[LUCIDNAMESIZE];
                         name of person beginning session
};
```

Definition of new run record

```
struct Lucidbegin {
      struct Lucidheader lb_head;
                        standard record header
      u_long lb_runno;  run number
      char lb_name[LUCIDNAMESIZE];
                        name of person beginning run
};
```

```
Definition of actual event record
struct Luciddata {
      struct Lucidheader ld_head;
                  standard record header
      unsigned long ld_data[1];
       The remainder of this record
       type is dynamic, and depends
       upon the user's specification as
       found in the symbol table record
};
```

Definition of comment record.

```
struct Lucidcomment {
      struct Lucidheader lc_head;
                  standard record header
      char lc_name[LUCIDNAMESIZE]; name of commenter
      char lc_text[4];  body of comment
};
```

Definition of end of run record.

```
struct Lucidend {
      struct Lucidheader le_head;
                        standard record header
      char le_name[LUCIDNAMESIZE];
                        name of person ending run
};
```

Definition of null (padding) record.

```
struct Lucidnull {
      struct Lucidheader ln_head;
                        standard record header
};
```

Appendix

A

The CAMAC Module Database

LUCID's knowledge of CAMAC modules comes from information stored in a system database file, which doesn't change very often. The file is named "~lucid/lib/camac.modules", and although anyone can look through the file, only the system administrator is allowed to change it. Chapter 2 is entitled Using LUCID, and it describes how you can get LUCID to use your own module definition file, which is nice when you want to test a new module, or use an uncommon feature of an existing module. Most users will ask the system administrator to add a definition to the system database, since there is no practical limit on the number of modules that LUCID can know about, and other experimenters will eventually want to use that module.

The module definition file contains simple descriptions of CAMAC modules, which tell LUCID how to access the modules. This includes commands for simple reading and writing, but also how to enable and disable LAMs, and how to read and clear modules in a single operation. LUCID "silently" reads this CAMAC database every time you start it up, so you normally never need to know of the file's existence.

Defining the Modules

Definitions are listed one after another in the database file. Each definition starts with a line containing the word module, followed by one or more names for that module. For example, the definition for a Kinetic Systems model 3615 scaler starts off like this:

```
module "Kinetic Systems 3615 6-Input 100 MHz Scaler"
         "scaler3615"
         "ks3615"
         "3615"
```

You can have several "aliases" for a CAMAC module, but each of them must be surrounded by double quotes. Recall that when you specify a type of CAMAC variable in your READER description file, the type of variable must match one of the names given in the database. For instance, our first example under The DEFINE Section on page 3-4 was:

```
define doubles "3615" crate 1 slot 10 addr 0-2
```

LUCID recognizes the module "3615" because it appears as one of the aliases in the CAMAC database file.

The first name for a module should be a long, descriptive one; That's the name LUCID will display if it thinks you're using a module incorrectly in your READER description file. The other names should be written in decreasing order of length, as shown in the first example. There is no reasonable limit on the number of aliases you can have for a module.

As with all LUCID files, the lines in the CAMAC database are free format. The recommended format for the CAMAC database is that each module line start at the left hand margin, and that all subsequent lines for that definition are indented by one TAB character. This makes it easy to read the definition and find module names quickly, and it also allows you to print a list of all the modules very easily, using the "grep" command in UNIX. Comments are encouraged inside the database file, and are introduced by an octothorpe (#) character, except when it is inside a character string. All characters from that point to the end of that line are ignored, as in all other LUCID files. You should also leave at least one blank line between subsequent CAMAC module definitions, just to denote the end of one definition and the start of another.

After all the aliases for a module are given in the file, all of the appropriate CAMAC capabilities should be listed.

CAMAC Capabilities

There are twelve different CAMAC capabilities that LUCID understands, each of which are selected by a unique keyword. These keywords can appear in any order after the name aliases.

Readsize and Writesize

The readsize keyword tells LUCID the maximum number of data transfers that this module can accept at one time. In the example of our 6-input scaler, the statement would be:

