DL/CSE/TM41

technical memorandum Daresbury Laboratory

DL/CSE/TM41

DATA ACQUISITION AND CONTROL USING "LabVIEW"

by

P.A. BUKSH and I.W. KIRKMAN, SERC Daresbury Laboratory

APRIL, 1991

Science and Engineering Research Council
DARESBURY LABORATORY
Daresbury, Warrington WA4 4AD

ISSN 0144-5677

Data Acquisition and Control using 'LabVIEW'

P.A. Buksh and I.W. Kirkman

April 16, 1991

SERC Daresbury Laboratory, Warrington, UK

Abstract

This paper describes National Instruments' software package, 'LabVIEW', and discusses its potential usefulness within the laboratory. An introduction to the product and an explanation of the concepts behind it are first given, then its use is illustrated with an example of a specific application by the Soft X-Ray group. Recommendations are made as to the type of applications for which it might more generally be used.

1 Introduction

There are many instances around the site of experimental arrangements that require associated data acquisition and control systems. Examples range from a simple laboratory instrument such as a voltmeter, whose reading is perhaps monitored across an RS232 interface, to beamline monochromators requiring several stepper motor axes to be driven and data channels to be recorded. In the majority of cases such systems are controlled by a microcomputer running compiled software written in a conventional language such as FORTRAN or C. (For instance, around the SRS storage ring the majority of stations are controlled by an LSI-11 computer, running a large FORTRAN program and driving CAMAC.)

The development time for systems using such an approach can be relatively long. This is principally because the programmer has to write code to control virtually every aspect of operation, from setting the value of a DAC to plotting a graph of retrieved data on the terminal screen. Once operational, the software may also be rather inflexible and difficult to modify, particularly by anyone not connected with its original development.

For these and other reasons a new generation of high-level languages have recently appeared (of which LabVIEW is one of the best known) which can dramatically reduce development times and increase subsequent flexibility. They do this by providing a large number of software modules to perform many of the tasks that normally *need* to be coded explicitly. Put simply, they allow the programmer to specify *what* he wants to do rather than exactly *how* he wants to do it.

In mid-1989 the Soft X-Ray (SXR) Group was engaged in the development of a two-axis reflectometer, which contains both active components (stepper motors) and channeltron detectors. The types of experiment planned for the detector were rather varied and required that the physical arrangement of its components be easily changed or added to; an adaptable and flexible control system was therefore necessary. This led to the adoption of the IEEE-488 (GPIB) interfacing standard and a LabVIEW control enviroment; the program subsequently produced is described in some detail in section 3. This solution turned out to be relatively cheap, flexible, easy to interface, and required a fraction of the development time that might have been necessary with a more conventional approach.

Our experience with LabVIEW has led us to believe that whilst it may not necessarily be the ideal solution for every situation, there are likely to be many applications within the laboratory for which its use could certainly be recommended.

2 LabVIEW Overview

It is not possible to provide more than a brief introduction to programming in LabVIEW here; of course, detailed user guides and reference manuals are provided with the product.

LabVIEW runs only on Apple Macintosh platforms at present, though versions for other machines such as PCs and workstations are believed to be under development. It makes extensive use of the Macintosh's graphical user interface, and as a result conventional lines of code are not generally used; instead programs are represented pictorially. To build up a LabVIEW program the user selects options from standard libraries of pictorial entities, which provide all the facilities available with conventional languages (such as FOR loops, WHILE loops, IF statements, input/output, arrays, etc). Perhaps the best way to explain this is by way of a simple example.

The following is a short FORTRAN subroutine which takes the two floating point numbers passed to it, 'X' and 'Y', and returns the number 'Z', where $Z^2 = X^2 + Y^2$:

```
      SUBROUTINE HYPOTENUSE ( X , Y , Z )
      REAL X , Y , Z
C
      Z = ( X**2 + Y**2 )**0.5
C
      RETURN
      END
```

In practice, this routine would be compiled and perhaps kept as an independent object module (or as part of an object library), and could be 'called' by any other program or routine that needed to perform this operation. In a similar fashion a LabVIEW module, known as a *virtual instrument* or *VI*, can be built up to perform the same function. The name virtual instrument arises because LabVIEW characterises any software module as if it were a bench instrument that can be 'wired up' to other bench instruments. Passing data values across 'wires' between VIs is analogous to passing data values between subroutines in FORTRAN.

A LabVIEW version of the code is shown below in figure 1.

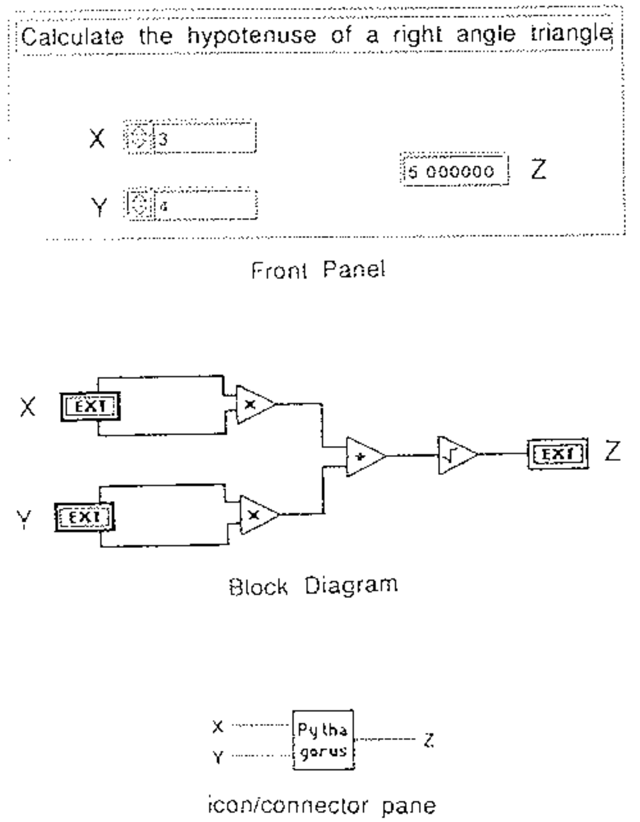


Figure 1: LabVIEW instrument to calculate the hypotenuse of a right angled triangle

The first point to be made is that the program is in two parts: the *front panel* (remember the analogy with a bench instrument) and the *block diagram*. Essentially, the block diagram constitutes the 'code', while the front panel is what the user sees when the program is actually running. Beneath the block diagram is shown the user-created

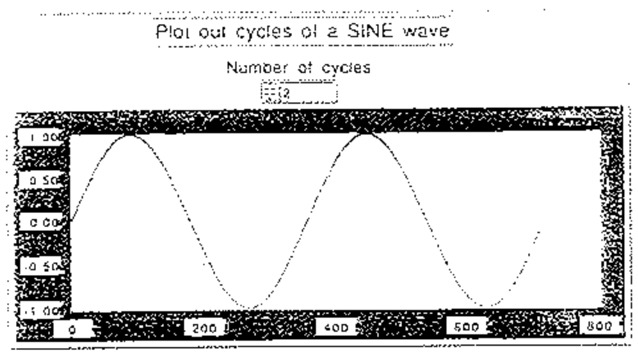
icon representing the module. Associated with the icon is a *connector pane* (not shown here) through which parameters are passed into or out of the instrument. In this example the values 'X' and 'Y' enter through *terminals* on the left of the connector pane, while the calculated answer, 'Z' is passed back through a terminal on the right. The connector pane configuration is selected from a menu of many different available types, depending upon requirements.

On the front panel there are three boxes representing the parameters X,Y and Z. The boxes associated with X and Y are known as *digital controls*, because their value can be passed into the module from outside; whilst that associated with Z is known as a *digital indicator*, because its value is calculated within the module (actually on the block diagram), and depends upon the values of X and Y. When the virtual instrument is created these digital controls and indicators are 'attached' to both the terminals on the connector pane, and to entities on the diagram.

