The UCESB unpacker generator
Long write-up — documentation and manual
© HÅKAN T. JOHANSSON, 2010
Department of Fundamental Physics
Chalmers University of Technology

Abstract

A tool to get quick access to nuclear physics experimental data on-line and off-line. The system can help to unpack event-wise experimental data from e.g. nuclear physics experiments, and provide easy access to the various data members, in form of hbook ntuple files, root trees, or plain C structures in any program via the network, but also in user functions or for quick viewing from the command line. It can also create plots showing the correlation between data occurring in different detector channels, sometimes helpful to quickly verify that cabling schemes are correct.

This is achieved by defining a descriptive parseable language that is used to express the structure of the binary data. Data structures and related unpacking methods are generated at the same time from the description. The language is suitable to define the meta-data placed around the (possibly modified) module data stream in order to group events, assign trigger and/or timestamp information, separated from the actual channel data. Typical examples for different data specifications are shown in the write-up.

The system can in particular be used as an input data filter reading from a variety of different input streams and formats provided by different systems, and provide standardised output data streams for further analysis or simulation steps.

This work has in part been supported by the EU FP6, and by the the Royal Society of Arts and Sciences in Göteborg.

The extensive and patient testing by the LAND group at GSI is deeply appreciated, as well as the suggestions for the program and this write-up.

Keywords:
Data Acquisition, Analysis, Quick-n-Dirty

Contents

Chapter 1

Introduction

Information recorded by a data acquisition system (DAQ) must be transformed before being easily used with a data analysis tool (like `paw` [1] or `root` [2]). This write-up describes and is a manual for a tool to help taking that step. The creation of the processing stage to unpack data from the raw data files produced by a DAQ, ordered by read-out module, into something that is suitable for processing and rapid monitoring and visualisation, ordered by detector channel, is a repetitive and rather boring task. It does contain a multitude of interesting aspects, mostly in the realm of computing however, but is not what one would like to spend time on every time a few modules or detectors are to be tested.

This tool is named `ucesb` after the working principle it adheres to: "unpack and check every single bit".

1.1 Overview

A `ucesb`-based unpacker is a tool to do rather "quick and dirty" checks of event-based data produced in e.g. nuclear physics experiments. This is done by providing the necessary pieces and infrastructure to perform the recurring tasks:

- A generic input stage for file reading, including on-the-fly decompression of data files as well as reading the data over the network from a tape-robot via remote file I/O (`RFIO`) or directly using `tcp/ip` from a running DAQ.

- Extraction of the event-wise data from a few record/buffer-based file formats (and `tcp` transports).

- Simple data structures suitable for fast cleaning between events and random order filling with in-order extraction, while retaining zero-suppression[1].

- A "language" to specify the layout of data produced by the DAQ read-out of modules. A parser and program to generate `C++` code based on the specification, that can unpack the data into suitably generated data structures. The generated unpacker performs rigorous checking of the data integrity versus the specification given. In cases when the data format for one reason or another is too complicated to be specified in the provided language, hand-written unpacker functions can be seamlessly integrated.

- Along with the raw data format specifications, a mapping describing how the raw data correspond to physical detector channels can be given. From this, data structures arranged by detectors are generated. The program performs the data mapping for each event.

- Ability to dump selected parts of the data (from the internal data structures) as `ntuple`s into `hbook` or `root` files, usable by e.g. `paw` or `root`, or plain `C` structures over the network.

- A few modes of simple but effective on-line data monitoring.

The user can thus concentrate on the work of dealing with the data from the module or detector of interest itself.

1.2 Intended audience / skills needed

What to expect? The average user (physicist?) is surely able to use the command line options. The greatest desire would probably be to as quickly as possible generate `paw` ntuples or `root` files.

Describing a simple data format, to unpack data from some new module, and to specify an accompanying data mapping, is a little more intricate but still straight-forward and only requires some extra will and discipline[2].

[1]With few exceptions, the data structures are designed to allow the program to operate such that the event processing time is proportional to the non-zero data size in each event, and not the worst case event size (when all channels contain data), which usually is orders of magnitude larger.

[2]The sometimes thought of as nasty, very rigorous, checking of the data integrity is actually quite helpful here.

Implementing some quick data analysis, averaging, or other consistency checks within the user function, should also be fairly easy, but may at first seem daunting. The difficulty is that it requires knowledge of how the various members of the data item will be named, and how to handle the container data structures for zero-suppressed arrays. This path would most likely be treaded by a DAQ-close person, with the `paw/root` idea appealing more to others.

A more complicated task is to construct the unpacker functions for more involved data formats — which usually is the equivalent of saying that the data formats sports one or more levels of explicit or implicit zero suppression where the output can be anything between small and sparse up to rather dense. Handling this efficiently requires quite some care.

Using the monitoring facilities provided from the command line is quite easy. Adapting them to new kinds of data will by necessity involve diving into the usage of the classes and functions performing the wanted tasks.

In addition to this documentation, the program source comes with over a dozen unpackers, handling data from various smaller or larger experiments. They span the entire range from doing nothing at all with the data, to completely furnishing their own data structures, see Appendix A.

1.3 Quick start

Seeing is believing!

1. Obtain the sources, by cloning the GIT repository:

   `git clone http://fy.chalmers.se/~f96hajo/ucesb/ucesb.git`

2. Build all example unpackers:

   `make`

   or just one of them (this example handling data from the S245 experiment):

   `make land`

   One common build problem is that the default parser generator `yacc` does not handle the extensions used in the syntax descriptions — edit the first lines in the `Makefile` by removing the hash comment markers (`#`) from `#YACC=bison` and `#export YACC` to explicitly use `GNU bison` instead.

   For some of the examples below to work, support for `cernlib` [3] or `root` [2] and `ncurses`[3] is needed — recompile with:

```
make clean
make land USE_EXT_WRITER=1 USE_CURSES=1
```

3. Let the program process a data file by running:

   ```
   land/land /misc/scratch.land1/s245/lmd/r06_0308.lmd
   ```

   Obviously, access to some appropriate input file is needed.

4. To create an ntuple (requires `cernlib` or `root`), add:

   `--ntuple=RAW,r06_0308.ntu` or `--ntuple=RAW,r06_0308.root`

   to the command line. The ntuple will include all `raw` level variables.

5. With `ncurses` support, let's go to the movies!

   ```
   --watcher=POS*T:POS*E,N1_1-5_?T
   ```

6. Further options are listed by:

   ```
   land/land --help
   ```

7. Some data from a test setup is available online,

   `http://fy.chalmers.se/~f96hajo/ucesb/mwpc_test1.gz`

   It can also be used for testing:

   ```
   make mwpclab USE_CERNLIB=1 USE_CURSES=1
   mwpclab/mwpclab mwpc_test1.gz
   mwpclab/mwpclab mwpc_test1.gz --ntuple=UNPACK,mwpc.ntu
   mwpclab/mwpclab mwpc_test1.gz --watcher
   ```

[3]Generally available with UNIX-like operating systems.

Chapter 2

Data levels

The work done by the unpacker program creates a bridge between data representations suitable for long term storage in a file, and a representation suitable for manipulations in the memory of a computer. Actually, much of the computing efforts in analysing nuclear physics experiments is dealing with similar kinds of conversion problems — between data representations suitable for long term storage (archiving and documentation[1]) and the calculations that one needs to perform.