```
readsize 6
```

Similarly, the word writesize tells LUCID how much data can be written to the module at one time. These read and write sizes are used to check and possibly limit the amount of data transferred to or from a module in a single operation[1]. With respect to defining a CAMAC variable in a READER description file, this size determines the array size of the variable. All modules should have a readsize or writesize defined. An option for "hold mode" transfers can be specified with these keywords; this will be discussed under HOLD Mode Transfers on page A-4.

Shortword and Longword

Most CAMAC modules transfer 16-bit values, or less. However, LUCID must be able to communicate with 24-bit modules as well, such as high density ECL coincidence registers. The keywords shortword and longword tell LUCID whether the module will transfer 16 or 24 bits, respectively. If neither of these keywords is present, LUCID assumes the module will transfer 16 bits. Only one of these keywords may be included in one module definition.

Checking Module Identity

Some CAMAC modules have an internal status value which might contain a serial number or I.D. number. If a module's definition includes a check statement, then LUCID's program will check that value before it starts the experiment. This is useful when setting up an experiment, because LUCID can help remind you when modules have been moved around. The general format of this statement is:

```
check is f1 a1: data & 0140 = 0100
```

This tells LUCID to use F(1) A(1) to read the status value. It also tells LUCID that within that piece of data, the sixth and seventh rightmost bits must be zero and one, respectively. Octal (base eight) values are most commonly used here, since they represent bit positions. Decimal values can be used, but should be avoided to prevent confusion. In the this example, the value 0140 represents 001100000 in binary. The value 0100 represents the binary value of 001000000. The first value tells LUCID which bits are of interest, and the second value tells what state those corresponding bits should be in. Thus, the sixth rightmost bit must be a zero, and the seventh rightmost bit must be a one.

1. For more information, refer to the last part of the Section entitled Reading Data on page 3-14.

If this test is performed and the result isn't what you've told LUCID to expect, then an error message will be displayed when you try and start the experiment.

Read

All other capabilities in the database file are associated with CAMAC function codes and sub-address values. The read keyword tells LUCID what function code to use when reading the module, and what sub-address codes (A-codes) are acceptable. When the user tells LUCID to read this CAMAC variable, it will repeatedly use the function code with each of the A-codes to do it. If a subscript is given for that variable, then only that A-code will be used. In any case, the statement must have the form:

```
read is f0 a0-5
```

This tells LUCID to use F(0) with a sub-address code A(0) through A(5). Also, the keyword "is" is optional. If the module only understands one A-code, then the definition would be what one would expect:

```
read is f0 a0
```

LUCID will warn you if it finds a function code for reading which is not between 0 and 7. It also complains if the A-codes are not between 0 and 15.

HOLD Mode Transfers

Certain types of CAMAC modules allow reading or writing several channels without changing the A-code every time. They keep track internally of what has been accessed, and automatically proceed to the next "channel" after every transfer. These are called *hold mode transfers*, because the computer holds the same A-code on every transfer to or from the module. In some cases, hold mode transfers can be performed slightly faster than ordinary transfers. More importantly, CAMAC only allows sixteen different A-codes to be used, and some modules need to transfer more data than can be done in sixteen transactions.

With hold mode transfers, a large amount of data can be transferred in a single read statement from LUCID. For instance, a waveform digitizer can transfer an array of 1024 bytes with a single read statement from the READER. To tell LUCID that a module uses hold mode reads or writes, you must include the word hold after the readsize or writesize statements in the database. For instance, an appropriate statement for the previously mentioned waveform digitizer would be:

```
readsize 1024 hold
```

Hold mode can therefore be used when reading or writing.

Write

The write keyword tells LUCID what function code to use when writing to this module. The statement is essentially the same as the read statement, except that LUCID will complain if the function code is not between 16 and 23. The same rules for hold mode transfers also apply. For example:

```
write is f17 a0-9
```

Read and Clear

When the user wants to read and clear a module with one operation, LUCID uses the CAMAC operation specified with the readclear keyword. Normally, the function code is F(2), but LUCID warns you if it is not between 0 and 7. The hold mode transfer for reading applies to this operation.

