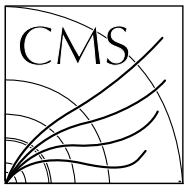
Available on CMS information server

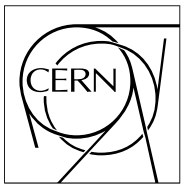
CMS NOTE 2004/014



The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



September 15, 2003

A Software Package for the Configuration of Hardware Devices following a Generic Model

N.Almeida, R.Alemany, J.C. da Silva, J.Varela[1]

LIP,Lisbon, Portugal

F.Glege

CERN, Geneva, Switzerland

Abstract

This paper describes a software package developed in C++ under the Linux environment that is intended for automatic hardware configuration in VME or PCI buses. Based on a generic model, users specify the configuration procedures and data in configuration files. Actual hardware configuration is performed by the software package, accessed through a simple C++ interface. The model is well suited for storage of configuration data in XML files or databases. The package is now being used in the local data acquisition system of the Electromagnetic Calorimeter of the CMS experiment at CERN.

[1] Also at CERN, Geneva, Switzerland

1 Introduction

Electronic device configuration is one of the main tasks needed when developing software to control hardware devices. This is a low level task, requiring often manipulation of bit fields in hardware registers. Usually the configuration procedures are highly dependent on specific hardware and are frequently hard-coded in the user application. In order to avoid this, we have developed a software package in C++, under the Linux environment, which allows automatic configuration of hardware devices. This software uses a generic configuration model to be followed by users when describing their device configuration. Configuration data can be retrieved directly using Extensible Markup Language (XML) [1] files or databases (ORACLE and MySQL) driven by the Dstore application of XDAQ [2, 3].

The developed package, named Generic Configurator, is divided in two sub-packages: the Item Builder and the Configurator. The item builder allows the construction of data structures associated to a given hardware register, while the configurator is responsible for setting the configuration data in the registers bit fields. This note starts by describing the software dependencies of the generic configurator (Section 2). The item builder is presented in Section 3. The configuration model and the generic configuration procedure are described in Section 4 and in Section 5, respectively. Finally, the conclusions are drawn in Section 6.

The description of this software is done using the Unified Modeling Language. A detailed application program interface (API), documentation, examples and code are available[2].

2 Software Dependencies

The Generic Configurator package uses the Hardware Access Library (HAL)[3]. HAL consists on a high level interface allowing bus independent access to VME or PCI modules from user applications. The hardware registers to be accessed must be defined in an address table (Table1). Presently HAL supports bus adapters for PCI, the PCI-VME interface from National Instruments and the SBS PCI-VME interface.

Table 1: Example of two entries of VME items in the HAL address table.

Item	Address Modifier	Width [Bytes]	Address	Access Mask
CHANNELS	09	4	00000000	FFFFFFFF
WEIGHTS	09	4	00000FFF	FFFFFFFF

3 Item Builder

The Item Builder allows the association of a hardware register, called item, to a data structure. These structures must be defined in a XML file, the Item Builder Structure file. Figure 1 shows the relationship among classes in the item builder package. There are two basic types of items: single and composite ones. Single items are associated with only one data field, while composite items manage contiguous memory regions. Users can specify if the memory region associated with the item is to be transferred into/from the hardware as a set of single bus accesses (default) or using block transfer operations. The access address can be set to be incremented automatically upon access (default) or to remain constant (FIFO access). A priority attribute can be specified. This is used when users want to build a list of ordered items. Memory regions can be defined as a set of words forming a contiguous block of bit segments (see example in Figure 2) or as a set of words, each one defining a set of bit segments (see example in Figure 3). In Figure 2 a composite item named CHANNELS is defined with a block of two words divided in single bit segments, each bit representing a channel state. The size of each word is the size defined in the HAL address table for the corresponding item (in this case 4 bytes, see also Table 1). In Figure 3 a composite item named WEIGHTS is shown with a block of 1000 words, each word with two 12-bit segments, representing two different weights (WEIGHTA and WEIGHTB).

