COMP

DL/SCI/TM86E

LENDING COPY

technical memorandum Daresbury Laborator

DL/SCI/TM86E

A NOVEL DATA ACQUISITION SYSTEM BASED ON A GENERAL PURPOSE CONTROL LANGUAGE

by

G. OSZLANYI and M.C. MILLER, SERC Daresbury Laboratory

APRIL, 1992

G92/84

Science and Engineering Research Council

DARESBURY LABORATORY

Daresbury, Warrington WA4 4AD

ISSN 0144-5677

A Novel Data Acquisition System based on a general purpose Control Language

G.Oszlanyi and M.C.Miller

31st March 1992

Contents

1 Introduction

Flexible data acquisition systems are of primary importance for most experimental methods. Unfortunately the logic of experiment control in most cases is obscured by mixing it with hardware specific and data analysis related code. This ends in large and unmanageable code and otherwise valuable data acquisition programs have to be abandoned just because of changes in hardware or computer environment. Often very similar but equally specific data acquisition programs are written over and over again.

In this document we outline a different approach to data acquisition programs. We describe a general purpose control language together with a layered hardware access which was successfully used to control a four-circle, triple axis single crystal diffraction experiment at Edinburgh University.

The main initiative for such a system was the new six-circle single crystal diffraction station currently being built at the 16.3 beamline of the Daresbury Synchrotron. A traditional control program could not possibly cover the data acquisition needs of the wide range of expected experiments at this new station.

Although the data acquisition system was initiated by station 16.3, it will suit other stations at the SRS equally well. In this document we will try to highlight the general principles of the completed data acquisition system. The large number of macros specifically needed to operate a four circle diffraction station have been fully tested and form an essential part of that data acquisition suite. For clarity however, only examples of general interest are included in this document.

Both previous experience in writing single crystal diffractometer control [1] and careful planning of the program structure [2] helped to realize the data acquisition program long before station 16.3 is operational.

2 Program Structure

Figure 1 shows the overall structure of the control program. As the figure suggests the three parts of the program, namely the Command Language Interpreter (CLI), Generic Control Layer and Specific Control Layer are logically separate programs.

All the logic flow control, mathematical calculations and data formatting are managed by the Command Language Interpreter (CLI). However the actual procedures are contained in macro files which are small, easy to read and easy to modify ASCII program files.

In most cases the user of the program might want to use only a few macros (eg. scan macros), and forget about all the freedom offered by such a command language. On the other hand, more experienced users can create their own specific procedures or calculations, which is possible even during runtime. In fact the control program is an empty box without variables or calculations, and it becomes a four circle control program only after executing an initializing macro.

This can be achieved because the Command Language Interpreter is free of the burden

of direct hardware control. Only the true logical flow of the experiment is contained in macros and their interpretation is the task of the CLI.

The Command Language Interpreter does not hold information on the actual hardware. It communicates with the Generic Control Layer, sends commands and receives data. The very general commands are translated to the specific hardware calls depending on the the type of hardware and function to be performed. The Specific Control Layer guarantees that different types of hardware performing similar functions can be transparently called.

Control programs are never truly portable. Such a program might depend on the particular hardware used, the particular bus, programming language, operating system, graphics interface etc. The only way to circumvent this problem is to confine the hardware dependent part of the program to separate hardware Specific Control Layers and to rely on system specific solutions as little as possible.

All this might sound rather complicated but the actual fact is that the resulting code is much shorter and easier to maintain than an equivalent hard coded control program. More importantly the implementation of new experimental procedures does not require hard coding at all, and instead only slight changes in macro files are needed. Even a major upgrade in hardware will only affect the Hardware Specific Control Layer, leaving everything else intact.

Figure 2 shows how the control program relates to the operating system and other programs. There is no point in duplicating functions supplied already by other programs. The data acquisition control program should not pretend that it is a windowing operating system, an editor or a desktop publishing program. Although the graphical presentation of the data, system editors and networking support are all indispensable for the data acquisition program, these tools are widely available and should be used.

For security, a data acquisition program should write the data on disk files as soon as possible. Once the data is safely stored, the data acquisition program can give access to data analysis. This should be performed by other programs, which suit users individual needs. This does not prevent data analysis while data is being collected, as multitasking operating systems can manage files shared by different processes. In the course of a time consuming data collection the data acquisition program can run in the foreground while data analysis is done in the background.

The variety of graphics and data analysis programs is immense. It is very probable that simple graphics support included in the data acquisition program would sooner or later become inappropriate for certain applications. We would like to stress, that in principle such a graphics interface can be easily included in the data acquisition program, but in most cases we do not find it a justifiable decision.

As far as the code is concerned we used strictly ANSI standard C language to ensure future portability of our program.

3 Command Language Interpreter

The Command Language Interpreter uses an interpreted instrument control language. Its syntax is very simple and the number of language elements is small. The few syntactical rules and control statements needed to use the language are contained in the following sections and in the corresponding appendix.

Although we did not aim to emulate any of the existing high level programming languages, some similarities to C can be found. This is pronounced in the case of relational operators, input/output formats and control flow statements. The similarity helps the user to learn the syntax much faster than if we had imposed a completely new syntax.

3.1 Command Line and Command Block Syntax

The natural unit of user input is a line of code, derived from the keyboard or a disk file. One such line can consist of one or more commands which are separated by semicolons or the end of the line.