```
readclear is f2 a0-15
```

Clear

A simple clear operation may be specified by the user, which is different from the "clear" in a read and clear operation. In this case, the entire module is cleared (reset) to it's initial state. Normally, the command is F(9).

```
clear is f9 a0
```

Enabling and Disabling LAMs

LUCID will try to enable LAMs in those modules which use LAMs for triggers. It will also try to disable LAMs from all other modules. To specify the "enable LAM" command, the statement is:

```
enable is f26 a0
```

LUCID will complain if the function code is not between 24 and 31. Similarly, to disable LAMs from a module, the statement looks like:

```
disable is f24 a0
```

Testing for LAM Status

If the user tells LUCID to wait for a module in his READER description file, then LUCID simply waits for a LAM status bit to become true, usually because that particular module will not cause an interrupt when a LAM is ready. We often think of LAMs as being the same as interrupts, except a LAM is essentially something that can be tested. An interrupt is a trigger, which is often caused by a LAM.

The test LAM function is recognized by most modules, and the statement should look like:

```
testlam is f9 a0
```

LUCID's READER program will repeatedly issue the command until the "Q" response from the module is true, meaning that a LAM state exists. This is used only for the wait command from the user.

Clearing LAMs

When LUCID has processed an interrupt (LAM) trigger, it will try and clear the interrupt state, so it can accept a new one. Modules usually respond to an F(10) command to clear the LAM state. Keep in mind that this is different from the read and clear operation, and from the clear operation; It simply resets the interrupt that has just arrived from the module. The command is:

```
clearlam is f10 a0-5
```

Notice that individual channels may interrupt in some modules; Specifying several A-codes will cause LUCID to clear all of them.

Self-Clearing Modules

Certain modules will reset their interrupt state automatically, after the last channel is read out. For instance, the LeCroy model 2259 Peak Sensing ADC has twelve inputs, and will issue a LAM when it has finished conversion and is ready to be accessed. When the last input has been successfully read out, the module

clears it's own LAM; it essentially sends itself and F(10) command. To make use of this feature, you may use the autoclear statement within the CAMAC module database file. This tells LUCID not to clear the module if the user causes the last channel to be read out.

Q-Stop Modules

Some modules make a variable amount of data available at a time, and indicate the end of the data transfer by a change in the module's Q state. The keyword qstop indicates this. The event data saved from a Q-stop module is automatically compressed, as described in the following section.

Self-compressing Modules

Modules, such as Q-stop modules, that provide a variable amount of data each event are considered *self-compressing* modules. The CAMAC Module Database provides a good deal of flexibility in defining the format of the data for the two most common cases: a simple list of values, and a list of channel-value pairs. The statement used for both these cases is

```
channel length CL offset CO value length VL offset VO
```

The lengths and offsets of each field are specified as bit positions counting from the least significant bit. The offset bits are counted starting with the least significant bit, which has an offset of 0. A length of 0 indicates the field isn't used, so a module with a `shortword` word size and a compression definition of

```
channel length 0 value length 16 offset 0
```

would indicate no channel data and a single shortword for the data value.

The READER does not actually use the compression information provided! Rather, this information is made available to the LOOKER to decode the compression.

A more complete discussion of different compression formats can be found in the section LUCID Data File Format on page 7-15.

Sample CAMAC Module Database

The following definitions originate from a CAMAC module database on an active LUCID system. The modules are in no specific order, but are presented to show how the statements should appear in the file. You should consult the user's manuals for each of these modules to see how the CAMAC commands are represented here. Remember that modules may be duplicated if they have different names, or slightly different functions. If LUCID finds duplicate module names, it will use the first name in the file.