---

2 http://cmsdoc.cern.ch/~nalmeida/start/gconfig.htm

3 http://cmsdoc.cern.ch/~cschwick/software/documentation/HAL

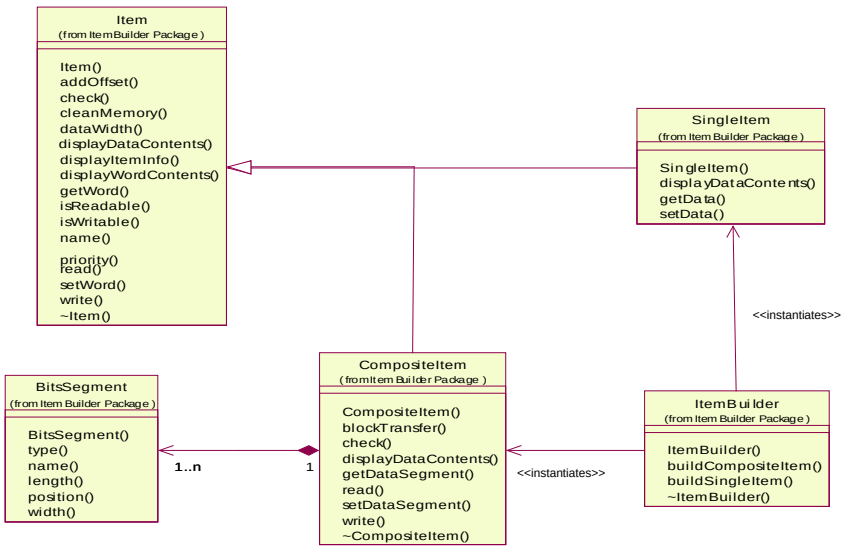


Figure 1: Class diagram of the Item Builder package.

After instantiation of an item object, the object is responsible for memory management allowing setting or getting any defined bit field. As an example consider a composite item as shown in Figure 3:

```
CompositeItem * weights = itemBuilder->buildCompositeItem("WEIGHTS");
```

If we want to set all weights A to value1 and all weights B to value2 we write:

```
for(ulong position = 0; position<1000; position++) {
      weights->setDataSegment("WEIGHTA", position, value1);
      weights->setDataSegment("WEIGHTB", position, value2);
}
```

The data content of the item can be transferred into/from the hardware with the methods write/read (in this case as block transfers):

```
weights->write();
```

Users may inspect all bit segments defined in the item through the method `displayDataContents()`, or use the method `displayWordContents()` to see all the memory contents regardless the bit segments definition.

```
<CompositeItem name="CHANNELS" >
  <ContinuousBlock length="2">
    <BitsSegment name="CHANNELSTATE" width="1" />
  </ContinuosBlock>
</CompositeItem>
```

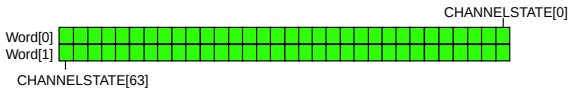


Figure 2: Composite item named CHANNELS with a contiguous block of single bits representing channel states. The left side shows the XML definition of the item in the Item Builder Structure file. The right figure gives a graphical representation of the declared memory region.

```
<CompositeItem  name="WEIGHTS"  blockTransfer=”on” priority=”2”>
 <SegmentedBlock length="1000">
<BitsSegment name="WEIGHTA" width="12" position="4"  />
<BitsSegment name="WEIGHTB" width="12" position="16"/>
 </SegmentedBlock>
</CompositeItem>
```

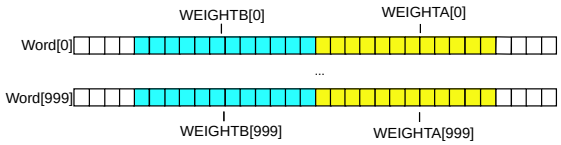


Figure 3: Composite item named WEIGHTS with a segmented block of two bits segment (WEIGHTA and WEIGHTB). The left side shows the XML definition. The right side figure gives a graphical representation of the memory region.

4 Configuration Model

Configuration data are classified as single or segmented configurables. Single configurables have a single data field that is associated with the data contents of a given single item (defined in Section 3). Segmented configurables are bit fields of a given composite item (defined in Section 3). In order to group configurables we introduce the Structure entity. Modules are specialization of structures with a given associated address (see Figure 4). A hardware device is represented as a module. The main module may contain sub-modules. When configuring items of a given sub-module their address offset will be determined according to the modules to which they belong.

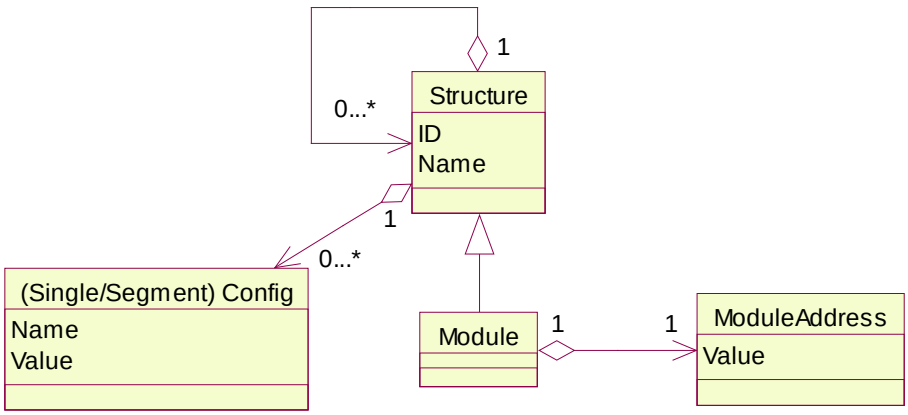


Figure 4: UML conceptual representation of the configuration model.