More than one command can form a command block, which can extend to a number of lines. Control flow statements always involve the use of command blocks, so their use is described in Subsection 3.6. A single command is simply passed for execution after parsing. In the case of command blocks, input is continued until a complete block can be passed for execution. If the input comes from the keyboard this means waiting for subsequent commands from the keyboard.

A single command consists of a number of associated arguments which are separated by white space characters. The interpretation of a single command depends only on its first argument. It can be a control flow statement or a non-control statement. If it is a non-control statement, the CLI should be able to decide if it is an inbuilt function, a macro or it is simply an assignment. The syntax is simple but strict, any assignment should have the following form:

```
<variable_name>=<expression>
```

No spaces can be embedded in an assignment, and no inbuilt function or macro name can contain the equal sign.

Inbuilt functions have a higher priority than macros. If an inbuilt function and a macro file exist with identical names, only the inbuilt function will be found. Because of the small number of inbuilt functions, this does not seriously limit the possible choice of macro names.

The command syntax is identical for inbuilt functions and macros:

```
command <input_arg1> <input_arg2> ...  = <output_arg1> ...
```

The only extra syntactical rule is that the equal sign should be surrounded by white space characters. Input arguments can be expressions or variables, output arguments can be

variables only. The exact matching of input and output variables is more strict in the case of inbuilt functions. Macros should be written to check that the number of parameters and the dimensions of these parameters are appropriate.

The Command Language Interpreter is case insensitive. It converts all input arguments not in quotes to lower case, and its internal variable and function names are all in lower case. Anything within quotes is left as is, which is important for the use of upper case strings.

3.2 Data Types

There are two data types supported by the CLI: matrix-numeric and string. Matrix means n by m double precision storage in general. As a special case scalars are represented as 1 by 1 matrices, vectors are represented either as n by 1 or as 1 by n matrices. Strings are stored as a sequence of ASCII characters terminated by ASCII null.

All variables must be declared before they can be used. Variables can be global to the whole of the program or local to a particular macro. It is a good practice to use only local variables, and confine the declaration of globals to an initializing macro. Such globals are existent throughout the lifetime of the program, which is important for variables like wavelength, unit cell parameters or orientation matrix in the case of a single crystal diffraction experiment. Global or local variables are declared through the use of the two inbuilt functions, `global` and `local`. The syntax is described in Appendix 10.2. It is possible to get information on the dimensions of the declared variable at a later time. The inbuilt function `size` serves this purpose.

If numeric variables are simply referenced by their name then all elements of the variable are referenced. All numeric variables can be referenced as if they were vector or matrix arrays:

```
variable[index]
variable[first_index,second_index]
```

When referenced by one index, the row continuous data storage should be taken into account.

A string must be declared to be of sufficient length to hold any data later assigned to it. The main use of strings is to store filenames, and they cannot form string expressions. Special attention must be taken not to use string variable names which might interfere with numeric variable names. String constants must not be enclosed within special characters (e.g. quotes).

3.3 Expressions

Constants and variable numeric elements can form arithmetic expressions using arithmetic operators and mathematical functions. It is important to note that only scalars and single elements of vectors and matrices can be used in expressions, the use of complete arrays is not allowed. Note that no white space characters are allowed anywhere within an expression.

Appendix 10.1 shows the arithmetic operators and mathematical functions supported. For convenience, trigonometrical functions use degree units.

Mathematical functions have higher precedence than arithmetic operators but there is no arithmetic operator precedence at present. The correct operator precedence can only be achieved by the use of parentheses, which in any case is good practice.

By combining arithmetic expressions with relational and logical operators, relational expressions can be formed. These are used in conjunction with control flow statements such as `if` and `while`. Appendix 10.1 shows the list of relational and logical operators. The syntax for the simplest relational expression is:

```
<arithmetic_expression>-<relational_operator>-<arithmetic_expression>
```

More complex relational expressions can be built from simple ones and logical operators in the following way:

```
<relational_expression>-<logical_operator>-<relational_expression>
```

Mathematical functions and arithmetic operators have higher precedence than relational or logical operators. Correct relational and logical operator precedence can be achieved by the use of parentheses. No white space characters are allowed within either arithmetic or relational expressions.

3.4 Inbuilt Functions

Inbuilt functions are hard coded elements of the language, which provide the minimum functionality to build up macros. Some language elements, like variable declaration or hardware functions, must necessarily be implemented as inbuilt functions. In the case of others (eg. `inv` for matrix inversion) there is the possibility of using macros instead, but the gain in speed favours the inbuilt function solution.

Although it is quite easy to extend the range of inbuilt functions, we have tried to minimize their number. Appendix 10.2 shows the complete list of inbuilt functions available.

The implemented inbuilt functions fall into one of five categories:

- Hardware function related
- Language syntax related
- Input/Output related
- Calculation related

- Operating system shell and use of other programs

Hardware related inbuilt functions are very simple routines which do nothing else than move requests to and return data from the Generic Hardware Control Layer.

Setting up a new experiment means that a few hardware related inbuilt functions will have to be written. This will happen only if a completely new hardware class is included in the system. Adding a different type of equipment with similar existing functionality will not affect the Command Language Interpreter.