```
module "Accelerator lab homemade 16-channel ECL D-
Flip/Flop Coincidence Module"
         "SAL Dffcoin"
         "SAL Dff"
         shortword
         read is f0 a0

module "Accelerator lab homemade 8-channel ECL Delay
Module"
         "SAL Ecldelay"
         readsize 8
         shortword
         write is f16 a0-7
         read is f0 a0-7

module "Accelerator lab homemade super tagger unit"
         "SAL supertagger 2"
         "SAL stu"
         shortword
         read is f0 a0
         clear is f10 a0

module "Accelerator lab homemade tagger trigger
module"
         "SAL supertagger"
         "SAL stm"
         shortword
         read is f0 a0
         clear is f9 a0

module "Accelerator lab homemade output register"
         "SAL outreg"
         "SAL oreg"
         shortword
         read is f0 a0
         write is f16 a0
         clear is f9 a0

module "Interface Standards OD/48 48-bit output
register"
```

```
        "od/48"
        readsize 2
        writesize 2
        longword
        read f0 a0-1
        write f16 a0-1
        test is f8 a0
        clearlam is f10 a0
        disable is f24 a0
        enable is f26 a0
        check is f1 a1: data & 017600 = 02000

module "Jorway model 41 NIM output register"
        "J41"
        "Jorway 41"
        shortword
        write f16 a0

module "Kinetic Systems 3615 6-input 100 MHz Scaler"
        "3615"
        "ks3615"
        "ksc3615"
        "Kinetic Systems 3615"
        readsize 6
        longword
        read is f0 a0-5
        clear is f9 a0-5
        readclear is f2 a0-5
        enable is f26 a0
        disable is f24 a0
        testlam is f8 a0-5
        clearlam is f10 a0-5

module "Kinetic Systems 3924 LAM Encoder for Serial
Highway"
        "3924"
        "ks3924"
        "ksc3924"
        "Kinetic Systems 3924"
        readsize 1
        read is f1 a12
        enable is f26 a1
        disable is f24 a1
        testlam is f8 a0
        clearlam is f10 a0
        clear is f10 a0

module "LeCroy Model 2228A/2229 Octal TDC"
        "2228A"
        "2229"
        "LeCroy 2228A"
        "LeCroy 2229"
```

```
        timeout 100
        readsize 8
        shortword
        autoclear
        read is f0 a0-7
        readclear is f2 a0-7
        test is f8 a0
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

module "LeCroy 2248 8-Channel Differential ADC"
        "2248"
        "LeCroy 2248"
        timeout 106
        autoclear
        readsize 8
        shortword
        read is f0 a0-7
        readclear f2 a0-7
        enable f26 a0
        disable f24 a0
        testlam f8 a0
        clearlam f10 a0

module "LeCroy Model 2249A/SG 12 channel ADC"
        "2249A"
        "2249SG"
        "LeCroy 2249A"
        "LeCroy 2249SG"
        timeout 60
        autoclear
        readsize 12
        shortword
        read is f0 a0-11
        readclear is f2 a0-11
        test is f8 a0-11
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

module "LeCroy Model 2249W/2259B 12 channel ADC"
        "2249W"
        "2259B"
        "LeCroy 2249W"
        "LeCroy 2259B"
        timeout 106
        autoclear
        readsize 12
        shortword
        read is f0 a0-11
        readclear is f2 a0-11
```

```
        test is f8 a0-11
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

module "LeCroy Model 2256A 20 MHz Waveform
Digitizers"
        "2256A"
        "LeCroy 2256A"
        readsize 128 hold
        shortword
        read is f2 a0
        test is f8 a0
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