The model allows users to configure devices either from XML files or databases. In both cases the input of the Generic Configurator is a Document Object Model (DOM) [4] tree representing the device configuration. In order to minimize database storage space, it was chosen to represent modules and structures in terms of database tables and the module address, single and segmented configurables as columns of those tables. Concerning XML

files, each element is characterized by a `_userInfo` attribute. For modules and structures the tag name with its ID must also be specified as an attribute that is related to the table key in the database. For single configurable elements the tag name must be equal to the associated item name. For segmented configurable elements the tag name corresponds to the segment name and the respective index in which the data must be written. Figure 5 shows an example of a simple configuration with one main module, two sub-modules and one structure.

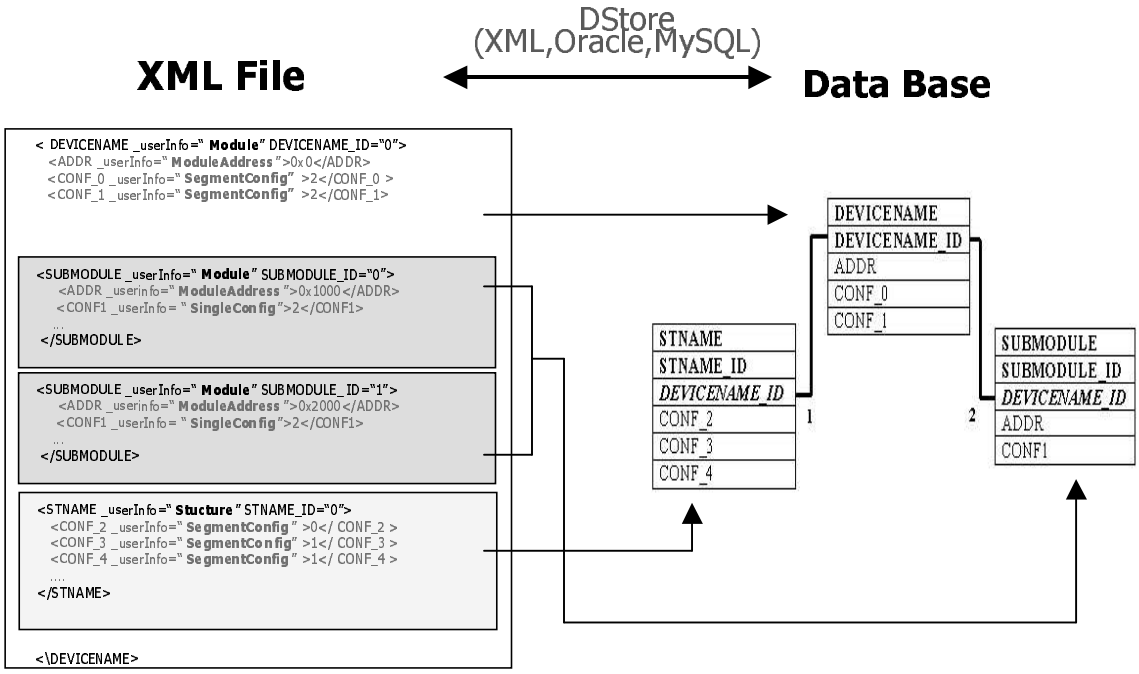


Figure 5: Example of a configuration XML file and its representation in terms of a database scheme. The Dstore application allows independent access to XML files or databases (ORACLE and MySQL).

5 Configuration Procedure

In order to configure hardware modules, user applications must interface with the Device Configurator (see Figure 6). When requested to load a new configuration, the Device Configurator triggers the parsing of the configuration DOM tree as shown in the sequence diagram in Figure 7. Whenever a new module is found the base addresses of the main module and associated sub-modules are calculated.

When configurable elements are found in a module or a structure, the configurator tries to find out if the item with the offset given by the tree level has already been instantiated. If there is no such item a new item is created by the item builder. Items are added to the container list according to their priority, allowing the definition of priority dependent configuration procedures.