2.1 Data flow

Figure 2.1 gives an overview of how data flows within a `ucesb` unpacker. Except for the input stage and associated buffering, the content of each event is sequentially filled into data structures modelling various layers of the setup — in a sense going backwards from the electronics to the physics of ions traversing detectors.

The processing of each event begins with the **unpacker** functions looping over all subevents of the event and calling the appropriate member functions to interpret the data in the input buffer. The unpack functions will place the data found in the `unpack` level data structure, which has a layout closely tied to the data arrangement in the file, usually representing various digital acquisition modules, DAMs.

The **map** functions loop over the non-zero data in the `unpack` structure and for items with a mapping defined, the values will be copied to the destination in the `raw` level. Before mapping is performed, an **unpack** user function (with access to the unpacked data) can be called. This is required

[1]On the use and power of text files and parsers within that realm, see [4], Section 4.2 and Appendices C and D.

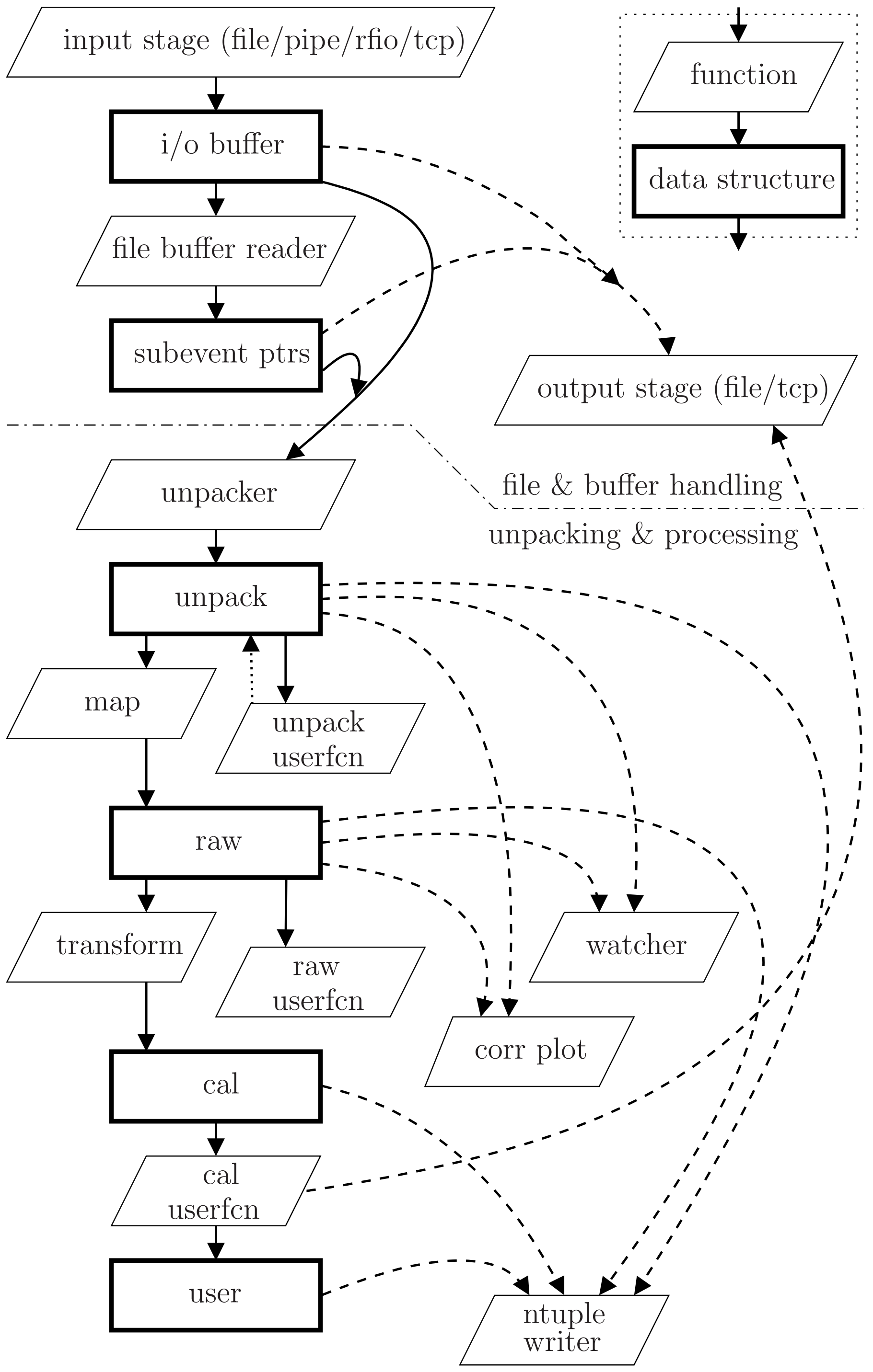


Figure 2.1: Data flow in a `ucesb` based unpacker. The dashed lines mark optional data paths enabled by various command line options.

in case the data has been collected in multi-event mode, i.e. such that several physical events are stored in each data-file event. In such instances, the `unpack` level data structures will hold one DAQ event, while the `raw` and subsequent structures are filled and processed once per physical event.

The remaining two data levels (`cal` and `user`) are not expected to be generally useful without specialisation — they would make the first steps of a data-analysis proper. Being too generic, one quickly finds them not providing any particular advantage for but the simplest kinds of analysis.

The **transform** routine converts the integer data in the `raw` level data structure into floating point values stored in the `cal` level, with linear transformations using parameters given in one or more calibration files.

If present, the **cal** (final) **user function** is called. It has access to all previous data levels, and can be used for arbitrary processing of the data. A `user` level data structure may be declared, and filled from the cal user function. Certain limitations apply to the layout of the user data structure, as its members are eligible for use by the ntuple writer and therefore must be parsable by the `psdc` helper program.

After all data levels have been filled, the monitoring and event dumping routines (e.g. ntuple writing) are executed.

2.2 Data structures

The programs are written in `C++` and make heavy use of both member functions with special names in each data item and structure to perform particular tasks, and template parameters to create arrays, etc., with appropriate sizes of containers and accompanying bit-masks. Templates are also used to overcome the usual unpacking plague of byte-swapping. The recurring idea is to let the compiler do the work of creating optimised code[2].

The program's main task is data juggling — to and from each data structure, and in many cases with seemingly random or non-linear mappings, e.g. when detector signals have been connected with contrived channels orderings to the DAMs to avoid cross talk. Also the ntuple generation and monitoring tools need easy access to the various structure members. As those access patterns often are determined at runtime, depending on groups of channels selected on the command line, the actual accesses within the program can only be furnished via pointers, not by name.

The data structures look and behave as fixed `C` structures, much more than `C++` classes. Member functions are not used to access the individual

[2]Good old `C` preprocessor `#define`d macros are also a big help — die-hard object oriented total-encapsulation evangelists will not find themselves completely at home...

items — any routine having a set of pointers may help itself — provided it respects the additional information it has regarding how zero suppression is implemented before dereferencing the pointers. The filling of arrayed data structures have member functions to assist in doing the lazily performed[3] item clearing needed. Templates are used to define array sizes and item types at compile time.

