ATLAS Level-1 Calorimeter Trigger 10 April 2001
Software Note 006 Version Draft 0.2

HDMC Changes

Murrough Landon

1 Introduction

This note sets out some suggestions for improvements to HDMC [1]. These are mainly focused on the use of HDMC as a hardware access layer in the DAQ environment rather than as a standalone program. Some of the necessary changes, eg A32 access to VME, are also required for standalone operation. We will probably also wish to develop a number of new Parts describing components of our new modules. These will also be of use in both environments.

The main areas addressed in the subsequent sections are:

- HDMC library organisation
- Changes to the basic Part and related classes
- Modules and composite components
- Registers
- VME access
- GUI developments
- New Parts
- Integration with the Online software
- Parts file syntax

2 HDMC Library Organisation

At present HDMC is organised as a single package. Until recently the various binaries were all linked with explicit lists of `.o` files they needed. The object files are now also linked into two separate shared libraries: one for the hardware parts and one for the GUI classes. However these are not used by all the HDMC related binaries.

In future developments it will be useful to have Parts which use classes from other packages, eg the Online software, other L1Calo packages etc. It will probably be more convenient if Parts which depend on external software were not included in the core HDMC library. On exception to this would be if we moved HDMC to use the Online software database instead of its private data files.

Using the `regbuild` program it is possible to generate the code for specific Register subclasses automatically. These would naturally have to be compiled into separate libraries.

HDMC is currently built using GNU make. ATLAS is moving towards a new software "configuration management" tool, CMT. This is intended to ease the process of producing coordinated releases of related software packages. At some point it will be desirable to move HDMC (and other L1Calo software packages) into this scheme. Especially when we develop Parts with mutual dependencies on other ATLAS or L1Calo software.

In summary, we should consider a more formal reorganisation of HDMC libraries and the package itself into a number of layers. There should be a core layer which can be standalone and other layers using the HDMC core but which also depend on other software packages. Some discussion is required to decide the most appropriate organisation.

3 Parts and the PartManager

At present Parts are always created either by the GUI or by the PartManager reading a parts file. These two components each take care of satisfying the dependencies of each Part. All Parts are managed by the Part manager in a single hierarchy. The PartManager can merge Parts from several parts files, but they can only be saved back to a single file.

In some cases it would be useful if Parts read from separately merged files could be saved back to their original files. Also, at present, merging of parts files is normally done interactively in the GUI. It can also be done by program, but it would be useful if the part file syntax allowed for other files to be merged automatically.

It is possible to create Parts directly with `new`, giving their dependencies explicitly in the constructor. In this way, Parts can contain other Parts as data members, with the component Parts being initialised by the parent Part constructor. However in this case, the component Parts are not made known to the PartManager and so do not, at present, appear in the GUI.

Changing this would require that a Part knows whether it is managed and has a link to its own PartManager. This was originally the case in HDMC, but this link was removed to provide a cleaner separation between the APIs of the Part and PartManager classes.

Letting the Part know about its PartManager would allow a Part to add its data member Parts to the PartManager. However since these Parts are intrinsic and automatic components of their parent Part, they should not be saved to the parts file.

The PartManager would therefore have to know (a) which of its managed Parts to save, and (b) which file (if any) they originated from and in which file (if any) they should be stored.

The Part registry and the dependency mechanism uses text strings to specify which other Parts or Attributes are required by a given Part. The mechanism to initialise Part dependencies requires no code changes in other code when a new Part is added which is clearly convenient. On the other hand it uses casting to evade C++ strong typing which is undesirable. Using the standard C++ run time type identification (RTTI) might be more robust and should allow some of the static infrastructure to be removed from the Part class. Older systems such as LynxOS are now accessed via the `busserver` so we should be able to rely on RTTI support from newer compilers on other systems.

