

Summary of the Unpacking and Packing Software Review

Jim Kowalkowski
Marc Paterno

1 Introduction

This document summarizes the results of the review of the DØ raw data packing and unpacking packages, **unpack_evt** and **unpack_reco**. Accompanying this text is a class diagram illustrating the modified design we recommend for this system. In reviewing this design, we were guided by the following use cases, which we understand to be the most significant:

- Unpacking in Level 3;
- Packing of simulated events, for use in tests of Level 3 code;
- Packing of simulated events, for efficient storage.

Because of the close relationship of this package to the Level 3 code, some of our suggestions concern modifications of a small number of classes in the package **l3base**. Furthermore, because of the close relationship between some of these packages to the various detector simulation packages, we have some comments on how those packages can beneficially use what is presented here.

We have given the most weight to issues of efficiency (both in time and in storage space) and to issues of maintainability, since it is very likely that the implementers of early code will not be available late in Run II. Loss of such expertise in Run I caused significant difficulty in code maintenance.

2 Summary

We found the high-level architecture and analysis of the requirements to be very well thought through and the problem to be well understood by the developers. The concept of having a view of the raw data available to users in a manner independent of the details of the implementation of the hardware within a system, crate, module or channel is highly desirable. Isolating the definition of how a channel is packed within a module and reducing coupling to this function is a valuable design property. The use of adapters to give various views of channels is a good concept. The use of a database to determine the crate configuration for a given run dynamically is an extremely value tools. Our recommendations focus on how to gain the most benefit from each of these design ideas.

3 Major Concerns

3.1 *MCHModule and MCHSystem*

Instances of the classes *MCHModule* and *MCHSystem* contain temporary event data during the process of packing or unpacking. These objects also contain information describing the configurations of systems and crates within the systems and the allowed modules within the crates. The lifetime of instances of these classes is generally a complete run or longer. While the information of the configuration of the system of modules during a run is clearly information critical to the unpacking or packing of data, we would prefer to see this information separated from the knowledge of how to do packing and unpacking. We propose the creation of a class *RunConfiguration*, which describes the configuration of crates and modules in a given run. Instances of *MCHSystem* can then be configured (and re-configured if necessary, as is the case when multiple runs are processed in a single execution of a program) using an instance of *RunConfiguration*.

Use of *RunConfiguration* would also help to decouple this part of the system from the stopgap use of RCPs to configure the *MCHSystem*. In the first version of this code, a *RunConfiguration* object can be instantiated by data given to it in an RCP object. In a later version, a more robust solution would be for the *RunConfiguration* object to be recovered from some appropriate database.

We also propose modifications below that would remove the temporary storage of event data from the *MCHSystem* and *MCHModule* classes. We think this makes use of these classes simpler to understand, which is important for maintenance.

3.2 *UnpDataChunk*

We believe that the amorphous nature of the class *UnpDataChunk* allows (and perhaps even encourages) developers to side-step the use of adapters; instead, they may directly manipulate the array of integers within a channel. Our concerns were reinforced when we looked at some of existing code that creates an instance of *UnpDataChunk*:

```
for(int cont=0; cont<=11; cont++){
    int modid = (cont << 4)|15;
    MCHID mid(sysid,modid,0);
    int version = 2;
    std::vector<int> data;
    data.push_back(0);           // trigger number
    data.push_back(cont+64);    // crate number
    data.push_back(12);        // ADC card count
    data.push_back(0);         // BLS mode
    data.push_back(15);        // ADC mode
    data.push_back(1);         // data source, 1 = MC
    data.push_back(0);         // data type, 0 = collisions
    data.push_back(0);         // error status
    Channel chan(mid,version,data);
    chvec.push_back(chan);
}                               // done looping over crate controllers
if(count>0){                   // store only if there are any channels
    UnpDataChunk* unptr = new UnpDataChunk(sysid,chvec);
    auto_ptr<UnpDataChunk> autochunk(unptr);
}
```

This code segment shows several calculations and manipulations that should be implemented within the packing and unpacking package rather than here.

- The knowledge of how to calculate the module ID is replicated here. If the method of packing the module ID is changed, it would be necessary to find all such snippets of code. The same can be said of the calculation of the crate number.
- The offsets within the vector of integers are used directly, rather than through an adapter. If a design change requires the reordering of items in this array, it will be very difficult to find all places in the code where changes would be needed.

In addition, this code uses the classes *vector<int>* and *Channel* inefficiently. We think that the design of *UnpDataChunk* will lead to other instances of such coding. We have included a few notes in the following section regarding a more efficient method for the use of *vector*.

4 Coding Recommendations

4.1 Use of Vectors

1. If the number of elements to be used in the vector is known at the time an instance is created, then it is best to create the vector with the correct size. This prevents the reallocation and copying that can

accompany the resizing of a vector. In this case, `vector::push_back()` should not be used, since that will put the added elements *after* the intended end of the vector.