3.5 Macro Calling Convention

For any command that is not an assignment or an inbuilt function, the first argument is taken to be a macro name. Automatically the extension .MAC is attached to it and the current directory is searched. If the macro file is not found then an error is generated and the CLI returns to its highest level with a thorough clean-up procedure. If the macro file is successfully found, then all following input arguments in the invoking command are evaluated, and a local copy is created of them for use in the macro. Expressions and variables can be passed to the macro as input parameters and variables as output parameters. It is the macro's responsibility to make sure that the correct parameters with the appropriate dimensions are passed to it. This macro calling convention allows variable length input and output parameter lists. The use of these parameters is described in detail in Section 4 of this document.

3.6 Control Flow and Command Blocks

To modify the linear flow of inbuilt functions and macros, the use of control flow statements is needed. Appendix 10.3 shows the syntax for all control flow statements supported by the Command Language Interpreter.

The control flow statements `if`, `for` and `while` invoke the use of command blocks. All commands after the control flow statement up to an ending end statement form a command block, and the execution of the command block is affected by the control flow statement. Control flow statements can be nested, but each level should end with a matching end statement.

The control statement `break` allows escape from the smallest enclosing for or while loop similar to the C language syntax. The control statement `continue` can be used as a way to speed up these loops.

Simple commands are executed immediately after they are input from the keyboard or a macro file. When input comes from the keyboard, it is sometimes advantageous to execute commands together, without for more input, even if the execution of the commands is a linear sequence.

For this purpose there is a fourth special "control flow" statement: `begin`. Begin waits for input of all commands up to the finishing end and then executes the whole command block without stopping for command input.

3.7 Input and Output

The implementation of input and output is a dangerous pitfall for data acquisition programs. Writing (and reading) fixed format data anticipates that we know before the actual measurement which measured quantities to save. This is the case with simple measurements, but not for experiments envisaged at station 16.3. From simple single axis scans, through temperature scans to reciprocal space coordinate vs. multichannel analyser spectrum data, all kinds of data formats will be needed. A modestly complicated hardware can allow even more diverse sorts of measurements and a variety of data formats.

Therefore it was absolutely necessary to make the data format changeable during run time for both output and input. This is accomplished by relying on the ANSI standard C library functions. The inbuilt functions `fscanf` and `fprintf` make the input and output of scalar numeric data transparent to the program. The format specifiers of C can be found in any language reference. Note that all variables in this CLI are either double precision numeric or string, so only the use of floating point or string format specifiers is sensible.

The files on which these inbuilt functions operate must be opened for input or output. Files are opened and closed using the inbuilt functions `open` and `close` respectively.

There is a log file facility in the CLI. The inbuilt function `print` writes simultaneously to the screen and to all files which are open for output. This means that a complete history of the measurement can be written to the log-file, or to the printer (which is just another file for the operating system).

Two other inbuilt functions: `?` and `input` are used to print and input variables using a predefined format.

3.8 Operating System Interface

There is one special inbuilt function, which executes other programs through an operating system shell. The format of this system function is:

```
! <operating_system_command>
```

3.9 Error Handling

Error handling is crucial for safe data acquisition. Errors can be divided into two categories: those which are fatal, such as syntax errors, major hardware faults and those from which there is a way to recover. The latter can be more properly called status information. The Command Language Interpreter can handle both types of errors.

In the case of fatal errors, there is an error message describing the error type and the command which caused the error, causing the CLI to return to its highest level. In the case of nested macros, a thorough clean-up procedure is executed removing all local variables, closing macro files and at each macro level a message is given describing which command failed. Therefore it is possible to trace back the cause of the error quite accurately.

In the case of status information, it is the users responsibility to handle the information. Most often user macros generate such information and return it as an output parameter. This is either checked or ignored by the user.

3.10 Help

The inbuilt function `help` gives help information on inbuilt functions and macros. While help for inbuilt functions is included in the hard coded routines, macros must contain their own help as described in section 4.

It is sometimes necessary to get information on the available variables. The inbuilt function `who` shows all global and local variables declared in the program.

4 Macros

Macros can be best understood by using and modifying them. Therefore we provide a number of working example macros in Appendix 10.4.

4.1 Parameter Naming Convention

Once a macro is invoked, two local variables are always created: `nargin` and `nargout`. These are scalars and their value is set to the number of input arguments passed to the macro and the number of output arguments expected from the macro. These two local variables always exist, but their value is zero when there are no input or output parameters.

Once a macro is invoked, a local copy is made of all input parameters and a new descriptor is created for each input and output parameter. This descriptor holds the name, address, and dimensions of the parameter. The name of these parameters is left blank however, and it is the macros responsibility to give local names to them. The inbuilt functions: `in` and `out` serve this purpose. The first thing any macro should do is to check the number of input and output parameters and give them sensible local names. Only after naming can the macro use the parameters.

Then the macro executes exactly the same way as if the input came from the keyboard, so that single commands and complete command blocks are executed. The only difference is that the execution of the macro is terminated either if an end of file is found in the input or if the command `return` or `error` is encountered. These are inbuilt functions which simply terminate the execution of the macro with a success or error return status.

4.2 Help for Macros

It is generally a good idea to keep code and help for that code together, because it is then easier to keep both up to date. This however, usually infers a penalty in terms of storage and performance. In the case of macros this problem does not arise, as all help for a particular macro is placed at the end of that macro file. It does not slow down program execution, as it is never read during macro execution time. However, if help is required, all executable lines are skipped before display of the help text.