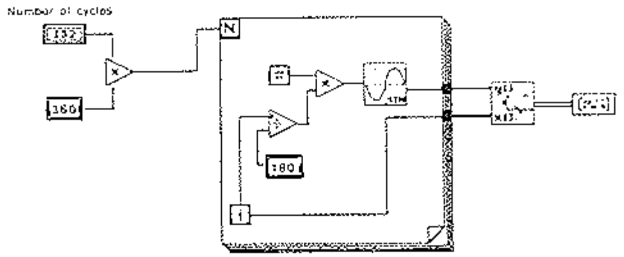
The diagram itself is reasonably self-explanatory. The values X and Y are wired to the arithmetic function icons representing multiplication, addition and square root (which are themselves LabVIEW VIs) with a wiring tool provided (not shown here). The calculated answer is wired to the digital indicator, Z, and would then be both displayed on the front panel and passed back out through the appropriate terminal on the connector pane.

This *sub-instrument* would be stored as a single, independent file containing both the source code (what is shown here) and the compiled code, which is automatically generated when the module is saved. (Note that compiled LabVIEW is claimed to run about as fast as compiled C.) Like all LabVIEW instruments it is capable of being run on its own if required; alternatively, if it is called by another instrument, it would appear on that instrument's diagram represented by its icon (as with the arithmetic operations sub-instruments shown in figure 1). Thus there is really little distinction between sub-instruments written by the user, and those provided as part of LabVIEW itself.

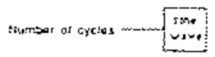
In this particular case the FORTRAN code is probably simpler than the LabVIEW equivalent. But suppose, for example, that the application called for data to be plotted to the screen. With FORTRAN, or most other conventional languages, this would entail linking in some sort of graphics package (such as GHOST or UNIRAS), which would not only provide opportunities for bugs and errors to creep in, but might also make the final code less portable to other sites or machines. In contrast, with LabVIEW the program may be only marginally more difficult to write than the simple example given above. This is because LabVIEW utilities for plotting data in various ways are already provided. To illustrate the point, consider the LabVIEW program shown in figure 2, which plots out a simple SINE wave.



Front Panel



Block Diagram



icon/connector pane

Figure 2: LabVIEW instrument to plot a simple sine wave

Here, an x-y graph has been called up from a standard plotting library and placed on the front panel. A digital control has been created to specify the number of cycles to plot (this is passed in through the connector pane). On the block diagram the LabVIEW representation of a FOR loop is shown, represented as a rectangle with the letter N in one corner. The value wired to N specifies the number of times the loop should run for, and whatever is inside the box will be executed on each cycle. The loop counter, represented as i, will then have values ranging from 0 up to N-1.

In this particular case i specifies an angle in degrees. Each time round the loop its value is first converted into radians, then fed into the SINE function. The values of i and sin(i) are wired to the edge of the loop, where they automatically accumulate into two arrays. When the loop is completed, these arrays pass into a module which converts them to a form suitable for the x-y plotter, the symbol for which is at the extreme right hand side of the block diagram.

This example illustrates an important philosophy of LabVIEW; namely, the fact that it is *data driven*. Entities on a block diagram are only executed when they have all the data that they require. In this case, the loop cannot begin until N has been calculated, and the data cannot be plotted until the loop itself has finished. If sections of code are not dependent upon preceding events, and the order in which they should execute is not specified, then they will run concurrently with equal priority. Thus, LabVIEW code can be made to run in a pseudo-parallel fashion. This can be particularly useful for synchronizing timing between different parts of a program. Placing a time delay upon the block diagram of a VI (using a simple instrument provided) fixes the total time spent in that VI before it finishes executing and returns control to the calling instrument (provided whatever else the VI has to do does not take longer than this delay).

In addition to the x-y graph shown in figure 2, a large number of other controls and indicators are provided for inclusion on the front panel (the x-y graph is really only a sophisticated indicator). These include: strip charts, scope charts, an assortment of dials, meters and knobs, and switches for boolean control. It is also possible to import pictures from packages such as 'Macdraw' to represent, for example, the ON/OFF states of a switch. In this way a front panel can be made to be intuitively easy to use, or even to look very similar to the laboratory instrument being controlled. An example of the former is shown in figure 3.

Shown is the front panel of a VI to control a Caburn dual-stepper motor driver system. The four arrows around the words 'sample stage' are actually latched boolean switches, and 'clicking' on one of them with a mouse when the program is running moves the sample stage in the direction indicated by the desired number of steps.

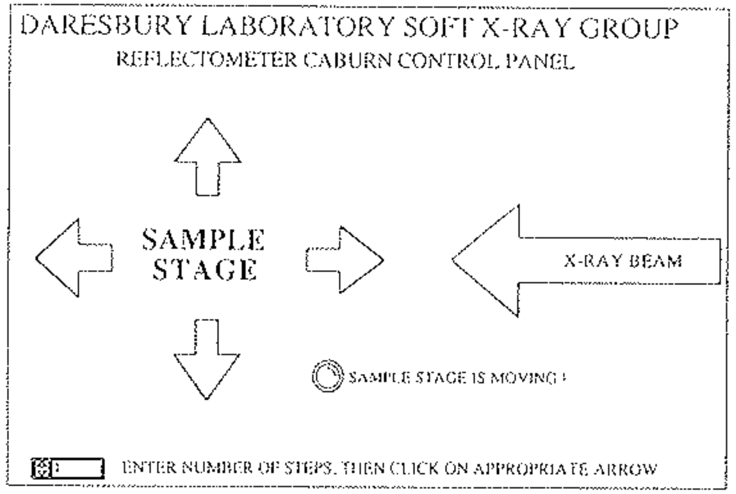

DARESBURY LABORATORY SOFT X-RAY GROUP
REFLECTOMETER CABURN CONTROL PANEL
SAMPLE STAGE
X-RAY BEAM
SAMPLE STAGE IS MOVING !
ENTER NUMBER OF STEPS, THEN CLICK ON APPROPRIATE ARROW

Figure 3: Front panel of a LabVIEW instrument to control a Caburn stepper motor driver system

Of course, in practice a program could consist of a number of VIs, not just a single one. Another feature of LabVIEW is its ability to display a *VI hierarchy*, showing the calling sequence of VIs within any given program. Figure 4 shows the VI hierarchy for the Caburn program described above.

Shown is the icon of each module in the calling structure. Of the twelve instruments, five are from LabVIEW libraries: open GPIB driver, GPIB read, GPIB write, number to integer and fraction, and bleep. It will be left to the reader to deduce which are which.

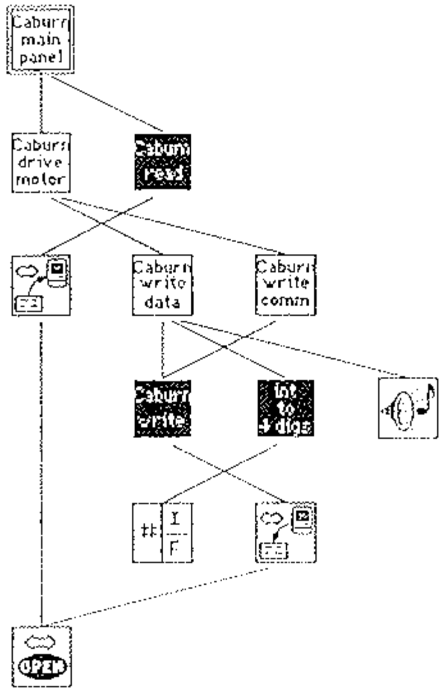


Figure 4: The VI hierarchy for the Caburn program shown in figure 3

A large number of LabVIEW sub-instruments are provided for inclusion on the block diagram, all available from pull-down menus (as are the front panel options). These include the usual arithmetic and boolean functions, instruments to create or access diskfiles in various ways, and instruments to perform digital signal processing, filtering, curve fitting and statistical analysis. The complete library of functions available actually requires over 12 MB of disk space and contains more than 500 VIs.

Another option provides for the user to write a module conventionally if desired, in either C or PASCAL. Known as a *code interface node*, such a module would appear on the block diagram in iconal form, just like any other VI.