2.2.1 Zero-suppression

Zero-suppression is a common and simple technique to avoid excessive memory access and processing by not storing or performing calculations on data from channels that produced no significant signals in an event, i.e. were noise-compatible. The idea is directly applicable when dealing with a set (array) of similar items of data, by simply leaving unused entries out.

The data structure holding such compacted data must be designed to efficiently assist all the algorithms that need to deal with the data:

Fast clearing Before each new event is processed, all data structures must be logically cleared. This is quick when only some markers that tell if data is available are reset, and not the data items themselves.

Avoiding memory management Trading memory for speed, much time is saved. By allocating memory enough for the worst case event (all channels having data) from the outset, or by only allowing the structures to grow, memory management is simplified.

Random order filling When either filling file input data into the `unpack` level, or mapping `unpack` module data into the `raw` level, the items usually do not appear in order. The data is filled into ordered structures by employing a bucket sort strategy. By handling the data array together with a bit-field marking valid locations, only the bit-field must be cleared between events.

Ordered extraction With the data structures being in order, extracting sorted data is trivial.

The program has two container structures for storing zero suppressed data (the prefix `raw` has become a misnomer):

[3]Lazy evaluation is to defer execution of a routine until necessary. In this case, individual zero-suppressed member items are not cleared until just before they are marked as used for a new event.

`raw_array_zero_suppress<T,n>` Stores `n` items of type `T` in fixed locations. A bit-mask is used to keep track of active slots. The index of each item is implicitly given by its slot. By default used for the `unpack` and `raw` levels, as those expect random fill order.

`raw_list_zero_suppress<T,n>` Similarly stores `n` item of type `T`, consecutively and each with an index. A counter keeps track of the number of used slots. Used for the `cal` level, as this expects to be filled in order from the `raw` level. (Items will be properly ordered even if given out of order, but performance will be lower, due to the insertion-sort behaviour.).

Both kinds of zero-suppressed containers only clear an item the first time (if at all) any member of it is filled each event. See Figure 2.2.

Multi-entry items

For read-out channels capable of delivering multiple hits for each event, a special version (`raw_list_ii_zero_suppress<T,n>`) of the list container that has no indices exists. The hit number index within each slot is in these cases usually meaningless, as it just gives the read-out order — multi-hit data are generally time signals, and as such ordered by themselves.

2.2.2 Multi-event support

Data collected in multi-event mode have the information of several physical events (triggers) stored within the same event of the data file. The reason for operation in multi-event mode is to reduce the overall dead-time for most events. Doing the data transfers coalesced in larger chunks, triggers are for most events only blocked during the analog-to-digital conversion in the modules. Each module stores data for many events and is read out fewer times. As the fast transfers (e.g. block transfers) usually do not differentiate between different events, the data for all the contained physical events will then be stored together in the subevent, followed by the next module. During unpacking and data mapping, this must be sorted out.

The `ucesb` based unpackers deal with this by allowing data items (e.g. a module) to be marked `multi`, whereby an arbitrary number of single-event instances are allowed to occur within each file event at the unpack level. After unpacking, the `unpack_user_function` is called, which is responsible to both tell how many physical events are contained in the file event, as well as assign each single-event item to one of the physical events. None of the supported file formats contain any explicit support for multi-event data, handling this is

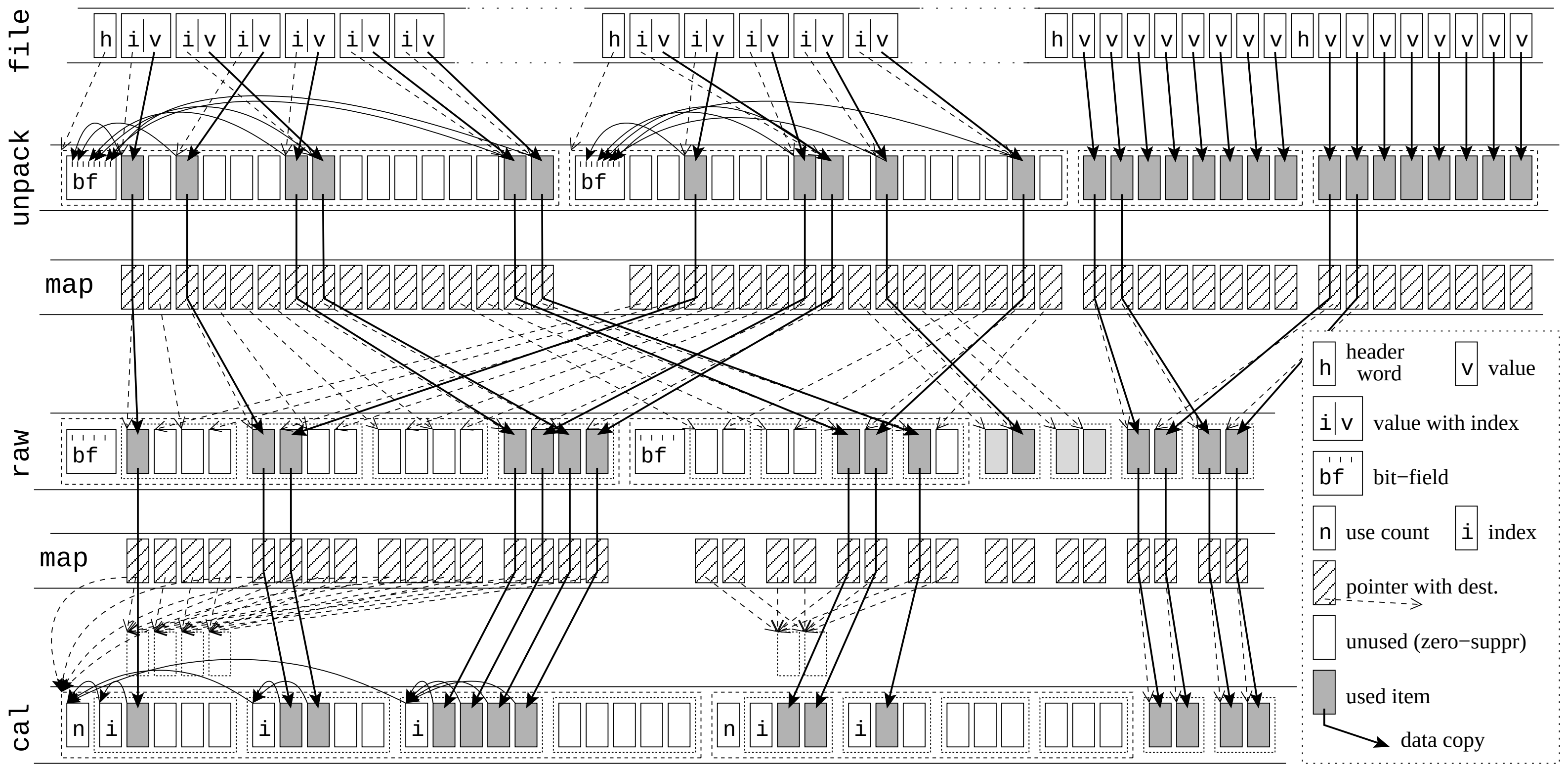


Figure 2.2: Data-item flow and book-keeping, from file to internal structures. The `unpack` level is filled by the unpacking functions. Data is then mediated through the `unpack` → `raw` and `raw` → `cal` pointer `maps`. The right part of the structures show fixed size entries, while the left part utilises zero-suppression, controlled by bit-fields at the `unpack` and `raw` levels, and counts and indices of valid items at the `cal` level. The data re-arrangement at the `unpack` → `raw` transition could be e.g. time and amplitude measurements for the same channels done in separate modules being put together.

thus up to the user. Modules supporting multi-event operation will generally place an event counter into e.g. the header word of each chunk of single-event data, thereby allowing omission of completely empty events.