5 Generic Hardware Control

5.1 Introduction

The generic hardware controller layer of the software provides a general purpose, hardware independent access to functions which may be provided by the various types of hardware present in the system.

Programmers know what real functions are supported by a particular type of chosen hardware – for example in a motor driving system – and traditionally, these very specific functions are accessed from many points in a monolithic application program. The very specific motor hardware parameters are freely mixed with user parameters such as incident beam wavelength or position offsets. The data acquisition program therefore becomes very tightly coupled to the chosen instrument control hardware, and the situation becomes even worse when operating system specific features are liberally invoked throughout the program. When the inevitable time comes to change the hardware or the operating system (or both!), a huge amount of effort is required to modify the program at numerous places.

5.2 Structure and Operation

Our data acquisition philosophy involves first deciding on all the hardware functions required by the application program without considering exactly how they would be implemented by a particular hardware driving system, such as a motor control system. All these functions are then made available by calling a generic motor controller software layer with a single point of contact with the application. At this interface, all transferred parameters such as positions are in user units e.g. degrees, rather than units specific to the motor driving hardware itself. The motors are referenced as name strings and not parameters required at the hardware level, such as axis numbers. Within the motor generic controller, a dynamic table is kept of which motors are controlled by the various hardware types supported by the generic layer. This is used to call the appropriate specific hardware controller to carry out the requested motor function. An example of this may be the driving of a motor to an absolute position (user units) with a wait for the end of movement. The generic motor layer invokes the specific hardware controller to carry out that function in hardware units derived from the user ones provided. After completion, a final position in user units is returned to the application layer, together with a status word for that motor

which defines the success or otherwise of the drive. The format of the status looks the same for all motor types supported.

A vital element is the hiding of the hardware from the application layer and the user, so he can drive any motor the same way without having to know details of the motor type in question.

The internal motor parameters are hidden from the application layer and only those layers which actually need to use the parameters have them stored there. The generic controller layer therefore stores all parameters which are independent of motor type and so are applicable to all motors. These parameters are stored in disk files which are read when the generic controller receives an initialise function request from the application. Some motor functions, such as setting a motor gearing ratio, would not result in any hardware access and can be entirely executed by the generic layer alone. This is completely transparent to the application layer. Other functions may require some handling by the generic layer together with further execution by the relevant hardware specific layer.

Not every specific hardware type will be able to support the functions requested to the same level. It may be the case for example that a function is not available as a single hardware instruction but it can be made available by a software mapping which is entirely transparent to the application. An example of this could be the emulation of a motor move to absolute position by a combination of the two functions, get current position and move relative to current position. For certain very specialised functions, they may not be available in some specific hardware controllers at all and in that case an appropriate error status would be returned for that function request.

5.3 Error Handling

After a call to the generic hardware controller, a single overall status is returned to the Command Language Interpreter. There may be a number of instruments involved in each function call and so a returned success status indicates that all have succeeded and no further error checking is required. In the case of error, a number of status codes are returned which do not conflict with those used by the application layer for its own error handling. The most common overall status simply indicates that not all instrument functions succeeded. Further information is available in the hardware status word which is returned for each instrument involved in the function call.

This hardware status word contains two ranges of status values reflecting messages from the generic layer and the hardware specific layer. It contains values indicating both failures in internal software parameter checking and a representation of the real hardware status, e.g. a motor limit set. It is important that the status codes are compatible between different specific hardware controllers within a single generic hardware type, to allow transparent error detection and recovery by the application layer.

5.4 Daresbury Group Effort

These ideas have been included in the Data Acquisition Group Wiggler Motor Working Party efforts to standardise and improve the software for driving all types of motor at Daresbury from application programs [3].

This modular approach to motor driving allows group members to independently program a number of specific motor controllers separately, once the parameter passing mechanism and details have become fixed.

Details of data structures have not yet been finally agreed but this has not prevented a fully operational generic and specific motor controller being written for and implemented on the Edinburgh University instrument. This currently uses provisional data structures for the generic layer internals and interface, but will be modified to use generally agreed structures when they are available, together with the full list of motor functions.

6 Specific Hardware Control

6.1 Structure and Operation

The hardware dependent software layer consists of a number of discrete modules, one per hardware type, with exactly the same interface to the generic hardware layer. This allows conditional calling of the relevant specific controller which is transparent to the application and ensures that adding a new hardware type is as trivial a programming task as possible.

As with the generic controller, the internal parameters are entirely hidden from the layers above and also from other hardware specific controllers which will each contain a different set of parameters reflecting the different underlying hardware architectures. They are loaded from disk parameter files with an initialise function and held in dynamic storage. In the files there is any information required for general access to the hardware e.g. RS232 port numbers, together with a list of hardware specific parameters for each motor included. A crucial element is the indexing of parameters by the motor name string in the file which allows matching of motors in the two layers of the motor controller without having to remember obscure motor numbers and match their order in the files, record for record.

The hardware name or number is derived from the user motor name which is contained in a field of the specific hardware parameter file. A bonus of this very flexible scheme is that automatic aliasing of motor names can be provided by simply mapping two different motor names onto the same hardware name freeing the application layer from the complexity of input string manipulation.

This layer performs operations in hardware units which are obtained from the user unit values by means of generic layer conversion parameters. For motor driving, these are usually the gearing ratio and the user offset.