Because LabVIEW provides most of the facilities of a conventional language, it is certainly a powerful software tool. However, its real usefulness to experimentalists lies in the fact that it also provides a simple and straightforward way of interfacing to many types of hardware. This can be done either through the serial ports on the back of the Macintosh, or alternatively via GPIB (provided a suitable interface card has been installed in the machine). In both cases appropriate VIs are provided to allow full communication to take place with any attached hardware. As an example, figure 5 shows the front panel and icon/connector pane of a VI that 'writes' a string to a device on the GPIB (its icon was also shown in the VI hierarchy diagram in figure 4).

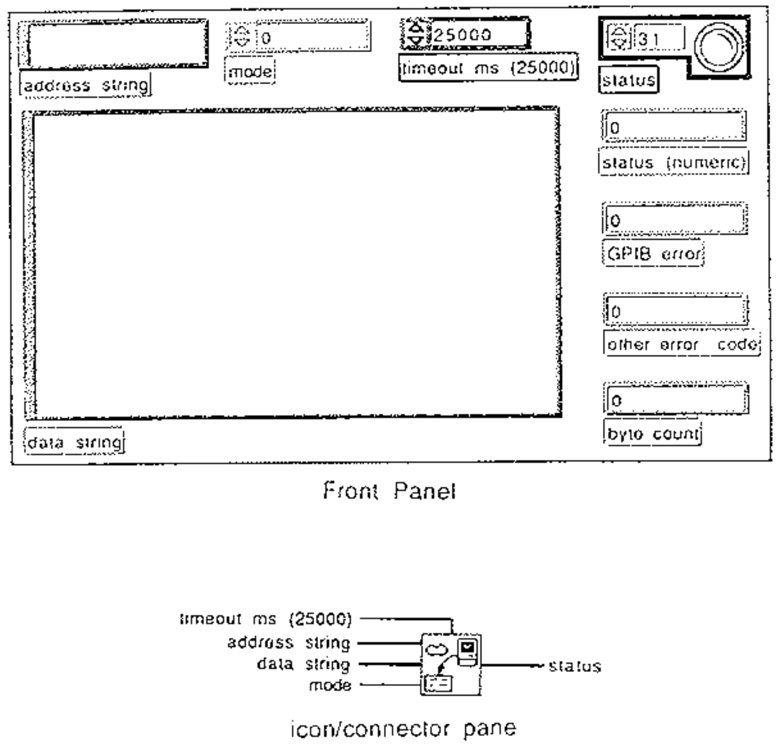


Figure 5: Front panel of a LabVIEW instrument to write a string to a device on an attached GPIB

The GPIB address of the device (in ASCII string form), the data string to be written, the mode (which specifies the string termination character), and the timeout (which will default to 25 seconds if not connected) are all passed into the instrument; returned from it is a status flag indicating whether or not the write has been successful.

In addition to the above, numerous instrument drivers have been produced which are tailored to control particular GPIB devices. A list of those currently available is given in Appendix A.

It is also possible to control instrumentation based upon other types of bus standard, provided that some interface to GPIB is available. For example, to control a CAMAC system from LabVIEW, the standard crate controller must be replaced with a GPIB-CAMAC crate controller (such as a LeCroy 8901A) which essentially turns the whole crate into a single GPIB instrument. Virtual instruments are provided which allow easy communication with individual CAMAC stations and sub addresses within the crate, using the standard 'N', 'A' and 'F' CAMAC notation. A system based on the VXIbus can be controlled in a similar way, though an interesting development by National Instruments is to place a Macintosh SE/30, running Labview, on a commander card inside the VXI crate - thus bypassing GPIB completely and allowing the full capabilities of VXI to be utilised.

Another way of controlling hardware or capturing data is through the use of plug in boards. Here again many manufacturers are providing LabVIEW instrument drivers for their products. A good example of this is National Instruments' own digital signal processing and analysis accelerator board, the NB-DSP2300, which comes complete with a full complement of LabVIEW modules and C development tools, and provides 33.3 Mflops for real-time control and analysis.

3 LabVIEW Implementation of a Control Program for the SXR Reflectometer

3.1 Background and Introduction

In the soft x-ray energy region, focussing, filtering and monochromatisation is performed using either grazing incidence geometry with mirrors and gratings, or non-grazing incidence geometry with very large d-spacing crystals. Both techniques pose difficulties for many experimental demands. In the former case the principle problems are the low collection apertures available at grazing incidence and the aberrations associated with such extreme geometry; in the latter case the naturally occurring crystals (eg mica, beryl) have a d-spacing that does not allow access to energies less than about 600ev at best, and the synthetic crystals (eg KAP, RAP, TlAP) are fragile and have very strong absorption features in this energy region. All these crystals also have very low peak and integrated reflectivities.

In order to overcome some of these difficulties, *multilayer* devices, consisting of deposited layers of alternating high Z and low Z elements, have been under development by a number of laboratories. With these, the d spacing can be tailored to a particular requirement and the elements used can achieve high normal incidence peak reflectivity throughout the soft x-ray energy range.

An EEC-funded collaboration between Daresbury Laboratory, Aberdeen University, and the FOM Institute for Plasma Physics (Nieuwegein) has been developing techniques for the manufacture and measurement of such multilayer devices. It is within this context that the reflectometer chamber has been developed, allowing multilayer diffraction to be measured to a high degree of accuracy.

3.2 Physical Description of the Reflectometer

Figure 6 shows the main elements of the reflectometer chamber and its associated electronic hardware. The reflectometer consists essentially of a large (70 cm) diameter, evacuated, stainless steel chamber, within which is located a platform, known as a *sample stage*, on which any device to be tested can be placed. Collimated x-rays from a source at one side of the chamber (CK_α, NK_α) pass through a thin mylar window and are allowed to strike the sample. The sample stage platform itself can be rotated about a line lying through the central axis of the chamber (axis 1), thus allowing the angle of incidence of the incoming x-rays on the sample to be altered. A channeltron detector is used to monitor the diffracted radiation, and this can be rotated in an arc centred also on the central axis of the chamber (axis 2). In front of the channeltron is a narrow exit slit of size typically 50 microns.

Both the sample stage and the channeltron detector are moved around their respective axes by means of stepper motor-actuated, high-precision rotation stages (microcontrol RT200), driven from a dual channel Maclennan driver unit (the accuracy of rotation being about 0.001 degrees). The sample stage can also be translated around in its own plane, thus varying the point at which the x-ray beam strikes the sample without altering their relative inclination. This is performed using in-vaccuum stepper motors, driven by a Caburn driver unit (using the separate VI shown in figure 3).

The output from the channeltron detector is first amplified and pulse shaped, then discriminated and fed into an Ortec 944 pulse counter.

3.3 Modes of Operation of the Reflectometer

The reflectometer can be operated in one of three ways:

- 'θ_1 only' (stepper motor axis 1) - here, the sample stage is rotated about axis 1, and the detector is kept fixed. This mode is used purely for the purpose of finding the 'straight-through' position, in which the grazing angle of the beam on the sample is zero.
- 'θ_2 only' (stepper motor axis 2) - here, the incident angle is kept fixed and the detector is rotated around to examine the order structure of the diffracted radiation. This mode is used, for example, to measure the efficiency of gratings.
- 'θ_1 and θ_2' (stepper motor axes 1 and 2) - axis 1 is rotated to vary the angle of incidence, while axis 2 is rotated by twice the amount to ensure that the detector sees the zero order of the diffracted radiation. This is the mode used for multilayer work, and from it the spacing of the layers, the number of layers, and some measure of the roughness of the surface between layers can be deduced.

