ATLAS Internal Note
DAQ-NO-72
7 August 1997

Diagnostic Software for the ATLAS Level-1 Calorimeter Trigger

J.Leake, M.Landon, D.Rees, S.Hillier, A.Connors,
E.Eisenhandler, N.Gee, T.P.Shah

(UK Level-1 Calorimeter Trigger Group)

1 Introduction

The ATLAS Level 1 Calorimeter Trigger group has been involved in a continuing demonstrator programme to evaluate some of the technologies required for the final ATLAS Trigger system. This has involved building a number of VME modules. The diagnostic software described in the note was required in the first instance for the testing and debugging of individual VME modules. From the outset it was realised that the modules would be part of a larger and more complex system and hence a design goal was that the system should be easy to configure in different ways. It should allow access to registers and memories in individual modules, provide useful context related help and, at the same time, be extendible to larger and more complex systems consisting of several modules. Lastly, but most importantly, a key aim of any diagnostic software is to expose errors in the behaviour of part or all of the system. This means building into the diagnostic software the ability to model the behaviour of the hardware, starting from a single module and growing to a system consisting of many modules. We chose to use C++ as the programming language and the graphical user interface was implemented using Tcl/Tk. The software runs on VMEbus single-board computers under LynxOS (a real-time POSIX-compliant form of Unix); the display is on X-terminals (or X emulators) with a Motif-like style.

In this note we describe briefly the motivation for choosing the programming language and the graphics tool (Section 2). Section 3 outlines the Tcl commands that were added to the interpreter. These commands provide the link between the user interface and the class library. The user interface to the class library is discussed in Section 4, and Section 5 describes the essential components of the C++ class library itself. This implements the hardware components (the modules, registers etc.) of the trigger system and also contains the simulation classes. Details of the Diagnostics Software and the hardware can be found on the Web at:

`http://hepwww.rl.ac.uk/atlas/l1/l1_slice.html`

2 Choice of the User Interface and the Programming Language

In order to make the configuration of complex systems more transparent, and enable the diagnostic software to be run by design engineers, technicians and physicists without an in-depth knowledge of the software involved, a graphical user interface was considered essential. We decided to use Tcl/Tk as the software tool to generate this interface. Tcl/Tk is free and is readily extendible and widely portable. It is also relatively easy to learn and, using the Tk toolkit, simple scripts can be written which present a pleasing graphical interface to the user.

From the perspective of the software, digital electronics modules share common features: they all consist of sets of registers and memories. In particular the Level-1 trigger modules conform to some programming model guidelines. This suggested that the inheritance and modularity features of the object oriented programming approach would be natural for this project. We chose to use C++ as it is the most widely supported object oriented programming language, and the software development team also had some previous C++ programming experience. On LynxOS the C++ compiler is GNU g++, which is also freely available on almost every other computing platform. Using g++ enabled the code development to be done on more convenient operating systems such as Digital UNIX and HP-UX.

3 The Graphical User Interface using Tcl/Tk

One of the advantages of Tcl (Tool Command Language) is that the set of commands understood by the Tcl interpreter can be extended to include user defined commands written in C (or C++). This is the facility we used to implement the user interface to our class library. Extensions to Tcl are usually implemented as "packages". A Tcl package typically consists of one globally visible initialisation routine and a suite of `static` support functions. We discuss two of these packages below.

3.1 `l1` Package

The `l1` package declares two commands: `l1` and `l1obj`. The `l1` command is used to set and query global options affecting the whole package and the class library. The options it accepts are described in the subsections below. The `l1obj` command creates named C++ objects of class XComponent or its subclasses. It also creates a new Tcl command, with the same name, to control the object, issue commands to it, etc. This is similar to the scheme used by the Tk part of the Tcl/Tk package for creating widgets, where a command to create a widget of a given class (e.g. a button) also creates a Tcl command to configure the widget.

3.1.1 The `l1` Command

Many forms of the `l1` command are used to set or query global flags applying to the whole package. They take an optional value which is used to set the flag. If the value is omitted, the current value of the flag is reported. Two of the more useful commands are `fakevme` and `forever`. The `l1 fakevme` command queries or sets the "fake VME mode". If selected, this means that VME I/O is done to a small internal array and not to real VME. This feature was useful while developing the code. The `l1 forever` command can be set to cause any VME access to loop forever, repeating the same operation. This is useful for debugging the hardware with an oscilloscope. Other `l1` commands just give help or information about the package.