2. An alternate possibility is to create the vector with the default constructor, and to use the function `vector::reserve()` to allocate enough room to store the required data. In this case, `vector::push_back()` can be used to fill the vector.
3. Vectors should not be cast to C style arrays. A C array should be used if the additional features of the vector class (such as fact that the class vector provides for copying and assignment). The C array can be more efficient, because there is no memory overhead with an array, unlike with a vector. If a vector is needed, then the iterator class `vector::const_iterator` or `vector::iterator` should be used for iteration; the vector should not be cast to an array, which is not portable.
4. Initializing a vector which is a class data member using an assignment statement from a temporary variable is inefficient. One should initialize such a data member in the colon-initialization list in the class constructor.

4.2 Operators

Unless there is special reason to do otherwise, define comparison functions such as `operator<()`, `operator>()` and `operator==()`, and the assignment function `operator=()` as member functions rather than as global functions. This is because these operators often require access to private data. One reason to make the functions global functions rather than member functions is if it necessary to have the compiler perform automatic type conversion on the left-hand operand. Since this is never done for the object on which a member function is being called, this can require the creation of a global function.

4.3 Space Allocation

Be very careful with the use of `new` and `delete` for allocating arrays, both of basic types and of objects. When creation of an array is wanted, be sure to use `"new unsigned int [array_size]"` and not `"new unsigned int(initial_value)"`. The latter allocates one unsigned int and initializes it to value `initial_value`, whereas the former allocates an array of unsigned ints of length `array_size`. Be sure to know at all times who owns the memory and when it should be resized and deleted. Also, be sure to use `delete()` to deallocate the memory allocated by `new()`, and `delete[]()` to deallocate the memory allocated by `new[]()`.

5 Design Recommendations

5.1 Overview

In producing the following design recommendations, we were concerned mostly with the issues of long term code maintenance and of efficiency, both in terms of speed and of space. We view maintainability of the code as a very important goal, and have stressed breaking up the functionality of the classes into more cohesive pieces to enhance maintainability.

We have divided the problem into three areas:

- classes which hold and manipulate packed raw data (classes related to Level 3),
- higher-level unpacked data classes,

classes which perform packing and unpacking and which describe the organization of the system.

5.1.1 Level 3 classes and utilities

We strongly recommend the addition of iteration methods to the class `l3base::crate`, to provide for iteration over the modules within the crate. The current implementation requires that users of `l3base::crate` have complete knowledge of the data format, and it forces users to invent their own method of iterating through the modules, which is prone to error. This should be done by the introduction of an iterator class, which

when dereferenced returns a reference to a module, and for which operator++() is defined. The class *crate* should have begin() and end() member functions which return such iterators.

We recommend that the package **I3base** take on the responsibility of providing a class that provides direct access to a given module inside a crate. This functionality is currently held with the classes *MCHCrate* and *VRBModule* within the package **unpack_reco**.

We recommend that *RawDataChunk* have the ability to be built using a flattening or serializing technique. This could involve having a pool of buffers large enough to hold the largest packed crate in a system, then unrolling the modules into the buffer one after another -- directly from the unpacked data.

5.1.2 MCH classes

We strongly recommend that the *MCHxxx* classes be used as a mini-database or repository for configuration information. In addition, these classes could contain the methods (tools) necessary to perform the packing and unpacking. These classes should contain no event related data or temporary results while packing and unpacking.

It was clear that, in the design of *MCHSystem*, considerable thought had been given to the time spent in creating objects with new() for every event. We recommend use of the class template *PoolManaged<T>* at the appropriate places in order to make allocation and deallocation of objects inexpensive. The design of *MCHSystem* can then be simplified, making it unnecessary for *MCHSystem* to retain as much state information as it does in the current design. Since *PoolManaged<T>* provides the needed functionality without requiring the user to use any special functions, it is possible to add this inheritance at a later date, when it is most convenient, or when profiling demonstrates that the need is critical.

5.1.2.1 MCHModule breakup

We highly recommend that specific hardware channel classes be created that the *MCHModule* class will use pack/unpack a module. The *MCHModule* should control the pack/unpack of a module and describe a configured module within a system.

5.1.3 UnpDataChunk

As noted on the accompanying diagram, the subclasses of *UnpDataChunk* may not be needed. If they are not provided, then *UnpDataChunk* (or, better yet, an external function that manipulates *UnpDataChunk*) would have to provide iteration over modules of a specific type within the *UnpDataChunk*. Furthermore, this iterator will have to use a dynamic_cast in order to return the correct flavor of module pointer.

5.1.3.1 Channel Reorganization

The *UnpDataChunk* channel poses many problems.

The *UnpDataChunk* channel is too amorphous for users to employ directly in a good design, but as we saw above, this does not mean that it will not be used. We recommend the introduction of concrete channel classes, which provide the appropriate interface and data tailored for the detector elements involved.

5.1.4 Databases

The RCP database, while having the advantage of availability, is not really appropriate for use in configuring systems for runs. It is our recommendation that the infrastructure group be contacted to find out what the status is of a real run configuration database.