The remaining loops over the data (mappings and user function calls), are then performed once per physical event. As special information contained within the file event may only apply to one single event, generally the last, this can be marked as such in the data mappings. (Using multi-event mode with e.g. the MBS [5], all but the last event in a multi-event series will not be able to invoke any operation from the DAQ software and can thus be implicitly taken as e.g. trigger 1. Any other trigger that needs some special handling already by the DAQ programs, e.g. some time calibration event, must terminate a multi-event series.)

2.3 File and event formats

The raw data produced by a DAQ are generally packaged in events, which in turn are stored in data files. This section describes the overall features of the formats the `ucesb` unpackers can be compiled to use. As the DAQ of any one particular experiment is using only one particular output data format, and the data format also affects some aspects of the unpacking, each unpacker can only be compiled to use one input data format at a time.

2.3.1 File buffering

Perhaps mostly seen as a relic from times when network speeds were prohibitively small and data had to be stored directly on tape, most data file formats specify that events should be packaged in some sort of records of lengths usually being a multiple of a large power of 2.

The fixed size buffers had the advantage of working better with the record-based tape storage formats, and still allows for a certain level of error recovery if a file is damaged[4]. A drawback is that the events generally never fit exactly into the records, and thus either space is wasted at the end of each record (on average, the size of half an event), or some events must be fragmented over several records, leading to inconveniences during unpacking.

[4]Also todays disk storage media and software is not completely fault-free, e.g. some S287 data files were somehow damaged during early storage, before being written to the GSI tape robot.

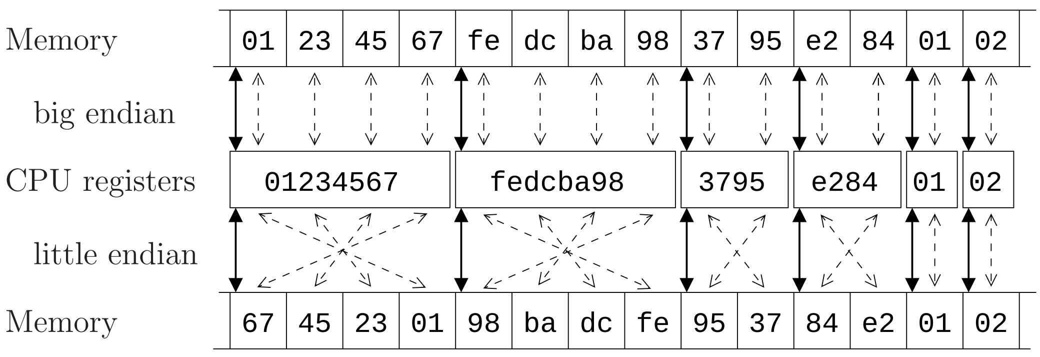


Figure 2.3: Big- and little-endian memory access conventions.

2.3.2 Endianess

The endianess of a processor architecture defines how it transfers multi-byte values between the CPU and memory. While the registers in the processor are generally just interpreted as one entity, the storage of any multi-byte content in memory (and by extension also on more permanent media) requires a convention, see Figure 2.3.

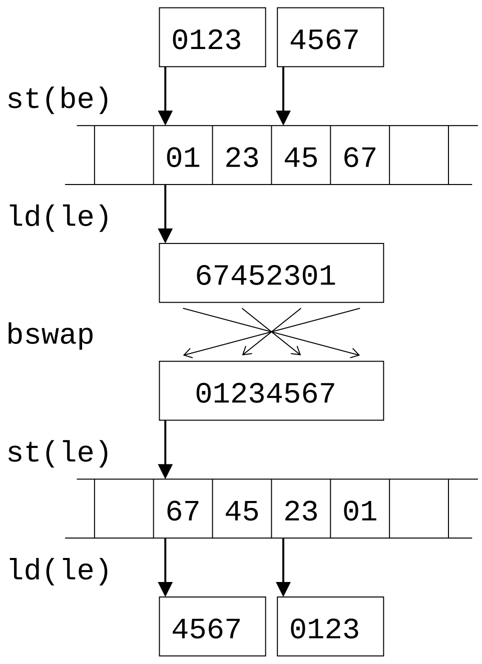
With the big-endian convention, the byte location containing the most significant bit is stored in the memory byte with the lowest address, i.e. the one selected by the pointer used to address the memory. This resembles the usual left-to-right writing of numbers, and is generally used by microprocessors with fixed-length (usually 32-bit) instruction words.

In the little-endian convention, the least significant byte is written at the first memory location, followed by increasingly more significant bytes. This has the advantage that the same memory pointer is able to point to the same object, either it is interpreted as being 8, 16 or 32 bits long. This is convenient for architectures using (or having their roots from) variable length instruction encodings, where immediate operands for the machine instructions often can be given in various sizes, depending on magnitude.

The `ucesb` format specification uses the little-endian convention of numbering bits in bit-fields from 0, starting at the least significant bit. (The big-endian convention usually start numbering the bits from 1 at the most significant bit.)

As the experimental data is collected using various computer architectures, and then often analysed using others, efficient handling of data conversion is paramount. Converting to and from the non-native endianess is easy, just flip the order of the bytes comprising each item — support for this is directly available in most CPUs since a decade. The trouble is thus not the speed impact of making e.g. an analysis program deal with byte swapping, but rather a book-keeping problem of knowing what and when to do it, i.e. remembering or recognising with what endianess the data were created.

In network protocols, this problem is usually dealt with by employing so called network order, which is the same as big-endian byte order. The data structures (i.e. the ordering of the various variables) are the same on all architectures. Each machine is responsible for converting every item before transmission and after reception, if needed. This is most often done by using the `hton*` and `ntoh*` macros[5], which translate into a copy on big-endian machines (frequently being completely optimised away), and one or a few simple instructions to perform the byte swapping on others.



`st` = store (CPU → memory)
`ld` = load (memory → CPU)
`be` = big endian
`le` = little endian

Figure 2.4: Scrambling.

Scrambling

There is one way to make matters worse. Assume some of the data is stored with 16-bit entries, but in the wrong byte-order. Now, swap the bytes of the data as 32-bit elements. Suddenly, the entries are all in the correct byte order, but also pair-wise interchanged, as in Figure 2.4.

2.4 File formats

For a detailed description of the formats, please consult the respective references/specifications. The following subsections mostly contain some comments on some aspects of the various formats. *They should not be taken as a suggestion to implement yet another format,* but rather as some remarks on pit-falls to avoid or at least to look out for and beware of.

[5]`hton*` is for host-to-network conversions and `ntoh*` is for network-to-host translations.