7 System Implementation

The Command Language Interpreter and hardware control in their current state of development have been successfully implemented on a 4-circle diffractometer in the Physics Department of the University of Edinburgh for single crystal data acquisition.

7.1 Hardware and Computer

The system has the overall features :

- Computer – 12MHz 286 (DECPC 220) running MS DOS 3.3
- Hardware control system – Harwell 6000 for both motor driving and counter timing via an RS232 interface
- C compiler – Microsoft C 6.0 in ANSI standard mode

Although the computer used is much slower than the 33MHz 386 PC (Viglen) on which the Command Line Interpreter was developed, the CLI overhead per instruction is still a negligible part of the execution time. Depending on the complexity of the instruction it is 0.3–2 milliseconds per instruction on the 386 and a factor of 8 slower on the DEC 286. In a data acquisition system where stepper motors are accessed via serial ports, motors start moving tens of milliseconds after receiving the move command and counting time is often much longer then a second, then the speed of the computer is not a limiting factor.

The following motor control functions were implemented :

- read parameter files
- initialise
- get current position
- get status
- move absolute synchronous (waits for end of drive)
- move relative synchronous
- move absolute asynchronous (returns without waiting for end)
- move relative asynchronous
- check position valid

For counter timing, a separate generic controller would normally be provided but with the Harwell, the scaler and timer appear as pseudo-motors and could therefore be easily supported by the motor controller. For scaling the extra function "count synchronous" was needed to complete the list.

7.2 Macros

A large number of macros can be developed without actually using the hardware. These were written before implementing the data acquisition system in Edinburgh and were tested on site and only slightly modified. The few macros which had to be written from scratch were site specific ones, as switching between detectors used in double crystal and triple crystal mode.

All four circle macros have been successfully tested.

- One axis scans
- Coupled two axis scans
- Search for reflections
- Reflection centring
- Orientation matrix determination from two reflections with known unit cell
- Orientation matrix determination from three reflections with unknown unit cell
- Least squares refinement of orientation matrix
- Bisecting ($\omega = 2\theta/2$) angle calculation
- Top reflection ($\chi = 90$) angle calculation
- Simultaneous move of all four motors to calculated position
- Reciprocal space scans

8 Portability

The portability of the program is based on four factors:

- Layered program structure, well localized hardware specific code
- The use of macro files instead of hard coded procedures
- Keeping data acquisition and data analysis separate
- The use of ANSI standard C programming language

The working Edinburgh program, written on a PC was implemented on the Daresbury Convex mainframe computer without any change in the source code. This does not only demonstrate portability, but such a program version makes the future use of station 16.3 more efficient. Users can get acquainted with the control program much before actually using the station.

9 Acknowledgements

Thanks are due to Bob Cernik for his support and proof-reading of this document and to Ursula Cox, Peter Hatton and Hugh Voss of Edinburgh University Physics Department for provision of their instrument and assistance.

10 Appendix

10.1 Expressions

In the following we list the arithmetic, relational, logical operators and mathematical functions which can be used to form arithmetic or relational expressions.

- Arithmetic Operators

  + Addition

  − Subtraction

  ***** Multiplication

  / Division

  ∧ Power of operator

- Mathematical Functions

  round(x) Round x to nearest integer

  abs(x) Absolute value of x

  sqrt(x) Square root of x

  sin(x) Sine of x – x in degree units

  asin(x) $sin^{-1}(x)$ – in degree units

  cos(x) Cosine of x – x in degree units

  acos(x) $cos^{-1}(x)$ – in degree units

  tan(x) Tangent of x – x in degree units

  atan2(y,x) $tan^{-1}(y/x)$ – in degree units

  n180(x) Transform x in the angle range $[-180, +180]$

- Relational and Logical Operators

  == Equal

  != Not equal

  < Less than

  <= Less than or equal

  > Greater than

  >= Greater than or equal

  && Logical AND operator

  || Logical OR operator

10.2 Inbuilt Functions

In the following we list the available inbuilt functions by category.

- **Hardware function related inbuilt functions**

  initmot Initialize motor controller. Return status in variable.

  ```
  Syntax: initmot <motor_name> = <variable>
  ```

  getpos Get the position of the motor in user units and store it in the variable.

  ```
  Syntax: getpos <motor_name> = <variable>
  ```

  getstat Get the status of the motor and store it in the variable.

  ```
  Syntax: getstat <motor_name> = <variable>
  ```

  movabs Moves motor to absolute position waiting for the move to finish. The target position in user units is given by the expression.

  ```
  Syntax: movabs <motor_name> <expression>
  ```

  movrel Moves motor relative to current position waiting for the move to finish. The relative move in user units is given by the expression.

  ```
  Syntax: movrel <motor_name> <expression>
  ```

  movabsa Moves motor to absolute position asynchronously, ie. not waiting for the move to finish. The target position in user units is given by the expression.

  ```
  Syntax: movabsa <motor_name> <expression>
  ```

  movrela Moves motor relative to current position asynchronously, ie. not waiting for the move to finish. The relative move in user units is given by the expression.

  ```
  Syntax: movrela <motor_name> <expression>
  ```

  waitmot Wait for motor to finish. Used in conjunction with asynchronous move commands. Return status in variable.

  ```
  Syntax: waitmot <motor_name> = <variable>
  ```

  count Count with the counter for the time specified by the expression. Return the counts in the variable.

  ```
  Syntax: count <counter_name> <expression> = <variable>
  ```