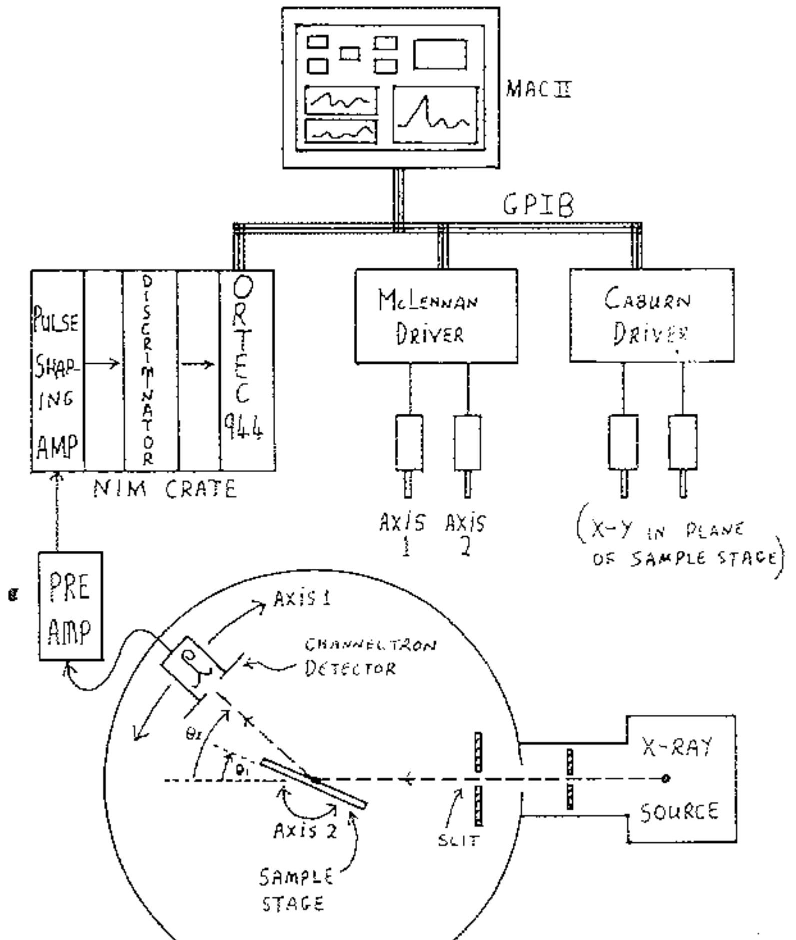


Figure 6: The main components of the reflectometer and its associated control equipment

3.4 Requirements of the Control System Adopted

The Maclennan driver, the Caburn driver, and the Ortec 944 counter all have a GPIB interface, and it was therefore logical that this should be used to control and monitor the equipment. Also, as the reflectometer was intended to be a test and development platform, at least to begin with, it was important that the control system be flexible, and not take too much time to develop (in other words the effort spent in writing the software had to be kept to a minimum). Financial constraints also limited the choice, there seemed little point in spending more on the control computer and associated software than on the experimental hardware itself. LabVIEW was considered to be by far the best solution to fulfil these requirements.

3.5 Description of the LabVIEW Program

The point should first be made that the program described here is only that initially developed for the instrument. Ongoing modifications have improved it considerably, though they have also enlarged it to a size beyond what could reasonably be described in this document.

The main panel of the LabVIEW program to control the reflectometer is shown in figure 7. As with all LabVIEW instruments, this front panel contains a number of controls and indicators of varying types. Consider each in turn:

- *no of increments* - a digital control (ie the user enters a value in the box before running the program) which specifies the number of times the stepper motor(s) should be moved. This also corresponds to the number of data points that will be taken.
- *steps per increment* - a digital control which specifies the number of steps to drive the motor(s) between each successive data point. In 'θ_1 only' and 'θ_2 only' modes (ie stepping axis 1 or axis 2 only) this corresponds to the actual number of steps that will be driven; in 'θ_1 and θ_2' mode, however, axis 2 will move twice the number of steps indicated.
- *delay (ms)* - a digital control which specifies a time delay between stepping the axes and taking data.
- *iteration counter* - a digital indicator which displays the number of increments performed by the program at any moment during a run.
- *axis 1 and axis 2* switches - boolean controls which switch the stepper motors ON and OFF, thus controlling the mode adopted. Note that if both motors are switched off the program will perform a 'time scan' only.
- *counting time* - a digital control which specifies the integration time to be spent counting at each point. The slider is controlled with a 'mouse'.
- *channel A / channel B* switch - a boolean control which determines the channel of the Ortec counter, A or B, to be plotted on the x-y graph at the end of the scan. (Only channel A is used in this application.)
- *type your comments here* - a string control which allows the user to type in a comment to be written at the start of the datafile created.

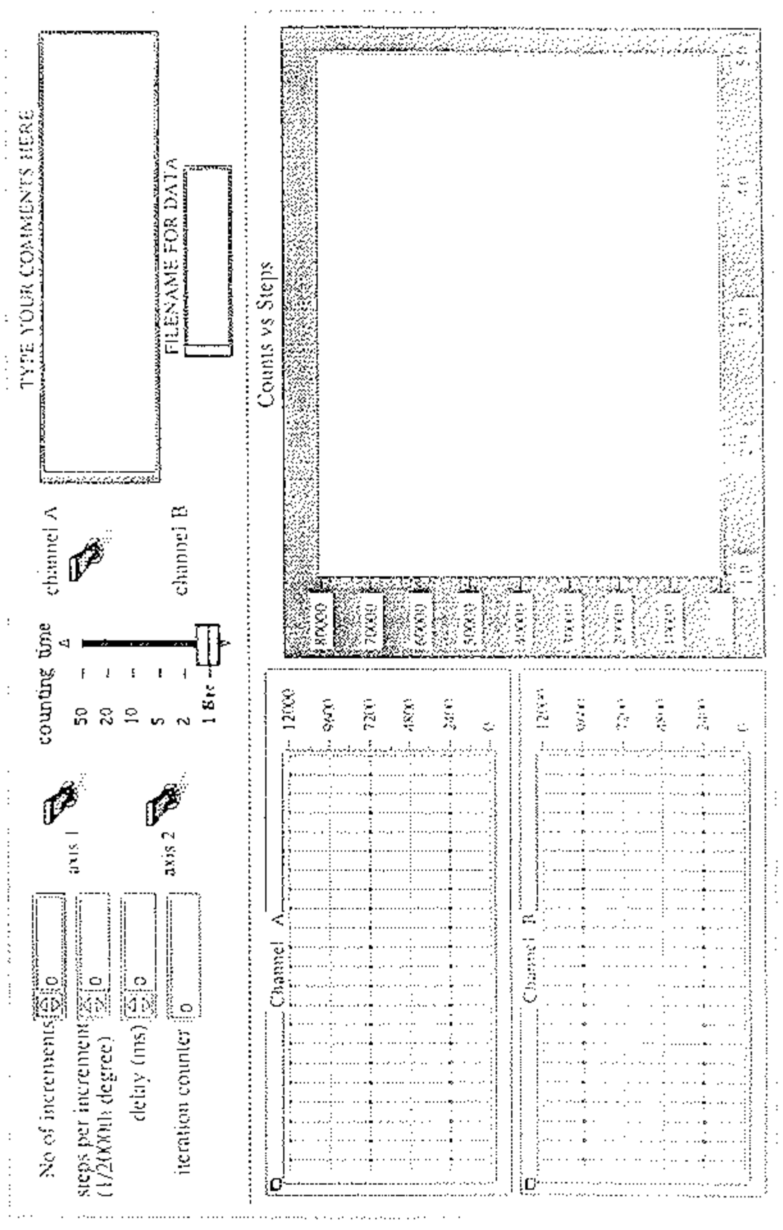


Figure 7: The front panel of the LabVIEW program to control the SXR reflectometer

- *filename for data* - a string indicator showing the filename used to store the data in. Note that this filename is specified automatically within the program, as will be demonstrated.
- *channel A and channel B* strip charts - indicators which plot the data read in real time from each channel of the Ortec counter. Note that the abscissa on these graphs is simply incremented by one between each successive data point.
- *counts vs steps* - an x-y graph indicator which plots the counts recorded for the channel selected against steps moved; updated only when the scan has finished.

Figure 8 shows the VI hierarchy of the program (as explained in section 2), from which it can be seen that a total of 15 sub-VIs are used (ie VIs lying beneath the level of the main panel itself). Of these, 9 are user written and the other 6 are from standard LabVIEW libraries.