4 Modules and Composite Components

The HDMC Module class is conceived as a means of providing VME Bus access to its component Register, Memory and other Parts. Named collections of Parts exist as Module instances and it is possible to duplicate these Part trees. It is possible to keep the collection of Parts comprising one type of module in a separate file which can be read and merged when a new instance of the Module is required. But in both cases, the two Modules can subsequently be edited independently – ie two instances of a CPM can diverge.

It would be preferable if the description of a Module could be identified as a single entity in the saved parts file, so that any changes to the definition of a given type of Module would be automatically propogated to all instances of that Module.

One way of doing that has already been discussed in section 3. This is to keep the descriptions in Parts files and modify the PartManager so that it keeps them distinct. Another approach, also alluded to in section 3, would be to define subclasses of Module for the different types of Module and include their component

Parts as data members. A variation of this would define these data members as pointers to Parts which would be read from a parts file. The advantage of this is that it wouldnt require changes to the Part class, but the drawback is that C++ code and the parts file must be kept in step.

The latter approaches appear preferable from the DAQ point of view as they would allow the Module subclasses to provide additional methods accessing known data members to perform useful functions specific to a given type of module.

All of our modules contain complex subcomponents, eg ASICs and FPGAs which themselves contain collections of simpler objects such as registers and memories. In general, these subcomponents will appear replicated at several offsets within the Modules address space. Although they could be defined as Modules in their own right, it seems more appropriate to define a new SubComponent (or SubModule) Part. As a subclass of Module it could be the parent of ModuleRegister and ModuleMemory objects while still being seen as a component Part of a Module.

Some modules however may have programming models which do not group registers in the most convenient way for the software to handle. In this case it is useful to be able to define alternative groupings of registers etc.

The parts file syntax has been extended to allow this. However this scheme involves merging separate parts files and hence suffers from the problem that the merged files can only be saved into a single file, losing the structure the original syntax defined.

Note also that the existing grouping scheme assumes that all Parts are defined in parts files. It would not work (presumably?) if the Parts to be regrouped were created as intrinsic data members of their parent Parts.

5 Registers

The HDMC Register class (and its GUI) is by far the most complex of all HDMC Parts. This is due to the large amount of flexibility in defining dynamic register formats which allow the meaning and size of some bit fields to vary depending on the values found in other bit fields.

While the end result may be very desirable, the complexity means that the code is hard to understand and hence difficult to maintain. This can act as a brake on other developments if they have any repercussions on the Register class.

For example, the `regbuild` program can convert simple register formats defined in the configuration file to Register subclasses. This allows simple access to the bit fields from other C++ code (as opposed in interactively in the GUI). However the present implementation does not handle the more complex register formats.

So far the `regbuild` generated classes have not been used extensively – not everyone is keen on this approach. But if it turns out that they are useful, the program will need extending to cope with all possible register definitions.

6 VME Access

At present HDMC only implements A24 access to VME. The various data widths are represented as separate address classes such as AddressD8, AddressD16, AddressD32.

It is not clear if these are sufficient to define whether A24 or A32 access is expected. In any case, none of the VMEBus subclasses yet provide A32 access – at least not under Linux. Some redesign of the Bus and Address classes may be required or be desirable. If we need to make significant changes to support A32, we should also consider support for VME64 operations.

The initialisation of AddressD16 and Address32 objects uses offsets respectively in 16 and 32 bit words. Many people have expressed the view that this is error prone and would prefer if all addresses in HDMC were expressed in byte offsets.

Changing the existing default would have serious implications as all stored parts files would have to be changed. The change could be made as part of a major new release of HDMC with an accompanying change in the format version flag of the parts file syntax.

7 GUI Improvements

Most of this note addresses HDMC issues connected with its use as a hardware access layer in non-interactive, DAQ type software. Changes to the GUI are not directly relevant to that, but are still important for diagnostic purposes which is HDMCs main strength at the moment. It is also important to maintain the separation between the hardware Parts and their graphical interfaces.