- **Language syntax related inbuilt functions**

  global Declare numeric variables global to the whole of the program.

  ```
  Syntax: global <scalar_name>
  Syntax: global <vector_name> <n>
  Syntax: global <matrix_name> <n> <m>
  ```

local Declare numeric variables local to a particular macro.

```
Syntax: local <scalar_name>
Syntax: local <vector_name> <n>
Syntax: local <matrix_name> <n> <m>
```

global$ Declare strings global to the whole of the program. The length is the maximum length of the string the variable can hold.

```
Syntax: global$ <string_name> <length>
```

local$ Declare strings local to a particular macro. The length is the maximum length of the string the variable can hold.

```
Syntax: local$ <string_name> <length>
```

size Get the size of a variable as row and column numbers and return them in a vector variable. Size will return two numbers even if the variable in question is a scalar, but then both numbers are set to one. If the variable is not defined then both numbers returned are set to zero.

```
Syntax: size <variable> = <vector_variable>
```

in Gives local names to the input parameters passed to the macro. These names are local to the macro and the parameters can be used only after naming them.

```
Syntax: in <local_name_1> <local_name_2> ...
```

out Gives local names to the output parameters expected from the macro. These names are local to the macro and the parameters can be used only after naming them.

```
Syntax: out <local_name_1> <local_name_2> ...
```

return Causes the CLI to return from the current macro one level back with a "success" status.

```
Syntax: return
```

error Generates user error and causes the CLI to return from any depth of nested macros to its highest level.

```
Syntax: error
```

exit Exit from the data acquisition program.

```
Syntax: exit
```

help Help without arguments shows the list of inbuilt functions available. Help with one argument shows help information on the particular inbuilt function or macro requested.

```
Syntax: help
Syntax: help <inbuilt_function_name>
Syntax: help <macro_name>
```

who Shows all variables declared together with their type (numeric or string) and dimensions.

```
Syntax: who
```

- **Input/Output related**

open Open a particular file for input or output depending on the mode. Mode can be: a - append, w - write or r - read. If mode is omitted, the default append mode is used. Open without any arguments will show the list of all files open.

```
Syntax: open <file_name> <mode>
Syntax: open <file_name>
Syntax: open
```

close Close a particular file that was open for input or output. Close without arguments will close all files open.

```
Syntax: close <file_name>
Syntax: close
```

? (Question mark) Show variables using internal format.

```
Syntax: ? <variable_1> <variable_2> ...
```

input Show the current value of the variable and prompt for a new input value. In the case of numeric variables the new input value can be a constant or an arithmetic expression. In the case of strings it can only be a string constant. If no new value is entered, the old value is preserved. If the variable is not a scalar or an element of an array, then input will prompt for all elements of the array. A non printable input character (eg. CTRL+A) will generate an error and stop the query.

```
Syntax: input <variable>
```

print Show variables with the format specified by the format_string. Output simultaneously to all files open for output. Print with only a format_string argument can be used to print messages. Print without arguments prints a new line (ASCII 13).

```
Syntax: print <format_string> <variable_1> <variable_2> ...
Syntax: print <format_string>
Format: %f,%e,%g,%s with modifiers
Example: print "TTH=%-8.3f COUNTS=%g" twotheta counts
```

fprintf Similar to print, but output only to one file specified by the first argument. This file must be previously opened for output.

```
Syntax: fprintf <file_name> <format_string> <variable_1> <variable_2> ...
```

fscanf Input the values of the variables from the file using the format given by the format_string. The file must be previously opened for input.

Syntax: `fscanf <file_name> <format_string> <variable_1> <variable_2> ...`

- **Calculation related inbuilt functions**

zero Set all elements of variable to zero.

```
Syntax: zero <variable>
```

copy Copy all elements of variable_1 to variable_2.

```
Syntax: copy <variable_1> = <variable_2>
```

row Copy a particular row given by row_expression from a matrix to a vector or vice versa.

```
Syntax: row <row_expression> <matrix_from> = <vector_to>
Syntax: row <row_expression> <vector_from> = <matrix_to>
```

col Copy a particular column given by column_expression from a matrix to a vector or vice versa.

```
Syntax: col <column_expression> <matrix_from> = <vector_to>
Syntax: col <column_expression> <vector_from> = <matrix_to>
```

det Calculate the determinant of the matrix and store it in the variable.

```
Syntax: det <matrix> = <variable>
```

inv Invert the matrix and store in inverse_matrix.

```
Syntax: inv <matrix> = <inverse_matrix>
```

prod Multiply variable_1 by variable_2 and store the result in variable_3. Prod works equally well for matrix-matrix, matrix-vector and vector-vector multiplications if the dimensions of the variables match.

```
Syntax: prod <variable_1> <variable_2> = <variable_3>
```

transp Transpose the matrix and store it in transposed_matrix.

```
Syntax: transp <matrix> = <transposed_matrix>
```

vlen Calculate the length of the vector and store it in the variable.

```
Syntax: vlen <vector> = <variable>
```

vcross Calculate the vector cross product of vector_1 and vector_2 and store it in vector_3.

```
Syntax: vcross <vector_1> <vector_2> = <vector_3>
```