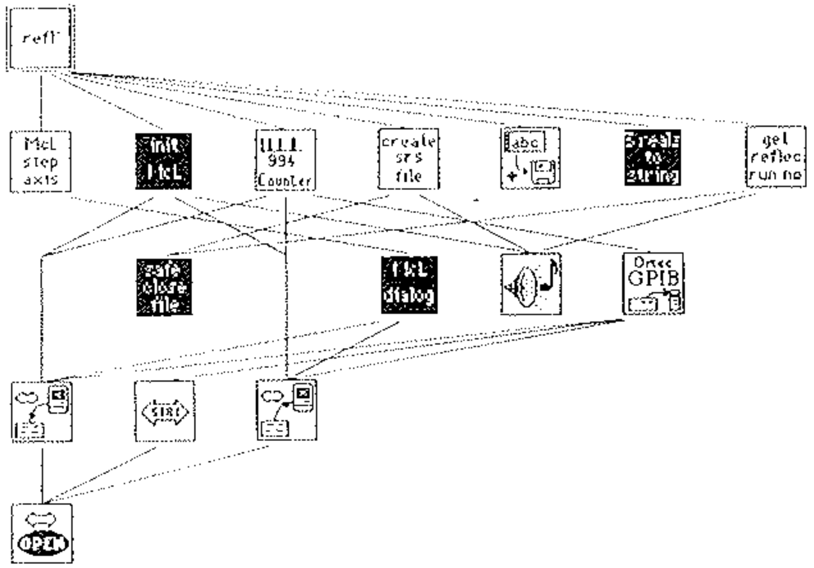


Figure 8: The VI hierarchy of the LabVIEW program to control the SXR reflectometer

Consider first the user-created VIs:

refl	the main program itself.
McL step axis	drives a Maclennan axis a given number of steps.
init McL	initialises the Maclennan driver unit.
LLL 994 Counter	initiates reads from the Ortec counter.
create srs file	creates an 'SRS' type file for the data.
3 reals to string	writes 3 'real' values into a single, formatted string.
get reflec run no	gets the run number to be used in the data filename.
safe close file	closes any file instructed to close, if open.
McL dialog	passes command or data strings across the GPIB to the Maclennan.
Ortec GPIB	communicates with the Ortec counter across the GPIB.

Now the standard LabVIEW VIs:

abc	opens a file, appends a string, then closes the file again.
	'bleeps' when called.
	writes a string to an address on the GPIB.
STAT	GPIB status.

reads a string from an address on the GPIB.

opens GPIB driver.

There is insufficient space here to discuss the workings of each of these VIs independently; rather, the way in which they are used on the block diagram of the control program (shown in figures 9a, 9b and 9c) will be examined. Note though that only the icons in the second row of figure 8 will be seen in figures 9a, 9b and 9c, since those below them are at a lower level of the program's calling structure.

To begin with, note the three rectanglar boxes in figure 9a with the word 'true' written on their top edge. These are *case structures*, and take as their input a boolean value wired into the question mark, '?', on their left hand boundary. Because a boolean value can have two possible states, 'true' or 'false', the case structure itself also has two states. Whatever is inside the 'true' case is activated when the boolean is true, and whatever is inside the 'false' case is activated when the boolean is false. In this example, the 'false' condition for all three case structures is empty, as is shown in figure 9b.

Because LabVIEW is 'data driven', as already discussed, the VI with the icon 'init McL' is the first part of the diagram that will run. This VI returns a boolean value indicating whether or not the initialisation was a success; if it was not, then the boolean will be false and the appropriate state of the outer case structure will be chosen. As this is empty the program will simply terminate. Although not shown here, the VI 'init McL' also puts up an error message to the Macintosh screen if this is the case.

Assuming the Maclennan driver unit is initialised successfully, and the 'true' outer case structure is chosen, the next piece of code to run will be that lying between the outer case structure and the next inner one. This is because the second case structure also needs a boolean value to be wired to it before it can know which case to choose.

The VI represented by the icon 'get reflec run number' opens up a file on a particular directory on the Macintosh's hard disk, in which is stored the last run number used (which is simply an integer). It increments this number, writes it back into the file, and returns the new value through its connector pane (this approach is identical to that adopted on most station computers). If for some reason it cannot perform this task, say because the run number file has been deleted or its directory has been renamed, then the VI returns a zero. The returned run number is tested against zero using a standard LabVIEW function: if it is non-zero, then 'true' is passed to the next case structure; if it is zero then 'false' is passed to it. The run number is also passed into a LabVIEW function that converts it to a character string; this string is then appended to another string constant, 'ref/', making up the filename to be used for the data. This filename string is wired to the filename indicator, so as to display it on the front panel, and also into the next level case structure.

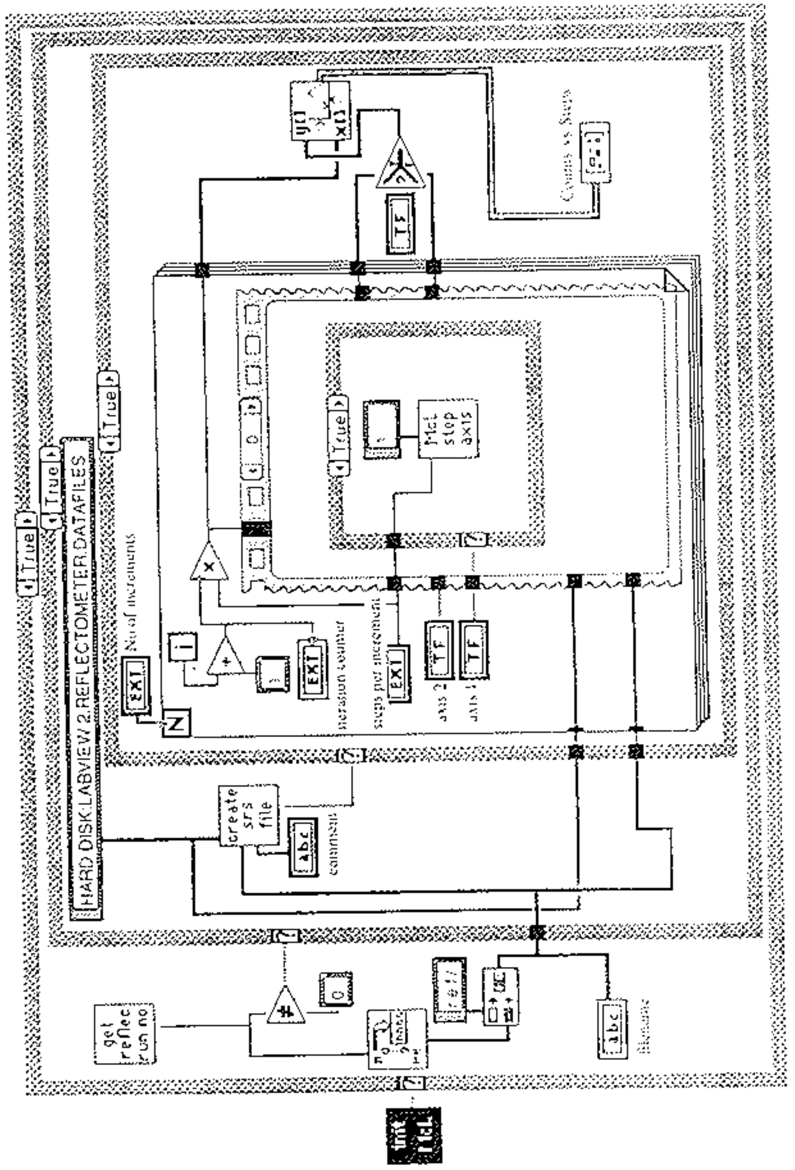


Figure 9a: The block diagram of the LabVIEW program to control the SXR reflectometer