An important aspect of a data file format is to not be more "chatty" than necessary. Fields must be filled with correct information, as having wrong information is even more harmful. Likewise, event sizes *do* matter, not only due to the sheer storage size needed for files containing millions of events, but also as I/O capacity easily becomes more limiting than CPU processing speed.

2.4.1 LMD – list mode data (MBS)

The GSI list mode data files, LMD, are based on fixed-size buffers containing events. The events in turn contain subevents, one or more produced by each read-out controller in a multi-branch setup. The subevents may be distinguished by several identification fields, but unfortunately have no clearly specified way of conveying the endianess of the originating processor. This becomes a problem due to the rule of any receiver doing 32-bit aligned byte-swapping to native order at all stages in the data transport. At the same time, all lengths are stored as the number of 2-byte words, thereby if not expressly encouraging the use of compact 16-bit encodings[6], at least inviting to the creation of confusion at unpacking to deal with possible scrambling. . .

During data collection, the buffers are handled as several together in so-called streams. Events that do not fit into the space left in one buffer may be fragmented into the next, or over several buffers if necessary, but may not span multiple streams. Streams can not be spanned since they make up the quanta of formatted event data emitted by the event-building process (the collection and juggling of subevents from various subsystems into complete system-wide events).

2.4.2 EBYEDATA – Daresbury MIDAS

The EBYE[7] file format is record based [6], and the data is stored by MIDAS in events [7]. Each data item is 2x16 bits and byte order information both for the record header and the data itself is stored in the header. Even the data format as such is specified, with each entry using 32 bits, 16 bits for the value, and the remaining bits used to identify the source of the value, divided into groups and items. This strictness leads to undesirable contortions when e.g. storing 32-bit valued entries from scalers.

The data format is not really a raw data format, but has already been somewhat sorted by the DAQ. While simplifying small setups, it is quite

[6]Using dense encodings is a good thing.

[7]Possibly meaning event-by-event, no direct explanation has been found.

cumbersome to use the format generally, and forces the DAQ to do quite a lot of mapping of the data — which can only be considered to be a bad thing — being a recipe for mistakes and trouble while correcting mismaps afterwards off-line!

The detailed unpacking of the fixed data is not handled by the input stage of a `ucesb` program, but by a format specification, see e.g. the `madrid` unpacker (cf. Appendix A). Both this and the following PAX format have no subevents. To be able to use the generic unpacking code with minimum changes, a dummy subevent declaration is used.

2.4.3 PAX – KVI data

The PAX file format specifies a simple record and simple events with a length and a type. The headers and data are arranged in 16 bit units (thus limiting the length of the events[8]), and the byte order is specified in the record header. There are provisions for fragmented events, although the `ucesb` unpackers do currently not support them.

2.4.4 HLD – HADES raw data

The HLD file format [8] does not use any records, the events and subevents are stored directly into the file, only adhering to certain alignment rules. Alignment of the event, and each subevent within, is given by the event header. Byte order of the event and subevent headers is well described, as well as the word size of each subevent. The scrambling problem does not apply, as the rule is that no automatic byte-swapping is done during data transport — final reader makes it right for themselves. Each event is associated with a time stamp, but in pseudo-clear-text and it only gives second resolution, despite its use of 64 bits.

[8]One should remember that 2^{16} is 64k, which really is a lot of space for one event.

Chapter 3

Command line options

The unpacker program comes with a set of common command line options, controlling the input, output and error handling. Options not available due to various compile-time choices are shown in parentheses. The following terse descriptions, explained more in detail in this chapter, are from the program itself:

```
 file://SRC        Read from file SRC.
 rfio://HOST:SRC   Read from rfio file SRC from HOST.
 event://HOST      Read from event server HOST.
 stream://HOST     Read from stream server HOST.
 trans://HOST      Read from transport HOST.
 --scramble        Toggle scrambling of data.
 --in-tuple=LVL,DET,FILE  Read data from ROOT/STRUCT.
(--merge)          No support for overlapping sources compiled in.
 --calib=FILE      Extra input file with mapping/calibration parameters.
 --max-events=N    Limit number of events processed.
 --print-buffer    Print buffer headers.
 --print           Print event headers.
 --data            Print event data.
 --debug           Print events causing errors.
 --event-sizes     Show average sizes of events and subevents.
 --quiet           Suppress harmless problem reports.
 --io-error-fatal  Any I/O error is fatal.
 --allow-errors    Allow errors.
 --broken-files    Allow errors again after bad files.
 --help            Print this usage information and quit.
 --output=OPT,FILE  Save events in LMD file (native/net/big/little,size=nM).
 --bad-events=FILE  Save events with unpack errors in LMD file.
 --server=OPT      Data server (stream:port,trans:port,size=nM,hold).
 --show-members    Show members of all data structures.
 --show-calib      Show calibration parameters.
 --ntuple=LVL,DET,FILE  Dump data as PAW/ROOT ntuple.
 --watcher=DET     Do ncurses-based data viewing.
 --corr=TRIG,DET,FILE  Create 2D correlation plot.
 --dump=LVL        Text dump of data from data structures.
(--threads)        No threading support compiled in.
(--files-ahead)    No threading support compiled in.
 --progress        Do ncurses-based thread monitoring.
```

3.1 Input data

The unpacker will read and process all input sources in sequence.

`file://FILENAME` Read data from a normal file. This option also works without a `file://` prefix, i.e. any argument will as last (default) action be attempted as a file path and name. If a file is compressed using `gzip` or `bzip2`, a process will automatically be started to decompress the file on the fly.

`stdin` can also be used for input by specifying `file://-`, such that the unpacker can be used in a pipeline. Note that decompression will not be attempted in this case.

`rfio://HOST:PATH` Read data using the rfio protocol. Only tested so far with the GSI tape robot (`HOST=gstore`). To encourage good tape robot hygiene, i.e. staging the files before usage, the unpacker will only read files that are already staged.

`event|stream|trans://HOST:PORT` Read data from an MBS TCP server providing data with either of the event, stream or transport protocols. `HOST` is the Internet host-name of the server, the `PORT` is optional — without it, the default port for the given protocol is used. Both the stream and transport servers deliver full buffers. For the event server, buffers are internally simulated, to fit the normal unpacking scheme. There is (currently) no way to specify any filters which the event server could use to select events for transmission. This option is naturally only available when the program is compiled to use the MBS/LMD data packaging format.

`--in-tuple=OPT,LEVEL,DET,FILE` Read data from either a `root` file or a `struct` producer/server, see Sections 6.1 and 6.2. Implemented by reversing the `--ntuple` option and using the same `OPT`ions, see further down. The layout of the input file or structure must match what would have been generated as output.

All input that cannot be read using `mmap` is internally buffered, see Section 7.1.1. When compiled with thread support (cf. Section 5.3), the filling of the internal buffers is decoupled from the data processing.

`--scramble` All MBS data transport processes perform 32-bit (4 byte) byte-swapping on all data in order to remove the need for the user to have to consider endianess issues. However, as the MBS/LMD data format cannot store endianess information about the original data producing machine, any 8- or 16-bit data may have been scrambled within the 32-bit words, making the user have to worry about scrambling instead. This option can possibly help. As different subevents may or may not have been scrambled, more drastic (in program) actions may need to be taken.