- **Operating system shell and use of other programs**

! Exclamation mark Executes operating system commands or other programs through an operating system shell.

```
Syntax: ! <operating_system_command>
```

10.3 Control Flow Statements

end End one of the `for`, `while`, `if` and `begin` control flow statements. If control flow statements are nested, each of them must be ended by a matching end statement.

```
Syntax: end
```

for First assign to the control_variable the start_expression value and then execute the commands enclosed by the `for-end` statements. Increase the control_variable's value by step_expression and execute the commands again. At the beginning of each cycle check if the control_variable's value exceeds the end_expression. If it does, then quit cycling and continue execution at the command following the end statement. Omitting the step_expression the default step of 1 is used.

```
Syntax: for <control_variable=start_expr> <end_expr> <step_expr>
        commands ...
        end
Example: for x=0 180 10
         ? x
         y=sin(x); ? y
         end
```

while Execute commands while relational_expression is true. Evaluate relational_expression at the beginning of the block.

```
Syntax: while <relational_expression>
        commands ...
        end
Example: i=0
         while i<100
         ? i
         i=i+1
         end
```

if Execute commands_1 if relational_expression_1 is true and continue execution at the command following the end statement. If it is false and there is an `elseif` statement in the if block then evaluate relational_expression_2. If this is true execute commands_2 and continue execution at the command following the end statement. If none of the `elseif` statement relational_expressions is true and there is an `else` statement specified in the `if-end` block then execute the commands enclosed by the `else` and `end` statements. If none of these conditions hold, then skip the whole `if-end` block and continue execution at the command following the end statement.

```
Syntax: if <relational_expression_1>
        commands_1 ...
```

```
          elseif <relational_expression_2>
          commands_2 ...
          ...
          else
          commands_n ...
          end
Example:  if x>100
          print "%g greater than 100"; print
          elseif x>10
          print "%g greater than 10 but not greater than 100"; print
          else
          print "%g is not even greater than 10"; print
          end
```

begin Execute commands unconditionally. Begin is not a true control flow statement, it affects only the way how commands are derived from the keyboard or macro files. Its main use is for executing commands from the keyboard without stopping between them for further input.

```
Syntax: begin
        commands ...
        end
```

break Escape from the smallest enclosing for or while loop, as defined in the C language.

```
Syntax: break
```

continue Speed up the smallest enclosing for or while loop, as defined in the C language.

```
Syntax: continue
```

10.4 Macro Examples

We have chosen the following macro files to illustrate the potential of using macros. The short help text at the end of the macro files should be self explanatory. This is the message that appears on the screen when help macro_name is typed during runtime. The four-circle calculations are based on the Busing-Levy [4] angle convention.

- ABSTTH.MAC - MOVE EXAMPLE

```
if nargin==1
   in pos
else
   help absth; error
end

local status; getstat tth = status
if status==0
   movabsa tth pos
else
   print "TTH MOTOR BUSY"; print
end
return

% absth <position> - MOVES TTH AXIS TO ABSOLUTE POSITION ASYNCHRONOUSLY
```

- CT.MAC - COUNTER EXAMPLE

```
if (nargin==1)&&(nargout==1)
   in tau
   out counts
else
   help ct; error
end

count channel1 tau = counts
return

% ct <tau> = <counts> - COUNT FOR tau SECONDS AND RETURN RESULTS
%                       IN VARIABLE COUNTS
```

- SCANTTH.MAC - SCAN EXAMPLE

```
if (nargin==4)&&(nargout==0)
   in start end delta tau
else
```

```
   col i hkl = hkl_list
   row i hkl = transp_list
   col i cart = cart_list
end

prod hkl_list transp_list = aux1
inv aux1 = aux1
prod cart_list transp_list = aux2
prod aux2 aux1 = ub
print "THE NEW UB MATRIX CALCULATED BY LEAST SQUARES"; print
? ub
calclat
return

% leastsq <n1> <n2> - CALCULATE UB ORIENTATION MATRIX USING 3 OR MORE
%                     REFLECTIONS WITH KNOWN INDICES AND CENTERED ANGLES.
%                     THE REFLECTIONS ARE STORED IN
%                     GLOBAL MATRICES HKLLIST[] AND ANGLIST[].
%                     USE ONLY REFLECTIONS FROM n1 TO n2 ON THE LIST
% leastsq           - DEFAULT: USE ALL REFLECTIONS ON THE LIST
```

11 References

References

[1] G.Oszlanyi
Ph.D. thesis (1990)
Eotvos Lorand University Budapest, Hungary

[2] M.C.Miller and S.Ahern
Specification for SRS Wiggler-2 Station 16.3 Data Acquisition System
DL/SCI/TM80E (1991)