3.1.2 The `l1obj` Command

The `l1obj` command creates C++ objects of class XComponent or its subclasses. These classes are described further in section 5. `l1obj` also creates an associated Tcl command with the same name as the XComponent object. The Tcl command is used to control and communicate with the C++ object.

The following examples illustrate the use of the `l1obj` command:

```
l1obj rack r1 -height 20

l1obj txm txm2 -base 0xd20000 -module_id 0x2107 -number 2 -x 14
```

The first command creates a `Rack` object called `r1` which is 20U high while the second command creates a level-1 Transmitter Module (TXM) called `txm2` with a VME base address of d20000 (hex) and module ID of 2107 (hex). It is TXM number 2 in some human-friendly numbering scheme and is in slot 14 of its enclosing crate (if any). Additional commands are available to create `Crate` objects and add modules (e.g. a TXM module) to crates.

The new Tcl command `txm2` can then be used to read, write or test registers and memories in the module, e.g.:

```
txm2 /Control read
```

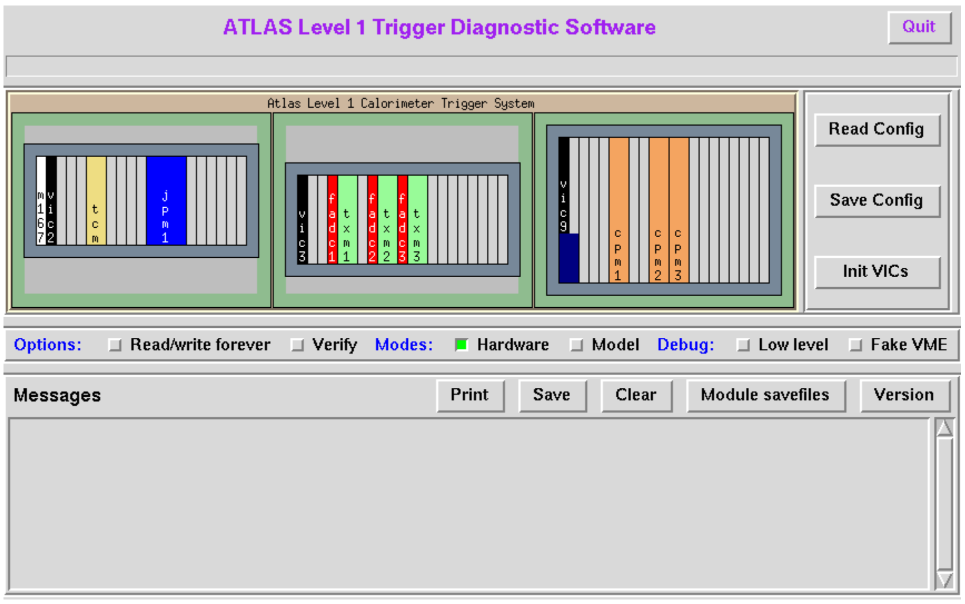
This command returns the value read from the control register of the TXM.

3.2 `component` Package

The `component` command creates a Tk widget which can be used to display a picture of an `XComponent` object and its subcomponents (if any). It is typically used to draw the `System` object and its component `Racks`, `Crates` and `Modules`. The `l1test` program described below invokes this command to display the system of racks, crates and modules under test.

4 The l1test Program

The `l1test` program is used for performing the testing and debugging. It starts by displaying the main configuration window (see Fig. 1) which consists of the following major elements: an "active" display of the known configuration; various buttons to load, save and initialise the configuration; a row of options; a scrolling text area for error messages and other information. The dominant feature of the "ATLAS Level 1 Trigger Diagnostic Software" window is a picture of the configuration showing a few (in this case three) racks, each with one or more crates containing several labelled modules. This configuration is initialised when the program starts but can be modified by reading a different configuration file by clicking the `Read Config` button. The `Init VICs` button initialises the vertical interconnect modules (VICs) connecting the VME crates, which downloads all the necessary inter-crate mappings. The configuration picture is active: it responds to the mouse in two ways. Firstly, as the mouse cursor is moved over the diagram, the information display above the picture shows some information about the object that is currently under the mouse cursor. Secondly, if the mouse is clicked over the picture a pop-up menu appears. This menu varies according to what lies under the mouse pointer. Over a module this menu shows a list of windows that can be opened for that module and other commands the module accepts; above an empty slot, the menu allows one to add a new