`--merge=N` When the experimental data has been written to several files concurrently, each containing a fraction of the events, it may still be necessary (or at least simpler) to perform the analysis in strict event order. With this option, the unpacker will keep (at most) `N` subsequent input files open, and from each have one pending event located with headers. Events are processed in order from the different files (most simply determined by inspecting the event numbers[1]). The compile option `USE_MERGING` is needed.

`N` should be 2 times the number of event builders that were in use. This is necessary as when the event builders switch files, they are likely to do it at roughly the same time, and there is no guarantee that the file that ended first will be followed/continued by the next file number in the queue. Using a lookahead of the same number of files as event builders, there should be no problem.

3.2 Output raw and processed data

The unpacker can write raw data files and create ntuples of processed information.

`--output=OPT,FILE` When handling data in LMD format, the events can be written to an output file. Support for adding and omitting subevents is available. It is important to note that events with unpack or other processing faults will not be written to the output file. The behaviour can be adjusted with some comma-separated `OPT`ions:

`native,big,little` Writes the file in the specified endianess, default is `native`. While the MBS standard approach is to let any data receiver do byte-swapping of data received to native order (on 32-bit boundaries) before any processing, this program defers swapping until the handling of each data word. This approach simplifies

[1]This currently needs an MBS patch, as normally the different event builders use a local numbering.

the treatment of scrambled data files. As a side effect the writer has to be able to swap data (in case it e.g. is merging files with different endianess), thus it became straight-forward to make the output endianess selectable.

`compact` Trim the last output buffer.

`size=nnn(k|M|G)` Limits the size of each output file to (approximately) the given size. Automatic file numbering is employed.

`events=nnn(k|M)` Limits each output file to contain the given number of events.

`eventcut=nnn(k|M)` Whenever the event number is a multiple of the specified number `nnn`, a new file is opened. This is similar to the previous option, with a twist: each numbered file will always contain the same set of events, irrespective of any event selections (mainly by faults) that may differ between various versions of the unpacker.

`newnum` The automatic file numbering will skip numbers for files which already exist.

`wp` Write-protect each file after it has been written. Can be used to prevent accidental removal of recorded data when the unpacker program is used to act as a DAQ output stage.

`[incl|excl]=[name|tag=min[-max][:tag2=...]]` Sets of subevents can be selectively chosen for inclusion in (or exclusion from) the output file. The selection is based either on the subevent `name` as declared in the specification (see e.g. Listing 4.2) or one or more values or ranges of the subevent header `tags`: `type`, `subtype`, `procid`, `subcrate` and `control`. Entire events can be (de)selected using their `trigger=value|range`.

`skipempty` Do not emit empty events (after subevent exclusion).

`FILE` The output filename is given as the last option. If it contains a number (before a possible `.lmd` extension), this is used for the filenumbering counter. With a `.gz` or `.bz2` extension, the output data will be piped through an appropriate compressor. To use the unpacker in a pipeline, use `-` for the filename to write to `stdout`. In this case, no file size limits can be applied, and any normal program output (text, `stdout`) will be sent to `stderr`.

`--bad-events=OPT,FILE` When debugging the unpacker specification itself — or the DAQ — the most interesting events are those that do not work properly. With this option, any event which causes an exception during unpacking or processing is stored in `FILE`. The same additional modifiers as to `--output` can be applied, except selection of (sub)events.

This option is only useful for events that are somehow malformed. If the integrity of the data-file records themselves has been compromised, no event extraction is possible and unpacking can not be attempted.

`--server=OPT` When handling MBS/LMD data, the unpacker can also act as a data server for other analysis clients, i.e. the opposite end of `stream://` and `trans://`. The compile option `USE_PTHREAD` is needed, as the server runs in separate thread — see Section 7.2.1. In addition to the `native,big,little`, `incl=|excl=`, and `skipempty` modifiers described above, the following `OPT`ions are available:

`size=nnn(k|M|G)` Set the maximum amount of memory used to keep data buffers from the time when they become available until read by the last client.

`hold` Do not allow any client to miss any data. Use judiciously, since if some client is slow, that will set the pace at which data is processed.

`stream|stream:PORT|trans|trans:PORT` Specify the protocol(s) to use, and optionally use alternative port numbers. The default is to set up a stream server.

`flush=n` Flush the output buffers every `n` seconds.

`--ntuple=OPT,LEVEL,DET,FILE` The unpacker can write `hbook` (`cernlib`, as used by e.g. PAW) or ROOT ntuple files of the data[2], from the various data levels available within the program, cf. Section 2.1. One or more `LEVEL`s can be selected using a comma-separated list of the keywords `UNPACK`, `RAW`, `CAL`, and `USER`.

The data from the selected levels is chosen by the list of names of `DET`ectors requested. If no specific detectors are requested, all available data will be included. The available member names can be seen with the `--show-members` option. Items can also be selected from specific levels by prefixing the item by the level name: `LEVEL:DET`.

[2]Also with `root` files, only simple data layouts, corresponding to `hbook` column-wise ntuples, are currently generated. Creation of more advanced `root` trees would allow other zero-suppressing data-structure constructs to be used, grouping variables naturally, but is not yet implemented.

Each data member can be described with a "path" consisting of name fragments and indices, e.g. `N4_2_1T`, the time of the first PM tube of the second paddle in the fourth plane of LAND. When two successive elements in the path are of the same kind, name or index, they must be separated by an underscore. Ranges of indices can be specified using a dash, e.g. `N4_1-10`. If a digit or underscore is part of a textual name, it must be escaped by a backslash[3], otherwise it will be interpreted as the start of an index or a separator. Single or several parts of the path can be wildcarded using a question-mark or asterisk, e.g. `POS1?T` or `N*T`.

The file type to generate is determined by the presence of `.nt`[4] or `.root` in the output `FILE`'s name. Further `OPT`ions to adjust the output:

`UR,URC,URCUS` When similar items from several data levels are requested, the names may clash. With these options, the names of items from the `unpack`, `raw`, `cal`, and `user` levels can be prefixed by `U`, `R`, `C`, and `US`, respectively.

`RWN,CWN,ROOT` Force the output type as a row-wise or column-wise `hbook` ntuple or a `root` file.

`STRUCT` Instead of writing a file, a `tcp/ip` server can be started which will deliver the requested data, event-wise, to any (number of) stand-alone program(s), see Section 6.2. With this option, a dummy `FILE` name is still needed but ignored — except if specified as `-` to request delivery of the protocol data directly to `stdout`.

`PORT=n` Used in conjunction with `STRUCT`, to use an alternative `PORT`.

`STRUCT_HH` Create the `C` header necessary to handle data received when using the `STRUCT` option. The header is stored as `FILE`.

`UPPER,LOWER,H2ROOT` Data member names by default use the type case they have in the data structures. They can be forced to either upper or lower case, or mimicking `h2root`'s default behaviour, making only the first character upper case.

`id=ID` Set the ntuple `ID`. Only numbers are allowed for `hbook` files.

`title=TITLE` Set the title of the ntuple.

[3]Most often another backslash is needed to escape the backslash itself on the shell command line, unless proper quoting is used: `--ntuple=TDC\\5\\6_T,...` or `'--ntuple=TDC\5\6_T,...'`. It is not recommended to use digits or underscores as part of variable names.