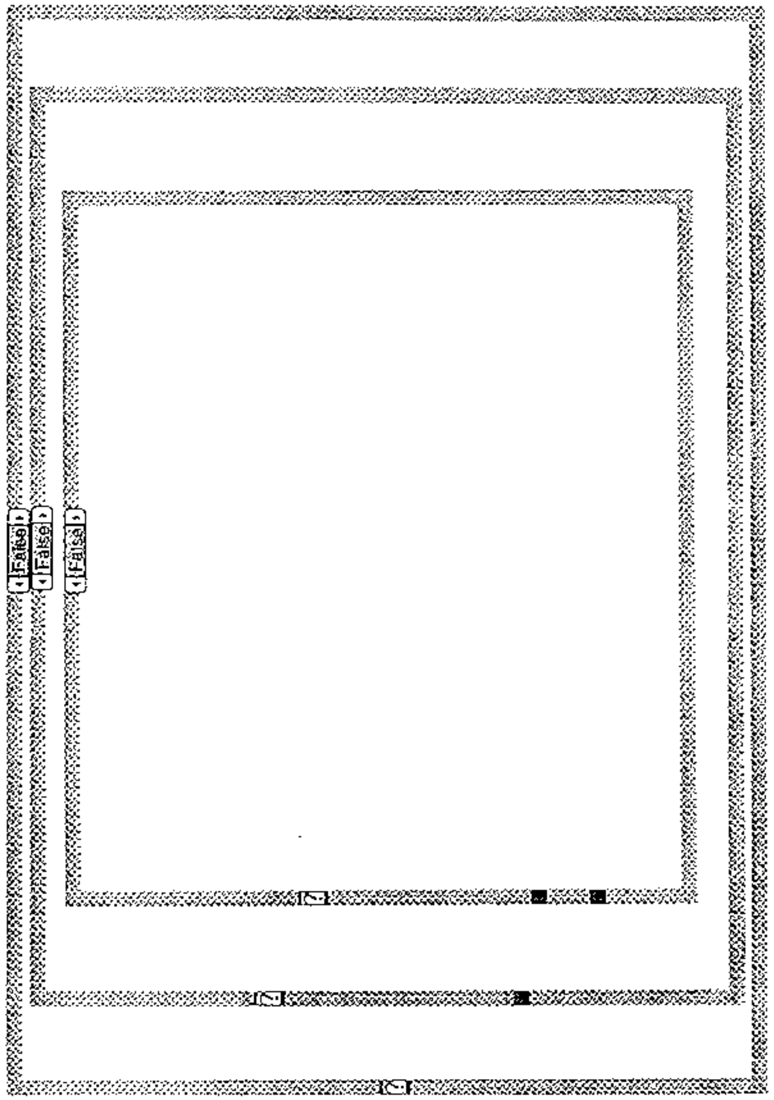


Figure 9b: The block diagram of the LabVIEW program to control the SXR reflectometer (continued)

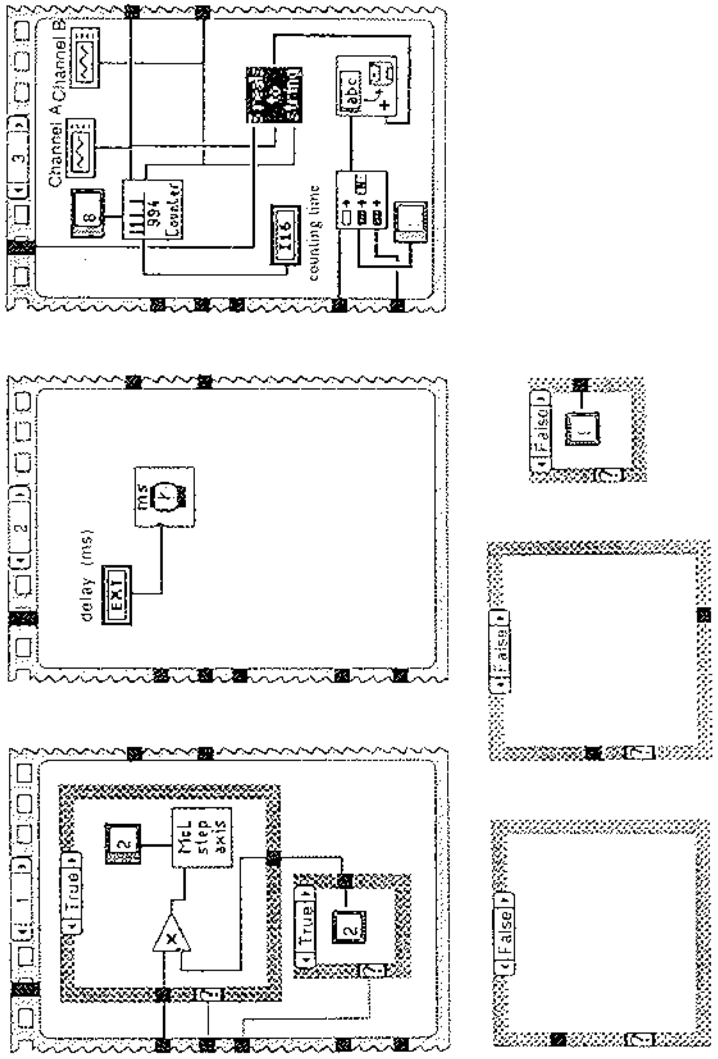


Figure 9c: The block diagram of the LabVIEW program to control the SXR reflectometer (continued)

Considering the 'true' state of the second-level case structure, the VI represented by 'create srs file' is then activated. This takes as input the directory on which to create the file (in the form of a string constant) , the filename itself (passed in from outside the structure), and the comment (passed from the string control on the front panel). This VI creates the file on the appropriate directory, writes in the comment string as header information, closes the file again, then returns a boolean 'success' flag to be wired to the third-level case structure. If the data file has not been successfully created for some reason, this boolean will be 'false', and the program will terminate. (Again, an appropriate error message would then be written to the screen from inside the 'create srs file' VI.) Note also that the filename string and the directory name string are wired to the edge of the next case structure, for subsequent use somewhere within it.

The main feature of the 'true' state of the third-level case structure is the LabVIEW representation of a FOR loop (as discussed earlier in section 2). This loop will execute for as many times as the value wired to 'N' in its top corner, which in this case is the digital control 'no of increments', whose value is set on the front panel. Inside this loop is another type of LabVIEW entity - a *frame sequence*. This is used when it is necessary to impose a particular order on the sequence of operations of a section of the code. In this case the frame contains 4 sequences (continued on figure 9c), which execute in the order 0, 1, 2 then 3.

Frames 0 and 1 step axes 1 and 2 respectively the appropriate number of steps (passed into the frame from outside from 'steps per increment'), but in each case the motor is only activated *if* it is actually switched on (as set by the axis 1 and axis 2 boolean controls on the front panel). The VI represented by 'McL step axis' takes as input only the address of the axis (ie 1 or 2) and the number of steps to drive it. A complication is that in frame 1 (which drives axis 2), if axis 1 is also switched on (ie the mode is 'θ_1 and θ_2') then the number of steps to drive axis 2 is doubled.

Frame 3 is simply a time delay after stepping the motors (whose value comes from the front panel).

In frame 4 the VI '944 counter' counts for the appropriate time (again the value comes from the front panel control), and returns the counts from channels A and B on the right hand side of its connector pane. These values are wired to the strip charts for the two channels (which display them in real time), and also to the VI '3 reals to string'. This VI takes the total number of steps moved so far (as calculated outside the frame and passed in through the top edge), and the two count values, and produces a single formatted string. This string is appended to the data file previously created, with the VI 'open, append string, close', which also requires filename and directory information passed in from outside the frame.

One point to make about this program is that while seeming perhaps a little complicated and difficult to explain in words, it is in fact rather compact - the whole thing can be printed off on just three sheets (not counting the sub-VIs). It is also visually very easy to follow, once familiarity with the various LabVIEW symbols has been gained. Note also that for each 'true' case structure shown there is also a 'false' one somewhere in figure 9c - it should not be too difficult to work out which corresponds to which.

Another feature of LabVIEW which this program demonstrates is the way that different types of variable are passed across wires of differing appearance (evident in figure 9a).