Options: ☐ Read/write forever ☐ Verify Modes: ☑ Hardware ☐ Model Debug: ☐ Low level ☐ Fake VME

Figure 1: *The configuration window for the l1test program*

module to the crate. New crates can also be added to existing racks, and new racks to the whole system configuration. Options are always present to delete an object or add a new one. Modules can be repositioned within a crate and rack sizes can be changed. Finally, any altered configuration can be saved for future use.

Below the configuration picture is a row of options. For instance, the option **Read/write forever** uses the `l1 forever` command to repeat the next VME operation in a tight loop for "scope" tests.

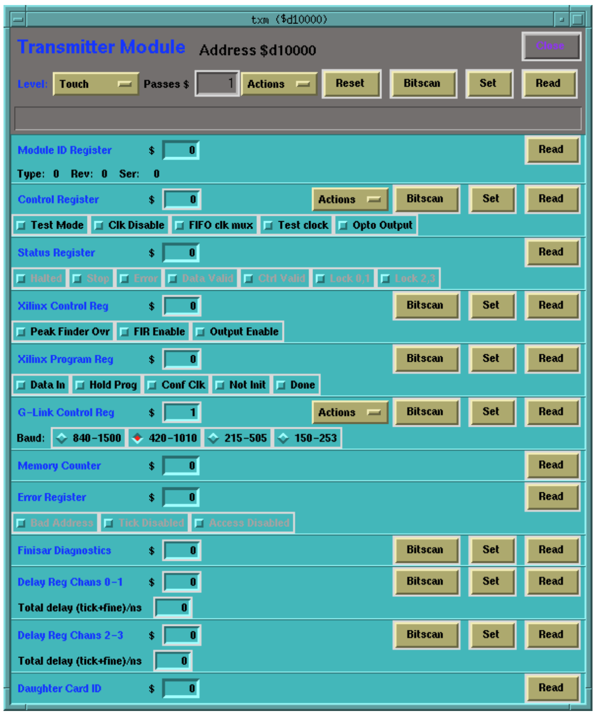
4.1 Register and Memory Windows

Register windows can be opened by choosing the `Registers` option from the pop-up menus for the Cluster Processor Module (CPM), Timing Control Module (TCM), Transmitter Module (TXM) and Flash Analogue-to-Digital Converter Module (FADC). An example of a TXM Register window is shown in Fig. 2. It shows the type of module and its VME address in the title line followed by a row of global command buttons which act on **all** the registers displayed in the rest of the window. There is also a display of individual registers, each of which can be read, written and tested individually. This display makes it easy to control and interpret the individual bits in the registers.

Similar detailed windows are available for the TXM, CPM and FADC memories. The Memory windows allow whole memories to be read, filled and tested with various patterns. The Tcl/Tk interface makes it relatively easy to include additional panels to perform operations in logically separate components of the module, such as individual ASICs. For example, on the Level-1 CPM module there are 15 ASICs of a particular type, each having the same internal register and memory structure. Each ASIC can be fully manipulated by one window which behaves in the same manner for each ASIC.