# NOTE: the 2277 can be used to read out multiple
values for a single
# channel. This is not directly usable in the looker.
Instead, the
# user must decode this manually.
#       To use automatic decoding, change the
'readsize' line to::
#       readsize 512 qstop channel length 5 offset
17 value length 17 offset 0
#
module "LeCroy Model 2277 32 Channel TDC"
        "2277"
        "LeCroy 2277"
        readsize 512 qstop  channel length 0 value
length 22 offset 0
        longword
        read is f0 a0
        test is f8 a0
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

module "LeCroy 2323A Dual Gate and Delay Generator"
        "2323A"
        "LeCroy 2323A"
        readsize 2
        shortword
        read is f1 a0-1
        clear is f9 a0-1
        write is f17 a0-1

module "LcCroy Model 2341S 16-fold coincidence
register/pattern unit"
        "2341S"
        "LeCroy 2341S"
        readsize 1
```

```
        shortword
        read f0 a0
        readclear f2 a0
        clear f9 a0-2

#
# The LeCroy 2373 MLU must be placed in INHIBIT mode
and the CAR register
# must be written before the memory pattern can be
written or read.
# e.g.
#       camac mlu A(2) F(16) data 0x50
#       camac mlu A(1) F(16) data 0
#       load mlu from "/tmp/foobar.mlu"
#       camac mlu A(2) F(16) data 0x40
#
module "LeCroy 2373 Memory Lookup Unit"
        "LeCroy 2373"
        "LRS 2373"
        "2373"
        readsize 65536 hold
        writesize 65536 hold
        shortword
        read is f0 a0
        write is f16 a0

module "LeCroy Model 2551 12-channel 24-bit scaler"
        "2551"
        "LeCroy 2551"
        readsize 12
        longword
        autoclear
        read is f0 a0-11
        readclear is f2 a0-11
        test is f8 a0
        clear is f9 a0
        disable is f24 a0
        enable is f26 a0

module "LeCroy 3511/3512 Single Channel Spectroscopy
ADC"
        "3511"
        "3512"
        "LeCroy 3511"
        "LeCroy 3512"
        autoclear
        readsize 1
        shortword
        read is f0 a0
        readclear f2 a0
        write is f16 a0
        enable f26 a0
        disable f24 a0
```

```
        testlam f8 a0
        clearlam f10 a0

module "LeCroy Model 4299 databus interface"
               "4299"
               "LeCroy 4299"
        shortword
        writesize 1
        readsize 100 qstop channel length 0 value
length 16 offset 0
        read is f0 a1
        readclear is f0 a0
        write is f16 a0
        clear is f9 a0
        enable is f26 a0
        disable is f24 a0
        testlam is f8 a0
        clearlam is f10 a0

module "LeCroy Model 4300 16 channel fast encoding
and readout ADC (FERA)"
        "4300"
        "LeCroy 4300"
        timeout 9
        readsize 16
        shortword
        read is f2 a0-15
        write is f16 a0
        test is f8 a0
        clear is f9 a0

module "LeCroy model 4413 16 channel 150 MHz updating
discriminator"
        "4413"
        "LeCroy 4413"
        readsize 1
        shortword
        read f1 a0
        write f17 a0
        enable f26 a0
        disable f24 a0

module "LeCroy model 4418 16 channel programmable
logic delay/fan-out"
        "4418"
        "LeCroy 4418"
        shortword
        write is f16 a0-15

module "LeCroy Model 4434 32-Channel 24-bit scaler"
        "4434"
        "LeCroy 4434"
        readsize 32 hold
```

```
        longword
        read is f2 a0
        test is f8 a0
        write is f16 a0

module "LeCroy Model 4448 48-bit coincidence
register/pattern unit"
        "4448"
        "LeCroy 4448"
        readsize 3
        shortword
        read f0 a0-2
        readclear f2 a0-2
        clear f9 a0-2
        test f8 a0-2

module "LeCroy Model 4508 Dual 8-input/8-output
programmable logic unit"
        "4508"
        "LeCroy 4508"
        readsize 4 shortword
        writesize 256 hold shortword
        read is f0 a0-3
        readclear is f2 a0-3
        clear is f9 a0-1
        write is f16 a0-3

module "24-bit word generator switch register"
        "wgr200"
        longword
        read f0 a0

module "LeCroy Model 2415 High-voltage power supply"
        "2415"
        "LeCroy 2415"
        shortword