3.6 Some Preliminary Results

The reflectometer has been in fairly constant use during the period late 1990-early 1991, and a great deal of useful data has been recorded on a variety of samples, including both gratings and multilayers. Figure 10 shows the results from one such scan (run number 310), in which the counts recorded are plotted against angle of incidence (θ_1) for a multilayer scan done using the 'θ_1 and θ_2' mode.

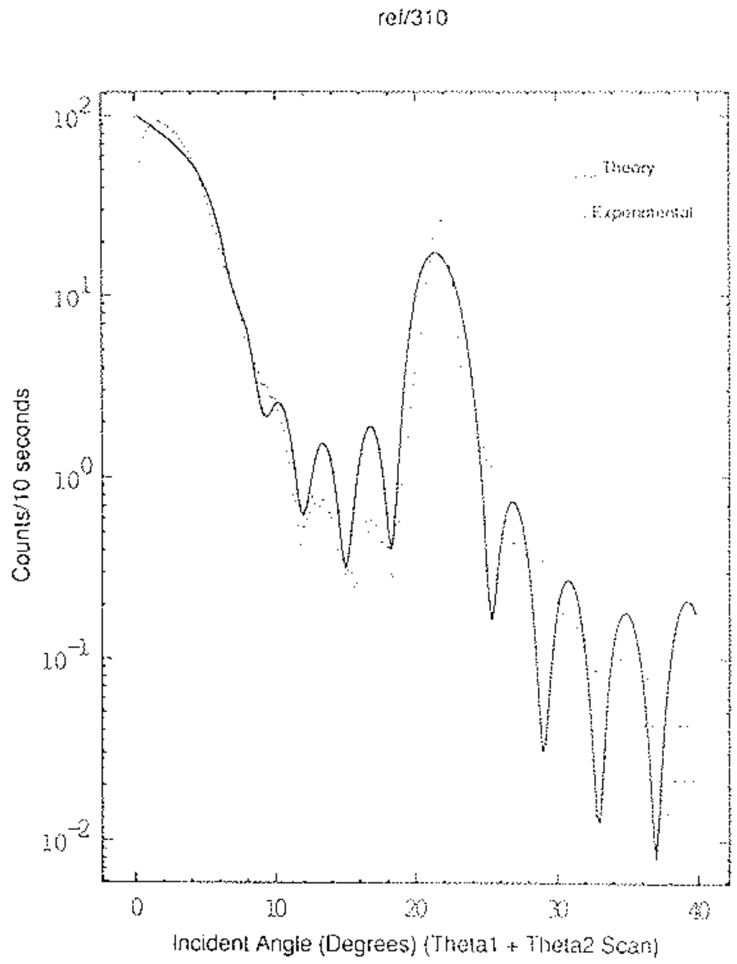


Figure 10: Typical results from a 'θ_1 and θ_2' multilayer scan

This type of data is analysed to give information on a range of multilayer parameters, including the local period, periodic errors, and the roughness of the interface between layers.

3.7 How Much Has LabVIEW Aided the Success of the Project?

The total time spent in developing the program described, given that a number of the sub-instruments had already been written, probably added up to about two or three days (of course, the program has received continuing modifications since this version was produced). It is doubtful whether this could have been achieved if LabVIEW had not been used.

Essentially, using LabVIEW has allowed us to concentrate on what is the most important aspect of the project - getting the reflectometer to function as required, and evaluating the performance of gratings and multilayers in an efficient manner. In other words, the software has become a genuine tool to support the overall project; a means to an end, instead of to some extent an end in itself.

4 The Future Use of LabVIEW by the Soft X-Ray Group

Our experience in using LabVIEW has already convinced us of its value for developmental and test purposes. For this reason, effort has been put into producing instrument drivers for much of the Soft X-Ray Group's GPIB-compatible laboratory equipment, examples of which include a Solartron 7150 DVM and a LeCroy 9400A oscilloscope. Our aim is to be able to 'mix and match' our equipment to some extent, so as to maximize both its flexibility and efficiency of use. To take an example, suppose a Solartron 7150 voltmeter used in some computer controlled set-up suffers a failure, and only a model 7150+ is available to replace it. Under LabVIEW the change could be made with the minimum of time and disruption, and might involve replacing only a single VI on the block diagram of the control program.

We have also looked seriously at the possibility of running a station (possibly 3.4) with LabVIEW, and have concluded that it would be perfectly feasible to do so. In the first instance the CAMAC setup would probably be retained (driven across a GPIB as outlined above), though in the future this could be replaced by separate GPIB devices, VXI, or even Macintosh plug-in boards. In this type of application, though, more than just the ease of programming must be considered. Others factors that are important are the general attractiveness of the user environment, the ease of data transfer and communication with other site machines, and of course the technical requirements of the station. Alternative options being considered include using a 386-based PC (also running CAMAC to begin with), or moving across to a VME system running OS/9 or UNIX. Though no decision has yet been taken in this respect, the matter is certainly being actively pursued.

5 Conclusions

It is not the intention of this document to seek to persuade the reader to use LabVIEW, nor to suggest that it is the ideal solution to every type of problem. On the contrary, the aim is to provide an objective assessment of its capabilities so that those unfamiliar with it may decide whether it fits their requirements. There is no doubt that despite its many useful features, there are still things about the product that may rule it out in some instances. These include the following:

- Many people may object to the idea of committing themselves to using a commercial product, since it may change and evolve with time, or even dissapear from the market completely.
- As with any other commercial package, the programmer is limited to the facilities and options provided. These may not be sufficient for what is required. For example, it is not possible to take full control of the Macintosh screen when a LabVIEW program is running, and the plotting utilities available may not always be suitable for the type of data to be displayed.
- While a graphical representation of code should be easier to understand than conventional lines of code, this might not always be the case - it depends of course upon how well the software has been written. Care should always be taken to break any application down into as many sub-instruments as possible, and to make each one as simple as possible. Whilst it is feasible to create a block diagram virtually as large as you wish (using scroll bars to move around it), in practice it is desirable to be able to view the whole diagram on the screen at once. For even moderately large programs this can present problems, particularly on a small-screened machine such a Macintosh SE/30.
- LabVIEW programs take up significant amounts of disk storage space. For example, the suite of reflectometer VIs described above (10 in all) requires 423 Kbytes of disk space (not including the standard LabVIEW library modules).
- LabVIEW has a few technical shortcomings which may or may not be put right in future releases. For example, it is not possible to open up an x-y graph on the front panel and plot points on it as and when they are available, say from inside a FOR loop. It is necessary either to wait till all the points have been collected before plotting them (say at the end of the loop), or to redraw the whole graph, axes and existing points each time a new point is added. Also, it is not possible to change plotting styles (points, line, dashed line etc) half way through a dataset. You could not for instance plot some points from a dataset with a circle, and the rest with a cross.
- For some applications other technical objections may arise. For instance, a GPIB cannot transfer data faster than about 1 Mbyte per second; whatever is placed on the end of it (CAMAC, VME etc) would be constrained by this.
- The cost of LabVIEW and a Macintosh computer to run it on may be too high for smaller applications.

Despite all of the above, if LabVIEW provides facilities sufficient for all your present or foreseeable needs, it certainly has many good points to recommend it:

- The graphical approach makes many aspects of programming easier, and also less prone to conventional error. When a program is being developed, for example, it is constantly checked for mistakes (such as a disconnected wire, or an integer number being wired to a string indicator), and will not run until they are put right. Many types of conventional 'syntax' errors are thus avoided.
- To some extent LabVIEW is self-documenting. Provided it is well written, the graphical code can be easier to read and understand than, say, 10000 lines of conventional FORTRAN.
- The time required to learn how to program in LabVIEW is vastly less than with a conventional language. It is hard to see how most of what would be needed to know could not be mastered by most programmers within perhaps a week or two.
- The development time for a LabVIEW solution will probably compare extremely favourably with that needed with a more conventional approach.
- The final product, in particular what the occasional user sees when running the software, can be given a pleasing and professional appearance.
- An extensive network of LabVIEW users now exist, and a library of freely available instrument drivers and more general VIs has been built up. (For instance, we obtained an instrument driver for our Solartron 7150 DVM in this way.) In fact, several manufacturers are now making instruments without conventional front panels at all, for use only in only computer controlled arrangements. This is making advanced instrumentation much cheaper.
- Because of its extensive range of instrument drivers, LabVIEW certainly 'demystifies' the whole process of software-hardware interfacing.