4.2 Autotest Windows

Autotest windows are opened by choosing the `Autotest` option from the pop-up menu for CPM, TCM and TXM modules. This is designed to enable the user to perform a series of pre-packaged tests on



```
txm ($d10000)
Transmitter Module   Address $d10000                                  Close
Level: Touch   Passes $ 1   Actions   Reset   Bitscan   Set   Read

Module ID Register        $ 0                                          Read
Type: 0   Rev: 0   Ser: 0
Control Register          $ 0          Actions   Bitscan   Set   Read
☐ Test Mode  ☐ Clk Disable  ☐ FIFO clk mux  ☐ Test clock  ☐ Opto Output
Status Register           $ 0                                          Read
☐ Halted  ☐ Stop  ☐ Error  ☐ Data Valid  ☐ Ctrl Valid  ☐ Lock 0,1  ☐ Lock 2,3
Xilinx Control Reg        $ 0                     Bitscan   Set   Read
☐ Peak Finder Ovr  ☐ FIR Enable  ☐ Output Enable
Xilinx Program Reg        $ 0                     Bitscan   Set   Read
☐ Data In  ☐ Hold Prog  ☐ Conf Clk  ☐ Not Init  ☐ Done
G-Link Control Reg        $ 1          Actions   Bitscan   Set   Read
Baud: ◇ 840-1500  ◆ 420-1010  ◇ 215-505  ◇ 150-253
Memory Counter            $ 0                                          Read
Error Register            $ 0                                          Read
☐ Bad Address  ☐ Tick Disabled  ☐ Access Disabled
Finisar Diagnostics       $ 0                     Bitscan   Set   Read
Delay Reg Chans 0-1       $ 0                     Bitscan   Set   Read
Total delay (tick+fine)/ns  0
Delay Reg Chans 2-3       $ 0                     Bitscan   Set   Read
Total delay (tick+fine)/ns  0
Daughter Card ID          $ 0                                          Read
```

Figure 2: *The TXM Register window*

the particular module in question (e.g. "soak" tests). These tests typically make use of the simulation of the module to compare its actual and expected behaviour. A similar scheme has been implemented to test the links and data flow between two separate modules. It is foreseen that this scheme will be extended to perform more elaborate system tests by tracking data flow through a whole chain of modules.

5 Class Library

In the following sections the main parts of the class library are presented in diagramatic form. For clarity these have been shown in separate diagrams. Two broad areas, covering the core classes and the simulation classes, are briefly described below.

5.1 Primitive Classes

The class hierarchy diagram for "primitive" classes is shown in Fig. 3. The base class for most of the important classes is the `UserIO` class. This is a pure virtual class which defines an object that can be addressed by the user interface. It has a name, an internal class type, and methods which provide the connection to Tcl/Tk. The next few levels, `Subcomponent`, `Component`, `XComponent` and their immediate subclasses, implement a "containment" or "configuration" hierarchy. A diagram of a simple configuration is shown in Fig. 4. The configuration hierarchy is quite distinct, almost orthogonal, to the class hierarchy. The idea is that most elements of the trigger system may contain subcomponents and may themselves be subcomponent of larger elements. For example, crates may contain a variable number of modules; modules consist of a fixed set of registers and memories.