[4]Allowing for both `.ntu` and `.ntp`.

`ftitle=FTITLE` Set the title of the output file.

`NOSHM` By default, shared memory is used to communicate with the external, `fork`ed process file writer. This options forces communication using a `pipe`.

3.3 Input data diagnostics

Several options provide quick access to the organisation and layout of the raw data files themselves and the packaging of the data.

`--print-buffer` Make a human readable dump of every buffer header (or equivalent, for non-LMD data formats)[5].

```
Buffer          0, Size   16384 Used       0 Thu 1970-01-01 00:00:00.000 UTC
       Events   0 Type/Subtype 2000    1 FragEnd=0 FragBegin=0 LastSz       0
File header:
  Label    R06
  File     R06_0308.LMD
  User     land
  Time     27-Sep-01 09:13:59
  Run
  Exp
Buffer    1082774, Size   16384 Used   10032 Thu 2001-09-27 07:13:46.905 UTC
       Events   3 Type/Subtype   10    1 FragEnd=0 FragBegin=0 LastSz    1444
Buffer    1082775, Size   16384 Used   15868 Thu 2001-09-27 07:13:47.023 UTC
       Events   4 Type/Subtype   10    1 FragEnd=0 FragBegin=0 LastSz    1456
```

`--print` Make a human readable dump of all event and subevent headers.

```
Event     7938336 Type/Subtype   10    1 Size    1444 Trigger  1
  SubEv ID      1 Type/Subtype   34 3100 Size     384 Ctrl   9 Subcrate   0
  SubEv ID      1 Type/Subtype   34 3200 Size     216 Ctrl   9 Subcrate   0
  SubEv ID      1 Type/Subtype   32 3100 Size     800 Ctrl   0 Subcrate   0
Event     7938338 Type/Subtype   10    1 Size     412 Trigger  7
  SubEv ID      1 Type/Subtype   34 3100 Size     392 Ctrl   9 Subcrate   0
```

`--data` Include the data payload, as hexadecimal, with each subevent.

[5]1970-01-01 is the Epoch of UNIX `time` and is seen in the example since the DAQ has apparently only initialised the time field of the file header to 0, not the actual time.

```
Event   270987188 Type/Subtype   10    1 Size     404 Trigger  2
  SubEv ID      1 Type/Subtype   34 3200 Size       8 Ctrl   9 Subcrate   2
 02000000 00004321
  SubEv ID      2 Type/Subtype   32 3130 Size      96 Ctrl   9 Subcrate   1
 28190af1 281c0a73 50910145 5096013e 5099016e b8b00157 68c00103 68c1014c
 68c2014a 68c3016a 68c40101 68c50142 68c60102 68c70129 68c80129 68c90144
 68ca0129 68cb0126 68cc013c 68cd0162 68ce011b 68cf0164 40b10240 40b4017a
...
```

`--dump=LEVEL` Make a textual dump of the unpacked data, along with the names of each variable in the requested `LEVEL` data structures.

`--event-sizes` (This option is only available for the MBS/LMD data format.) After event processing has finished (i.e. just before the program terminates) a summary of the subevents (and events) encountered per trigger type is produced. For each entry, the minimum, maximum and average payload subevent size is presented. The (subevent) header sizes are not included, but presented separately.

```
 type/stype    id crt ctrl    min    max  avg(ev) avg(tot) head  occurances

trig  1:                 (     0   7232)          4236.6 100.6 (      7885)
   34/ 3100    10:  2:  1 (   384    384)   384.0    19.0   0.6 (       390)
   32/ 3130     8:  1:  2 (     4   1516)   180.5   180.5  12.0 (      7885)
   32/ 3130     8:  2:  2 (     8   1368)    19.8    19.8  12.0 (      7885)
   34/ 3200    10:  2:  1 (   124    124)   124.0   124.0  12.0 (      7885)
   34/ 3500    10:  2:  1 (     4     64)     8.0     8.0  12.0 (      7885)
   82/ 8200    10:  0:  3 (  3320   3960)  3479.0  3479.0  12.0 (      7885)
...
trig  2:                 (     0   6376)          3748.1  88.6 (      3549)
   34/ 3100    10:  2:  1 (   384    384)   384.0    19.2   0.6 (       177)
   32/ 3130     8:  1:  2 (     0   1048)    41.8    41.8  12.0 (      3549)
   32/ 3130     8:  2:  2 (     8   1308)    12.6    12.6  12.0 (      3549)
   34/ 3200    10:  2:  1 (     8      8)     8.0     8.0  12.0 (      3549)
   82/ 8200    10:  0:  3 (  3292   3468)  3342.2  3342.2  12.0 (      3549)
...
all trig:                (     0  27628)          4200.3  97.5 (     11972)
   34/ 3100    10:  2:  1 (   384    392)   384.2    19.4   0.6 (       605)
   32/ 3130     8:  1:  2 (     0   1796)   187.5   187.5  12.0 (     11972)
   32/ 3130     8:  2:  2 (     0   1456)    61.6    61.6  12.0 (     11972)
 9934/ 3199    10:  2:  1 (     4      4)     4.0     0.1   0.4 (       437)
   34/ 3200    10:  2:  1 (     8    124)    89.6    89.5  12.0 (     11959)
   34/ 3500    10:  2:  1 (     0     64)     7.5     5.3   8.4 (      8410)
   82/ 8200    10:  0:  3 (     0  24600)  3463.8  3463.8  12.0 (     11972)
...
```

As a general rule, the program will not die immediately upon receipt of a `TERM`inate signal (as e.g. generated by pressing `ctrl+C`), but will instead as quickly as possible stop processing further events, and then run the finalisation routines (among others, making the (sub)event totals display). This behaviour is expected to be the generally most wanted. If the user insists (by sending the `TERM` signal several times), the program will immediately abort, though. The reason for not quitting directly is the desire to terminate cleanly, i.e. run the finalisation routines.

3.4 Error handling

The name `ucesb` is an abbreviation of "unpack and check every single bit", and came about for a reason. The easiest way to ensure that both the unpacker itself and the data are correct is to make, by default, a big fuzz about any small inconsistency, i.e. generously generate warning and error messages.

`--allow-errors` Normally, the program will terminate after 10 errors have been detected during unpacking or event processing (including any user functions). With this option, errors are not a reason for termination.

`--broken-files` Instead of completely aborting processing of data, each file is dismissed after 10 errors, continuing processing with the next file.

`--io-error-fatal` Immediately terminate the program upon any I/O error. Default behaviour is to treat an input source with an error as having reached end-of-file and to continue using any remaining sources. This option is particularly useful when performing merging and repeating (serving) of data from some event builders, as the stream servers sometimes disconnect themselves. By running the `ucesb` data repeater from within an infinite loop shell script, any time a server is disconnected, the connection process is started all over again. Without this option, data from the disconnected server would not be used again until all other sources also were lost, finally causing the repeater to restart.

`--debug` This option has the same effect as `--print --data`, but only for events which suffered some kind of unpack failure.

3.5 Processing and monitoring

There are some general kinds of "analysis" that can be applied to the raw data from a DAQ, independent of the kind of experiment. They can help to verify that the converter DAMs are operational, and to some extent that the channel mappings are correct.