To sum up, whether or not LabVIEW will be useful to you depends very much upon your application. For general use as a laboratory aid in controlling GPIB equipment it is hard to fault. On the other hand, for more complex projects that might not change very much once operational, particularly where long development times have been set aside anyway, other solutions must of course be considered.

Any LabVIEW application should at the very least bring the following three benefits: it should be easy to develop, it should be flexible and easy to modify or maintain, and the front panel should present a pleasing, professional appearance which will also make the application easy to use.

6 Acknowledgements

Thanks are due to all members of the Soft X-Ray Project Team, particularly Howard Padmore, who both initiated the reflectometer project and was primarily responsible for LabVIEW being chosen to run it. Other group members have also been actively involved in the development of LabVIEW software, including Andy Smith and Kevin Hollis. Kevin Hollis made available the preliminary reflectometer results described in section 3.5.

Appendix A : Currently Available GPIB Instrument Drivers

GPIB Instruments	Class
B&K 1049	Sine Generator/Noise Generator
B&K 1051	Sine Generator
B&K 2977	Phase Meter
EDC 520A	Calibrator
EG&G 5208	Lock-in Analyzer
EG&G 5210	Lock-in Amplifier
Fluke 45	Digital Multimeter
Fluke 2240C	Datalogger
Fluke 5101B	Calibrator
Fluke 5440B	Calibrator
Fluke 8502A	Digital Multimeter
Fluke 8505A	Digital Multimeter
Fluke 8506A	Digital Multimeter
Fluke 8520A	Digital Multimeter
Fluke 8840A	Digital Multimeter
Fluke 8842A	Digital Multimeter
Gould 4072	Digitizing Scope
Gould 4074	Digitizing Scope
HP 437B	Power Meter
HP 438A	Power Meter
HP 3314A	Function Generator
HP 3325A	Function Generator
HP 3421A	Data Acquisition Unit/Scanner
HP 3437A	High Speed DC Voltmeter
HP 3456A	Digital Multimeter
HP 3457A	Digital Multimeter
HP 3458A	Digital Multimeter
HP 3478A	Digital Multimeter
HP 3497A	Data Acquisition Unit/Scanner
HP 3561A	Dynamic Signal Analyzer
HP 4192	LCR Meter
HP 4275A	LCR Meter
HP 5180A	Digitizer
HP 5316A	Counter
HP 5334A/B	Counter
HP 5335A	Counter
HP 5342A	Counter
HP 5345A	Counter
HP 6030A (31A, 32A, 33A, & 38A)	Power Supply
HP 6624A	Power Supply
HP 6632A (33A &34A)	Power Supply
HP 7470A	Digital Plotter
HP 7475A	Digital Plotter
HP 7550A	Digital Plotter
HP 8116A	Function/Pulse Generator
HP 8452A	Spectrophotometer
HP 8566B	Spectrum Analyzer
HP 8663A	Frequency Synthesizer
HP 8902A	Measuring Receiver
HP 8903A	Audio Analyzer

GPIB Instruments	Class
HP 8904A	Multifunction Synthesizer
HP 11713A	Attenuator/Switch Driver
HP 54100A	Digitizing Scope
HP 54110D	Digitizing Scope
HP 54111D	Digitizing Scope
HP 54200A	Digitizing Scope
HP 54501A	Digitizing Scope
HP 54502A	Digitizing Scope
HP 54503A	Digitizing Scope
HP 59501B	Power Supply Programmer
Iwatsu DS6121	Digitizing Scope
Iwatsu DS6612	Digitizing Scope
Keithley 192	Digital Multimeter
Keithley 195A	Digital Multimeter
Keithley 196	Digital Multimeter
Keithley 220	Current Source
Keithley 228	Voltage/Current Source
Keithley 230	Voltage Source
Keithley 485	Picoammeter
Keithley 617	Electrometer
Keithley 705	Scanner
LeCroy 9100	Arbitrary Function Generator
LeCroy 9400	Digitizing Scope
LeCroy 9420/50	Digitizing Scope
Leybold Inficon PG3	Vacuum Gauge Controller
Newport Corp. 835	Optical Power Meter
Newport Corp. 855C	Programmable Controller
Nicolet 320	Digitizing Scope
Nicolet 4094B	Digitizing Scope
Philips PM3350/3350A	Digitizing Scope
Philips PM5193	Function Generator
Philips PM6666	Frequency Counter
Precision Filters 6201	Filter
Precision Filters 6201C	Calibration Oscillator
Prema 4000	Digital Multimeter
Prema 5000	Digital Multimeter
Prema 6000	Digital Multimeter
Prema 6031	Digital Multimeter
Racal-Dana 1992	Counter
Racal-Dana 1994	Universal Counter
Solartron 7061	Digital Multimeter
Solartron 7081	Digital Multimeter
Solartron 7151	Digital Multimeter
Stanford Research 245	Computer Interface/NIM Crate
Stanford Research 250	Gated Integrator & Boxcar Averager
Stanford Research 400	Gated Photon Counter
Stanford Research 510	Lock-In Amplifier
Stanford Research 530	Lock-In Amplifier
Tek 2230	Digitizing Scope
Tek 2430	Digitizing Scope
Tek 2432	Digitizing Scope

GPIB Instruments	Class
Tek 2465A	Analog Scope
Tek 390AD	Digitizer
Tek 7612 D	Digitizer
Tek 7854	Digitizing Scope
Tek 7912AD/HB	Digitizer
Tek 7D20	Digitizer
Tek AA 5001	Distortion Analyzer
Tek AFG 5101/5501	Programmable Arbitrary/Function Generator
Tek CG 5001/551AP	Calibration Generator
Tek DC 5004	Counter/Timer
Tek DC 5009	Counter/Timer
Tek DC 5010	Counter/Timer
Tek DM 5010	Digital Multimeter
Tek DM 5110	Digital Multimeter
Tek FG 5010	Function Generator
Tek PS 5004	Power Supply
Tek PS 5010	Power Supply
Tek RTD 710	Digitizer
Tek RTD 720	Digitizer
Tek SCD 1000	Digitizer
Tek SCD 5000	Digitizer
Tek SG 5010	Programmable Oscillator
Tek SI 5010	Scanner
Tek 11400	Digitizing Scope
Tek 11800	Digitizing Scope
Wavetek 23	Function Generator
Wavetek 75	Waveform Generator
Wavetek 271	Function Generator

CAMAC Controllers

Kinetic Systems 3988	GPIB Crate Controller
LeCroy 8901/8901A	GPIB Crate Controller

CAMAC Instruments

Bi Ra 5301	ADC
Jorway 60A	Input Register
Joerger S12	Scaler
Kinetic Systems 3074	Output Register
Kinetic Systems 3075	Output Register
Kinetic Systems 3112	DAC
Kinetic Systems 3514	ADC
Kinetic Systems 3516	ADC
Kinetic Systems 3525	Temperature Monitor
Kinetic Systems 4010	Transient Recorder
Kinetic Systems 4020	Transient Digitizer
LeCroy 2228A	TDC
LeCroy 2249SG	ADC
LeCroy 2256AS	Digitizer
LeCroy 4434	Scaler
LeCroy 8232	ADC
LeCroy TR8818	Transient Recorder
LeCroy TR 8837F	Transient Recorder
LRS 2550B	Scaler

Serial Instruments

Analog Devices	
μMAC 6000	Modular I/O Processor
Apple Imagewriters I, II	Dot Matrix Printers
Mettler PM4600	Balance
Ohaus GT Series	Balance
Spex CD2A	Spectrometer Drive System
Tek 222	Hand-held Digitizing Scope