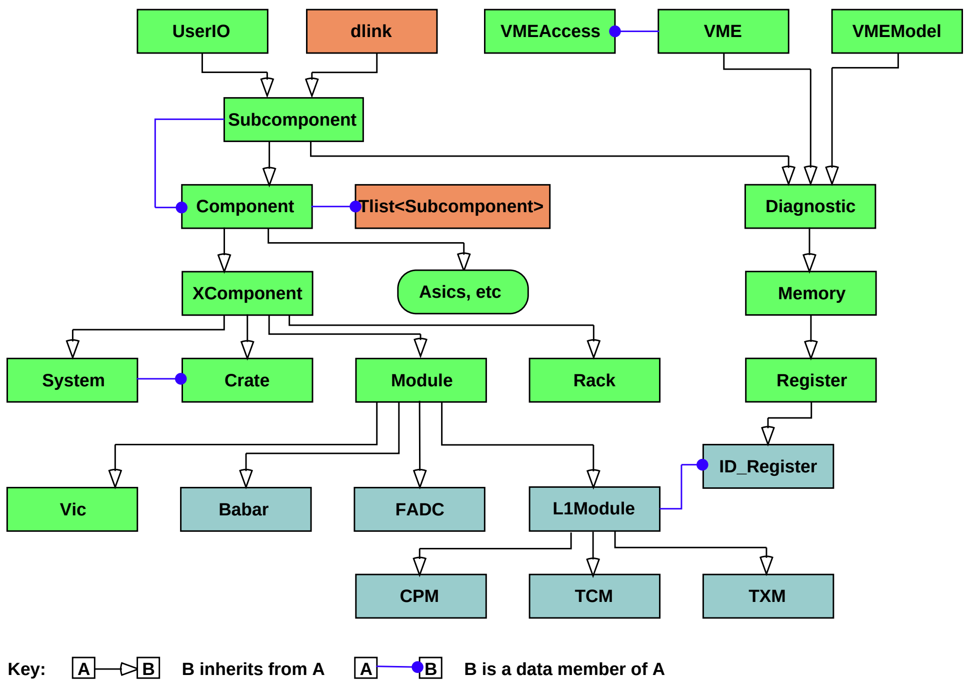


Figure 3: *The class hierarchy diagram for "primitive" class*

The `Subcomponent` class is at the bottom of the configuration hierarchy. It can only be a subcomponent of another object, i.e. their "container". In contrast, the `Component` class can additionally contain `Subcomponents` of its own. `XComponent` objects are `Components` that can be drawn on the screen (using X windows) as part of the configuration diagram. Their state can also be saved to and initialised from a configuration file, whereas other subclasses of `Component` or `Subcomponent` are expected to be in fixed configurations, such as `Registers` in a `Module`.

`XComponent` subclasses, such as `System`, `Rack`, `Crate` and `Module` are the basic elements in the configuration. They can be dynamically created from the Tcl/Tk interface using the `l1obj` command. Among their data members are their fill colour and text colour; their x and y dimensions and their x and y positions within their container, e.g. slot number in a crate or height (in U) within a rack.

Most `XComponent` subclasses have additional functionality: the `System` class knows about the "system crate", i.e the `Crate` in which the Lynx-OS CPU resides; `Crate` objects know about `Vic` modules which handle the VME address mapping between crates; `Modules` know their own VME base address, etc.

`VMEAccess` is a stand-alone class which implements the actual access to VME by the system. This is (almost) the only class which needs to contain any system dependency; there is also a dummy version for systems with no real access to VME but which may be used for code development. In the latter case the I/O to "VME" is always faked by I/O to a small internal array.

There is a single global instance of `VMEAccess` which is referenced by the `VME` class. `VME` is the base class for hardware access to a VME memory or register. It has a base address, and via `VMEAccess` it can Read and Write to Long or Word offsets from that address. `VMEModel` is the basis for the simulation of a VME register or memory. The Reads and Writes are done to an internal data member.

The functionality of both `VME` and `VMEModel` are combined in the `Diagnostic` class which is the key to the testing strategy of the diagnostic software. It is the base class of the `Register` and `Memory`

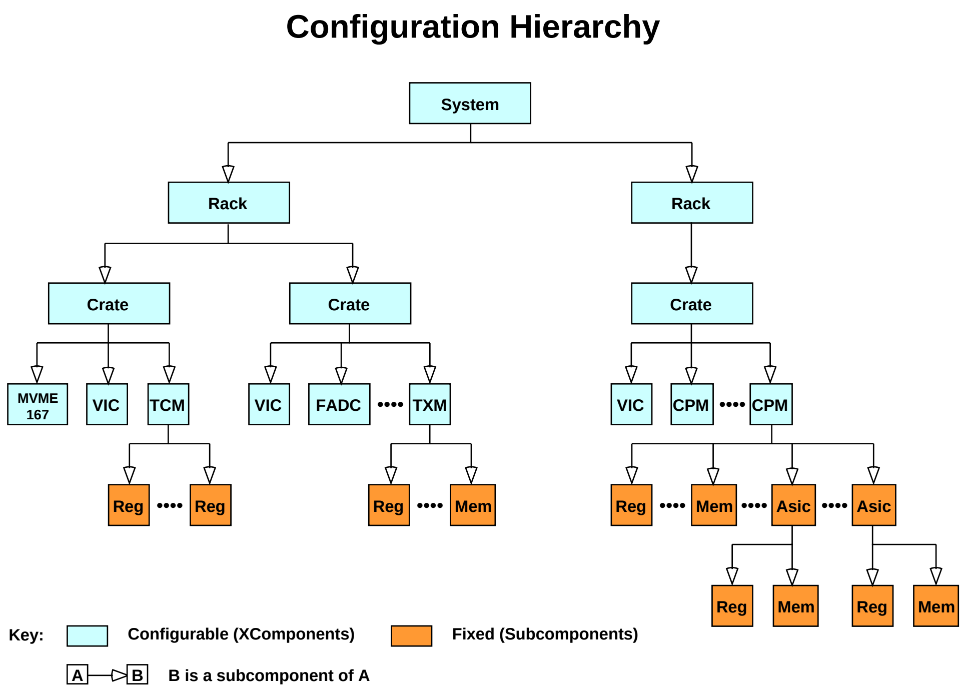


Figure 4: *A simple configuration*

objects that comprise most `Module` subclasses. Any Read or Write operations are performed to both the real hardware (`VME` class) and to the software model. Comparing the states of the hardware and model after an arbitrary sequence of operations allows the correct operation of the hardware to be checked in detail.