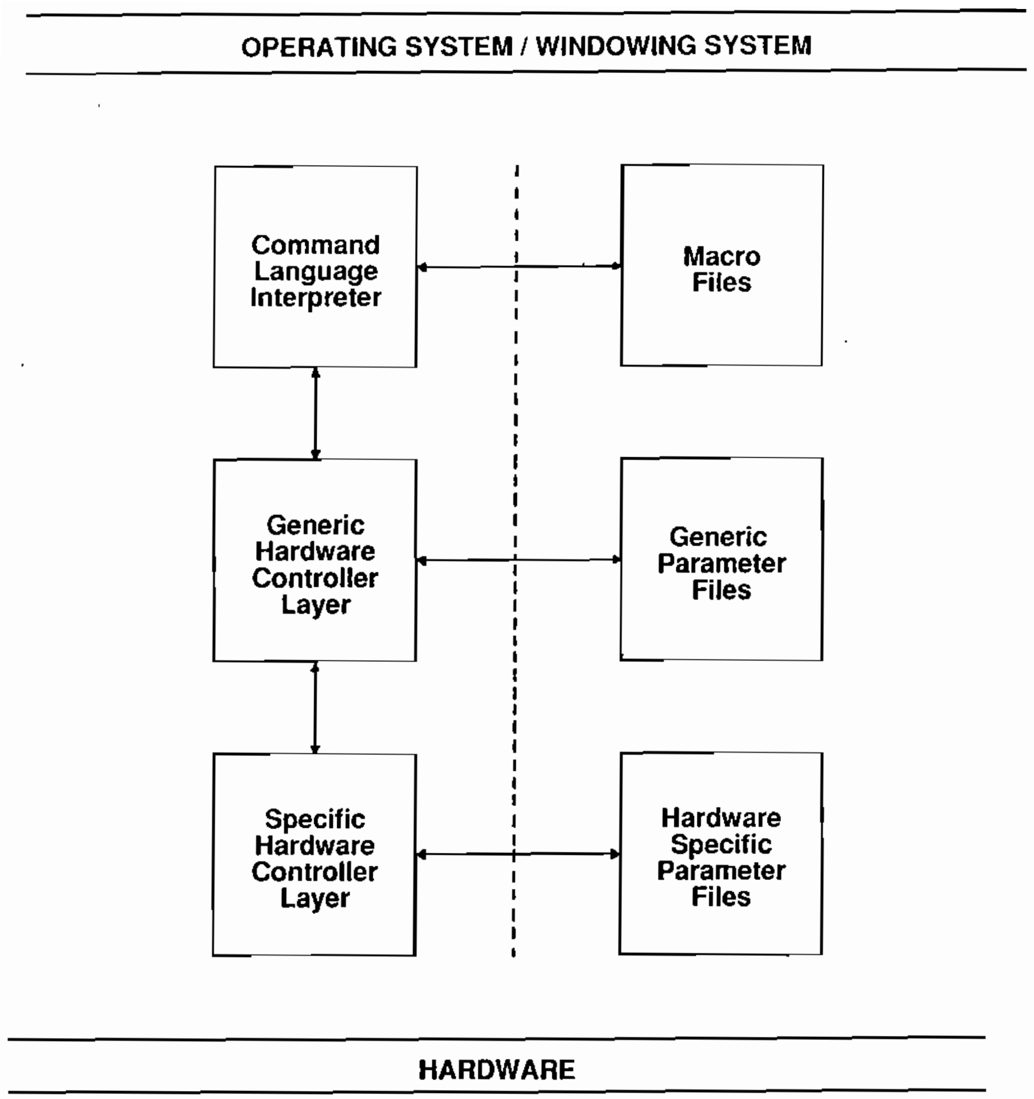
[3] Minutes of Wiggler-2 Motor Software Meetings (1992)
Data Acquisition Group internal publications
available from G.Mant (Chairman)

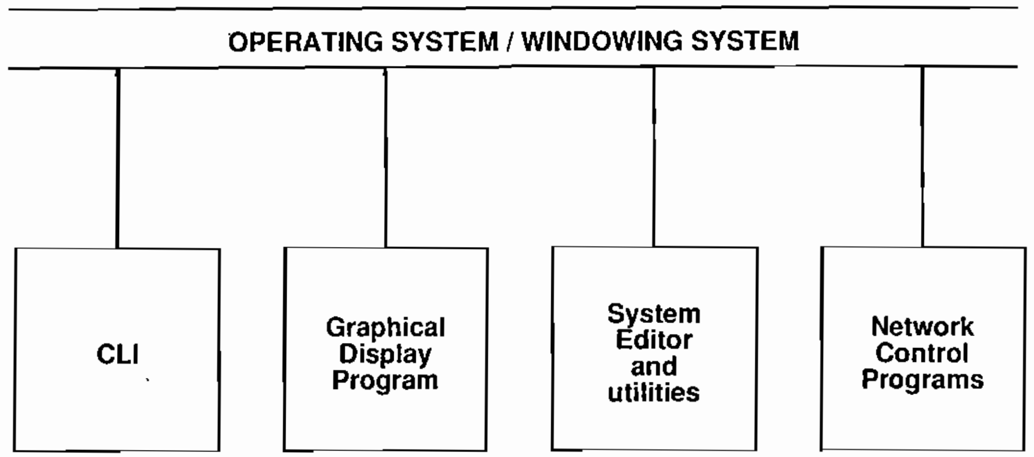
[4] W.R.Busing and H.A.Levy
Acta Cryst. *22*, 457-464 (1967)

Figure Captions

Figure 1. The overall structure of the Control Program.

Figure 2. The Control program in the context of the Operating System and other programs.


OPERATING SYSTEM / WINDOWING SYSTEM
Command Language Interpreter
Macro Files
Generic Hardware Controller Layer
Generic Parameter Files
Specific Hardware Controller Layer
Hardware Specific Parameter Files
HARDWARE


OPERATING SYSTEM / WINDOWING SYSTEM
CLI
Graphical Display Program
System Editor and utilities
Network Control Programs