The ModuleView GUI was intended to provide a more compact view of Modules consisting of many registers. This is useful as the screen can quickly become cluttered with many separate Register GUIs.

Unfortunately the present implementation of ModuleView suffers from a number of bugs and other deficiencies. The most glaring of these is that there appears to be an incompatiblity between the (supported) Qt tab widget and the (unsupported) flow layout. The flow layout provides the most compact display of a single register when the number of additional bit field widgets it requires is not known in advance.

If the ModuleView is to be used seriously this problem must be corrected. This may require abandoning the flow layout and using a fixed layout. This is likely to result in a less efficient use of the available screen space.

A number of other improvements to ModuleView are also highly desirable. At the moment it only displays Registers. It should also show Memory and other component Parts of a Module.

If we define Module subclasses to implement additional methods for specific types of module, it will be useful if these were also accessible via the HDMC GUI. This will presumably require corresponding subclasses of the ModuleView.

8 New Parts

Many new Parts, usually with accompanying GUIs, will be needed for our new modules and their components. This note does not attempt to describe them, but a list of the main ones may include:

- FPGA code: there is already a class for loading FPGAs. It may require extension and certainly new subclasses for loading different FPGAs via the various programming models of different modules.
- Firmware components: eg the Serialiser FGPA, CP "chip", JEM Input FPGA, JEM Main FPGA, etc. These all need description in terms of their subcomponents. While this can be entirely done via parts files, a customised class and GUI may be more useful.
- Playback memories: subclassing of the Memory classes to add specific decoding or packing of playback data into deviously organised memories.
- FIFO: ie a memory which is accessed via pointer and reset registers. Unlike directly VME mapped memories, read and write operations may affect the contents and must be used with care. This is mainly an issue for the GUI.
- TTCrx: description of its internal registers. Again access may be different on different modules (though we hope not).

9 Integration with Online Software

Most of the above has assumed that HDMC, or some of its classes, will either be used standalone or as a hardware access layer in other L1Calo software packages which may be based on components of the ATLAS Online Software.

Is there any additional need to integrate HDMC itself into the Online Software? An argument for not doing so is that the HDMC GUI could most usefully remain as a standalone development and debugging tool. This model sees HDMC as a core package on which others depend, rather than having HDMC depend on other packages.

One possible exception, and the main area where integration might be useful is if we decide to radically change the syntax of either the parts file or the configuration file. These are addressed in the next section. Even so, using the Online database component, OKS, as the basis for the HDMC data files need not introduce dependencies on the rest of the Online software.

10 Parts File Syntax

HDMC reads two types of data file: the parts file and the configuration file. Both of these are based on the same underlying syntax which is specific to HDMC.

The configuration file is used to define Register formats. It is normally edited and updated by hand. The parts file is maintained automatically by HDMC itself and contains the list of Parts created by the user with the GUI together with connections between them and any customised views of collections of Parts.

It has been a long term intention, as yet unimplemented, to replace the private syntax by something based on XML. This could however have very radical repercussions throughout a large body of HDMC code, in particular the Register class which depends heavily on the configuration file. Although the syntax is in principle hidden behind the ConfigStore and BitMapper classes, the API may not provide as clean a separation as may be desirable.

One way to improve this might be to convert the configuration file, after reading it, into a set of objects each of which would represent the one register format. This could be an extension of the approach taken internally by the `regbuild` program.

If the parts file and/or configuration file syntax is changed, it may be sensible to use the Online Software OKS component instead of XML directly. OKS implements an object oriented database, saving its files in XML.

It is clear that any change to the HDMC file syntax could involve a lot of work. If it is undertaken, many other improvements, eg changes to the parts file discussed above, could usefully be made at the same time. Note though, that all the parts file and configuration file use the same basic syntax at the moment, they are held separately in HDMC and the syntax of one could be changed before the other.

References

[1] HDMC home page
`http://wwwasic.kip.uni-heidelberg.de/atlas/projects/hdmc.html`