5.2 Simulation Classes

The simulation class library is a subset of the level 1 diagnostic software that provides a mechanism for modelling the behaviour of the hardware. The hardware description language VHDL has provided many of the concepts and terminology used in the simulation class. A diagram of the simulation class hierarchy is shown in Fig. 5. There are four main classes in the simulation class library: `Signal, Port, Entity and Simulator`. The `Signal` class objects are used to pass information between the entities in the model. `Signal` objects usually belong to an `Entity`. The `Port` class objects enable an entity to read and drive signals which are external to it; objects of `Entity` class are functional units that represent a digital system. A model will typically be built from a number of entities which are connected together by signals via ports. Finally, a global object of the `Simulator` class is in charge of the whole simulation process.

The simulation class provides a library of digital components (e.g. simple gates, memories, latches etc.) which can be connected to each other to simulate a new component capable of performing more complex tasks. To illustrate how the simulation works an example follows. A register in a module is a subclass of `Entity`. Writing a value to it drives signals from its ports to ports on other entities in the module. The simulator propagates changes until a new steady state is reached.

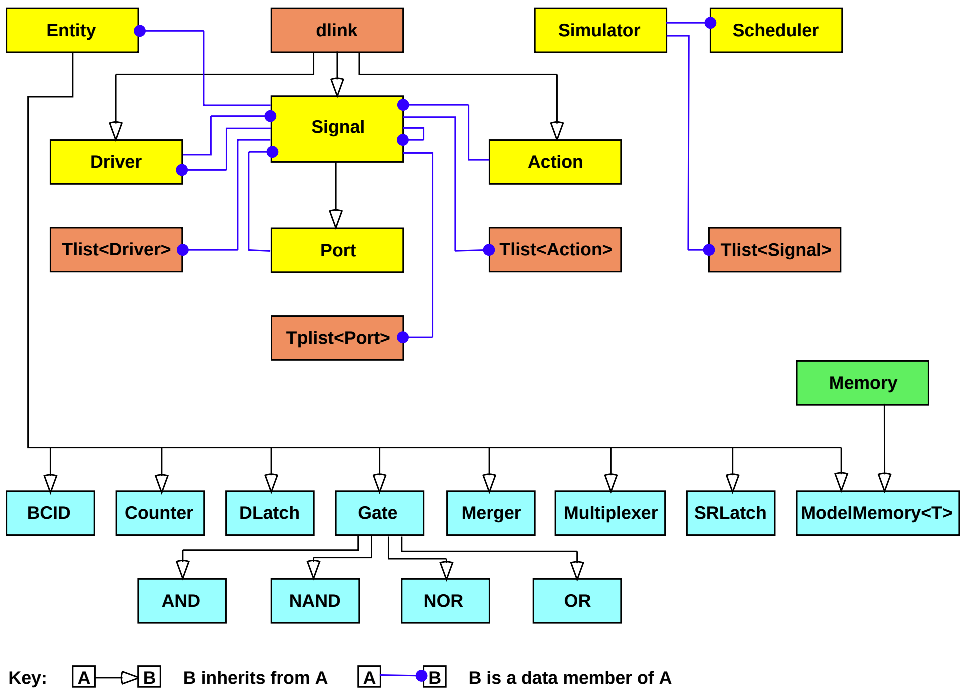


Figure 5: *The simulation class hierarchy*

6 Conclusion

The diagnostic software developed for the testing and debugging of VME modules of the ATLAS Level-1 Trigger processor has been described. A graphical user interface has been developed using the Tcl/Tk tool and the underlying software has been written in C++ using object oriented methods. This software package has been already extensively used and is being further developed in response to user requirements. It has proved to be flexible and user-friendly tool, easy to use for software non-experts for debugging and testing a variety of modules.