#
# Block of 8 TDC modules associated with 4 SAL D-
flip-flop modules
# See tagger template for an example of how this
`module' is used.
#
module "Block of 8 LeCroy Model 2228A/2229 Octal TDC"
        "2228A Block"
        "2229 Block"
        "LeCroy 2228A Block"
        "LeCroy 2229 Block"
        timeout 100
        shortword
        readsize 64 qstop channel length 16 offset
0 value length 16 offset 16
        autoclear
```

Appendix

B

Example Description Files

The example shown here is the reader and looker description file used with the “demodata” data file discussed in Chapter 1.

```
#
# acquire data from Le Croy 2249A ADC
#

define demo_adc  "2249A" C(1) N(12) A(0-11)
define scaler "3615" N(11) A(5)
define total integer*4
define gates integer*4

trigger pulse every demo_adc lam
trigger quit  every 40 seconds
trigger count every 30 seconds
trigger start every beginrun

event pulse:
         read clear and save demo_adc
         save demo_adc
         gates = gates + 1
         save gates

event count:
         read and clear scaler
         save scaler
         total = total + scaler
         save total

event quit:
         read and clear scaler
         save scaler
         total = total + scaler
         save total
         endrun

event start:
         total = 0
         gates = 0
```

Figure 7-1. “demodata” READER File

```
define spectrum[12] histogram from 0 to 2048
define increases, number,gate int*4