The accompanying class diagram shows several classes associated with the functioning of such a database. *RunConfig* is somewhat similar in concept to the class RCP, except that it is specific to the needs of describing a run configuration. *ConfigManager* provides the programmatic view of the database. It could be a Singleton class, that provides access to instances of *RunConfig* in a controlled fashion. *ConfigManager* should also be responsible for issuing identifiers for *RunConfig* objects, so that it is possible for an

UnpDataChunk to have a record of which *RunConfig* was used to create it. This is what the class *EnvID* was intended for.

5.2 Descriptions of Classes

Unless otherwise specified, all classes are defined in the *mch* namespace. The headings below show the various subsystems; these are also indicated on the accompanying class diagram.

5.2.1 Level 3

Classes in this subsystem have short (one event) lifetimes, and carry or provides views of event data.

l3base::RawDataChunk: This class is the holder of the raw data. It provides for iteration over the crates within the raw data.

l3base::L3Crate: This class provides a view of the data in a *RawDataChunk*. It provides for iteration over the modules in the crate.

l3base::CrateInfo: This class provides another view of a crate. It provides random access to modules, looked up by an (int) module identifier. It may not be necessary, since *l3base::l3crate* provides meaningful access to crate data in the proposed design.

VRBModule: This class provided a temporary buffer in which event data was held during the packing process. It is not strictly necessary in the proposed design.

5.2.2 Database (DB)

The classes in this subsystem all have long (more than one event) lifetimes, and carry no event data.

ConfigManager: This Singleton class provides access to *RunConfig* objects, and is responsible for matching an identifying tag with a particular instance of *RunConfig*.

RunConfig: Each instance of this class describes the configuration of modules and crates in one run – either a collider data run, or a simulated event sample. *RunConfig* objects are used to provide the description of a run configuration to *MCHSystem*, which actually does packing and unpacking. It provides access to descriptions of both “systems” and crates.

ModuleDesc: This class is the description of a single type of module. Each instance carries the type of and the estimated channel count for a specific module type. Any other information common to describing all types of modules should also go here.

SystemInst: This class represents a single “system”, such as the CPS, Muon or Calorimeter. It carries the description of the modules within the system, as well as the identities of the crates that need to be accessed in order to unpack (or pack) the module data.

CrateInst: This class represents a single crate, and carries the description of each module in the crate.

5.2.3 Packing and Unpacking (MCH)

The packer/unpacker classes in this subsystem have long (greater than one event) lifetime, and do not carry event data. These classes manipulate channel classes which do contain event data.

MCHSystem: This class provides the interface for both packing and unpacking. An *MCHSystem* is created using a *RunConfig* object that describes the layout of crates and modules appropriate for the given task of packing or unpacking.

MCHModulePU: This is the base class for all module packer/unpacker classes. Each subclass must implement the four member functions *pack()*, *unpack()*, *wordCount()* and *chanCount()*. Each *MCHModulePU* contains a reference to a description of the appropriate kind of module.

MCHCratePU: This class contains the knowledge necessary to unpack a single crate.

CalModulePU, *RawCalChannel*, *RawCalHeaderChannel*: These classes are an example of a the set of concrete classes that need to be written for each system. They would not be part of the **unpack_evt** or **unpack_reco** packages. The illustration shows several concrete channel classes, created while packing or unpacking calorimeter data. Each class has a buffer, which is a pointer into a *RawDataChunk*; the pointer indicates the location from which reading will occur during unpacking, or to which writing will occur during packing.

5.2.4 Unpacking

Classes in this system have short (one event) lifetimes, and carry event data.

UnpDataChunk: this is the base class for all concrete unpacked data classes. It provides for iteration over all modules within the chunk, but in the interface of *UnpDataChunk*, what is returned is a pointer to the base class *Module*. Most users will want to use the interface of a concrete data chunk instead.

CalSystem is provided as an example of a concrete subclass of *UnpDataChunk*. It provides an interface that allows iteration over modules, but of the concrete subclass *CalModule*, rather than the base class *Module*. Also shown are several concrete subclasses of *Channel*; it is expected that each system will define the appropriate channel classes.

CalChannelAdapter: This class is an example of an adapter class that can be used to given another view of a concrete channel. In most cases, the adapter class holds no event data; it merely provides new methods to access the data of the concrete channel class, perhaps by performing some simple (or not so simple) manipulations before returning the data to the user.

5.2.5 Packing

Classes in this subsystem have long (more than one event) lifetimes, and do not carry event data. Some of the classes do provide views of and access to event data, held in another object.

RawDataBuilder: Instances of this class are used to produce instances of *RawDataChunk*. A *RawDataBuilder* is configured using a *RunConfig* object. *RawDataBuilder* incrementally adds systems to a *RawDataChunk*, as those systems are added through the `addData()` function.

CrateData: This class provides a view of the crates within a *RawDataChunk*. Why?

ModuleData: This class provides a view of the modules within a *RawDataChunk*. Why?