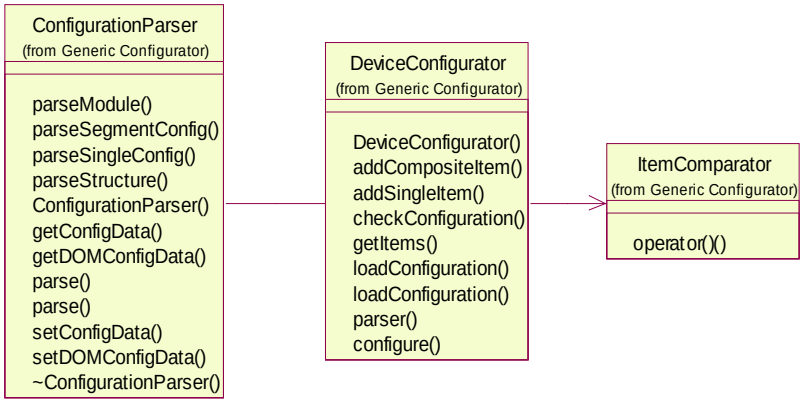


Figure 6: Class diagram of the Configurator package.

After the data has been copied into the item memory, the data are ready to be transferred to the hardware. This is done through the `configure()` method. The `checkConfiguration()` method allows to read-back data from hardware registers for validation purposes. In case of error(s) diagnostic information is given to the user. The configuration data are kept in memory allowing an optimized reconfiguration or configuration check at any time.

In the user application, after having built a new Device Configurator object, the configuration procedure is executed by the following code:

```
myDeviceConfigurator->loadConfiguration(myDOMTree);
myDeviceConfigurator->configure();
```

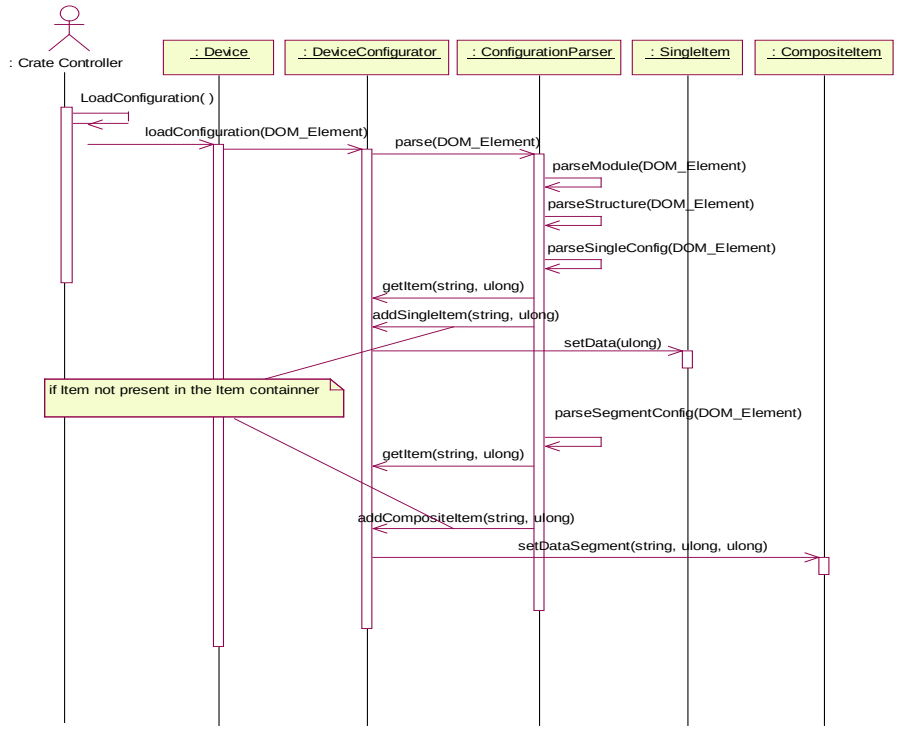


Figure 7: Sequence diagram representing the loading of a new configuration. In this case a Crate Controller makes a `loadConfiguration()` request to a Device object that is delegated to the Device Configurator.

6 Conclusions

A new package has been described that allows automatic device configuration. Three inputs are needed: the proprieties of hardware registers to be accessed (HAL address table), the definition of the data structure associated to each hardware address (Item Builder Structure) and, finally, a DOM tree with the configuration data. The package is being successfully used to configure complex electronic devices, like the Electromagnetic Calorimeter Data Concentrator Card in CMS, which has more than 11000 configurable parameters.

References

[1] J.Boyer, XML Version 1.0, W3C recommendation, 15 march 2001: http://www.w3.org/.

[2] Proceedings of the International Conference on Computing in High Energy and Nuclear Physics, CHEP 2001, Beijing, China, September 3-7, 2001, Ed.H.S.Chen, pp.601-605, Science Press, ISBN 1-880132-77-xx, J.Gutleber et al., *"Clustered Data Acquisition for the CMS Experiment"*.

[3] CMS CR-2003/007, V.Brigljevic et al., *"Using XDAQ in Application Scenarios of the CMS Experiment"*.

[4] DOM Level 3 Core Specification Version 1.0, A.L.Hors et al., W3C Working Draft 09 Jun3 2003: http://www.w3.org/.