beginrun:
         spectrum = 0
         increases = 0
         number = 0
         gate = 0
event pulse (some):
         increment spectrum using demo_adc
         gate = gates
event count: increases = scaler
         number = total
```

Figure 7-2. “Demodata” LOOKER File

Appendix

C

Known Problems

Like many other things today, LUCID is good, but not perfect. The first concern of any user of any software package is, "*What* doesn't work quite as documented?" The second concern is, "*Will* you add this feature?" The following sections try and address these concerns by documenting known problems as completely as possible, and by listing the requested enhancements that are most likely to added.

Known Bugs

- The C unary operator "!" is not supported as a conditional operator.
- Use of XLUCID properties causes difficulties when moving from one experiment to the next.
- The luciddraw utility program, when used at the same time as the XLUCID program, can lock up the experiment.
- XLUCID will allow you to change directory for working with different reader/looker files, but does not change the directory for the LUCID program. Changes made to the program files are not seen until XLUCID is restarted in the new directory.
- Shutting down an experiment completely from XLUCID does not always kill the *manager* process.
- The XLUCID message window can overflow.

Problems When Running Off-Line.

- Starting an existing off-line experiment on a new machine does not automatically find and copy the list of users who can access the experiment.
- When reading multiple run number ranges, pressing stop then rewind can hang XLUCID.

Problems When Running On-Line

- Stopping LUCID from a paused state does not give the user the option of *Just Stop* or *Resume then Stop*.
- SunOS does not always add an EOT when a tape device is closed. LUCID on-line data tapes will report an error after reading all the data, although the data and the generated histograms are fine.

Requested Enhancements

As you read this section, remember that these are only the *requested* enhancements that are most likely to be implemented. Although many of these changes will be done, and some are already being scheduled, there is no guarantee that all will be added.

Changes to READER

- Add Loops statements for the *experiment reader* file.

Changes to WRITER

- No known requests outstanding.

Changes to LOOKER

- Allow on-line data acquisition to continue while a looker is being rebuilt.
- Off-line lookers could optionally close the data stream at the end of the last requested run.
- increment statements should have the option to increment by a value other than 1 or -1.
- Processing of *some* or *all* events should be switchable outside the looker description file.
- The *print* statement should allow some type of formatting option.
- Add two-dimensional arrays.
- Allow “non-fatal” read errors: processing would continue at the beginning of the next run.

Changes to XLUCID

- Add saving of histogram data directly to a user program.
- Add log scale display on axes.
- Add *"bytes seen this run"* information when forwarding over data.
- Add a routine callable by user subroutines that accesses data in a lucid data structure.
- Add backspacing of LUCID tapes to the beginning of a session.
- Add other information onto histograms for printouts (e.g. date, run number).
- Complete the implementation of the *Density Plot* window.
- Add the ability to modify histogram data that can be returned to the looker.
- Implement the *Print* menu selection.
- Implement the *Layout* menu selection.
- Add simple histogram editing resulting in local variables.
- Add more statistical functions for histograms (*e.g.* first and second moments).

Appendix

D

Histogram Save File Format

The original format used by LUCID for saved histograms was extremely simple. It was assumed that histograms saved by the LOOKER would be re-read by the LOOKER, and histograms saved by XLUCID would be re-read by XLUCID. The new format makes the histogram files consistent, and provides the user with better information about the histogram saved.

Overview

The histogram file format constists of a header and data body for each histogram, allowing multiple histograms to be saved in the same file.

Header

The histogram header consists of one or more lines of free-format information. The different header fields consist of a keyword, possibly followed by a list of parameters. The number of parameters is fixed for each keyword.

Variable	Has a single parameter, which is the name of the variable that was saved.
saved	Followed by two parameters, which are the date and time the histogram was saved.
run	The single parameter is the run number when the variable was saved.
compress	No parameters; this indicates that fields that are zero have not been saved to the file.
array	This indicates that the histogram is an array, and has the lower and upper bounds of the array as its two parameters.
entry	*entry* indicates that this is just one element from a histogram array, the single parameter tells which element.

Histogram	Verifies that the variable saved was a histogram.
type	The variable type saved; the single parameter is *short, long, float, double* or *string.*
X	Indicates that the following *from, to,* and *bins* keywords apply to the X axis.
Y	Indicates that the following *from, to,* and *bins* keywords apply to tye Y axis.
from	A single parameter, which is the lower limit of values saved to this histogram.
to	A single parameter, which is the upper limit of values saved to this histogram.
bins	A single parameter, which is the number of bins for the current axis.
byvalue	No parameter; *byvalue* indicates that data is saved as an X or Y value representative of the bin information being written out.
midpoint	No parameters; this indicates that the X and Y values written out are at the midpoint of the bin. The default is that values are printed as of the lowest edge of the bin.
bybin	No parameter; opposite of *byvalue.*
binary	No parameter; indicates data is in binary instead of ascii. This allows smaller data files, but is not always portable to another computer and is diffucult to manipulate with anything but a dedicated program.
ascii	No parameter; opposite of binary.

Body

The body of the histogram file varies greatly depending upon the way the histogram was saved. However, the header information is sufficient to allow determining which format the data is in.

Ascii files have the data from one bin on each line. There can be up to four fields in each line, which are:

```
array-index x-bin-number y-bin-number value
```

The array-index only appears if the *array* keyword is in the header. The x-bin-number is present for all histograms, although it could be a bin number, the x value representing the lowest edge of the bin, or the x value at the midpoint of the bin, depending again on keywords (*bybin, byvalue, midpoint*). The y-bin-number

is only present for scatter plots (2-dimensional histograms), and, as the x-bin-number, the keywords *bybin*, *byvalue* and *midpoint* determine if it is a bin number, the y value representing the lowest edge of the bin, or the y value at the midpoint of the bin. The value field is always present, and is the count for the current bin.

Histograms that are saved as *binary* have the same fields as *ascii* saves under the same circumstances, with the major difference that the array index, x bin number, and y bin number are all saved as *float* values (real*4), and the value field is saved according to the type of histogram, which is usually *long* (int*4).

INDEX

Index

Index