`--watcher=OPT,DET` The watcher (also known as the DAQscope, see Section 6.3 with Figure 6.1 for an example) provides an `ncurses`-based quick view of the detector channel raw spectra, i.e. showing where in the conversion range the data appear.

The `DET`ector channels to be shown are specified similarly as when generating an ntuple. A colon may also be used as an item separator, in which case items demanded before the colon will be placed before any items requested after a colon. This is needed as the item selection routine picks items to display in the order they occur in the data structures of the program, which sometimes makes the information harder to read than necessary, by e.g. intermixing times and amplitudes[6].

`MIN=n` Set the first value of the range of raw values to be displayed in the spectra.

`MAX=n` Set the last value of the range of raw values to display.

`RANGE` For some calibration procedures, each event is associated with a certain value of a specific parameter. One example would be time calibration events, where, for each event, a specific delay between the common signal (e.g. start) and the signals injected into each TDC channel (e.g. stop) is somehow provided for. With this option, such a value can be passed along to the watcher subsystem. The values will then be displayed on the line below the single spectra to show approximately with which value each part of the spectrum is associated. This can be helpful to rather quickly[7] arrange for suitable offsets within the time calibrator signal paths to make them end up reasonably within range.

`COUNT=n` By default, the display will update (and clear) the collected histograms every 10000 events, 15 seconds or when an end-of-spill event is received (as indicated by the user code) — whichever happens first. With this option, updating will happen every `n` events.

`SPILL|BOS|EOS` Update the display on either both of, or only at begin- or end-of-spill.

[6]Thus does `--watcher=POS1-2_1-4T:POS1-2_1-4E` provide a more readable view of the same data than just `--watcher=POS`. Implicit indices and names in-between can be represented by a wildcard asterisk, e.g. simplifying the above selection to `--watcher=POS*T:POS*E`.

[7]As compared to the lengthy procedure when forced to use more elaborate but precise on/off-line analysis methods to determine the offsets currently in effect.

`TIMEOUT=n` Make the display update every `n` seconds.

`--corr=TRIG,DET,FILE` With this option, a 2-dimensional triangular picture of the correlations between the occurrences of hits in different channels is created and written to `FILE` after all events have been processed. The `DET`ector channels to plot are specified with the same comma- and colon-separated list as for the `--watcher` option.

It would in some circumstances be useful to also select which events should be included in the plot, which could be done with the `TRIG` option, not yet implemented, however. The user function can naturally be used to make more advanced selections.

The option can be given several times, producing many different pictures. The unpacker program generates the pictures internally in `pgm` format which is directly piped into a `convert` process, producing a file of the format chosen by the file name extension. A good choice is `png`, since it is loss-less, while `jpg` is not suitable, being lossy and thus introducing unwanted artefacts in the pictures.

`--thresholds` This option is a relic from the ancestor program that unpacked the multi-event data of IS430. It was used to calculate pedestal values (thresholds) to be loaded into the ADCs, to have efficient zero suppression.

In the author's opinion, pedestal determination for use by a running DAQ is one of the few calculations that should be done *within* the DAQ software (see Section 3.3.1 of [4]) — the option has as a consequence silently fallen into disrepair. To be able to do the calculations *also* outside the DAQ would not be evil, so perhaps it should be resurrected. . .

3.6 Miscellaneous

`--show-members` Display a list of all data members available within the program at all levels. The markings `z:` and `m:` means that the arrays are zero-suppressed or multi-entry, respectively — cf. Section 2.2.1.

```
UNPACK_vme_tdc3data32          .UNPACK.vme.tdc[3].data[z:32]
UNPACK_vme_tdc3eob_u32         .UNPACK.vme.tdc[3].eob.u32
UNPACK_vme_scaler0_data32      .UNPACK.vme.scaler0.data[z:32]
UNPACK_vme_scaler0_header_u32  .UNPACK.vme.scaler0.header.u32
UNPACK_vme_adc5data32          .UNPACK.vme.adc[5].data[z:32]
UNPACK_vme_adc5eob_u32         .UNPACK.vme.adc[5].eob.u32
UNPACK_vme_header_failure_u32  .UNPACK.vme.header.failure.u32
UNPACK_vme_header_time_stamp   .UNPACK.vme.header.time_stamp
```

```
        UNPACK_vme_header_multi_events .UNPACK.vme.header.multi_events
        ...
        RAW_BACK2E                     .RAW.BACK[2].E
        RAW_MONE_E                     .RAW.MONE.E
        RAW_MONDE_E                    .RAW.MONDE.E
        RAW_MONTGT_E                   .RAW.MONTGT.E
        RAW_DSSSD2F32E                 .RAW.DSSSD[2].F[z:32].E
        RAW_DSSSD2F32T                 .RAW.DSSSD[2].F[z:32].T
        RAW_DSSSD2B32E                 .RAW.DSSSD[2].B[z:32].E
        RAW_DSSSD2B32T                 .RAW.DSSSD[2].B[z:32].T
        RAW_DSSSD2FT                   .RAW.DSSSD[2].FT
        RAW_DSSSD2BT                   .RAW.DSSSD[2].BT
        ...
```

Each entry is presented twice, both in a `PAW`-like fashion (which e.g. is the format used to specify what signals to include with the `--ntuple=` or `--watcher=` options), and in a more `C`-like style, more closely resembling what is used within the program.

Note that all digits which are not specifying the size of an array are part of the variable names — those will need special escaping on the command line, as detailed under the `--ntuple=` option.

`--calib=FILE` The cable mapping information (used by `ucesb` to generate the `raw` and `cal` level data structures) is stored in the file `gen/data_mapping.hh` which is parsed by the unpacker program to generate the `unpack`-to-`raw` data mappings. With the same parser, also a set of calibration parameters (to be applied at the `raw`-to-`cal` mapping) can be read. They may be placed in the file `calibration.hh`, which is used if it exists. This option gives the opportunity to specify further files.

`--show-calib` Display a list of all calibration parameters.

`--max-events=n` Process at most `n` events before terminating.

`--help` Print a summary of the command line options and quit.

The `ucesb`-based unpackers are also envisaged to be compiled such that any heavy processing performed can be split between several processors, using multiple threads. The implementation (see Section 7.3) is not complete yet; the associated options are so far only of use to the developer.

`--threads=n` Use `n` threads for processing. By default, `n` equals the number of available processors.

`--files-ahead=n` Let `n` files be open simultaneously in advance, to prevent event processing from being delayed by a slow I/O subsystem. This would be particularly useful when the data files need CPU-intensive decompression, e.g. `bunzip2`.

`--progress` Show the file reading progress and the status of the various processing threads using a `ncurses`-based display.

Chapter 4

Raw data structure specification

The raw event data layout is described in a C structure like format. From this, the ucesb unpacker generator will create appropriate code to perform "unpacking with a vengeance" — verifying data integrity by checking all bits of the input.

4.1 Data format specification

The data format descriptions are given in a file project.spec in the project/ directory of the experiment. Before it is parsed by ucesb it will be processed by the C preprocessor, which does file inclusion, macro expansion and comment removal, see Listing 4.1. By putting -*- C++ -*- on the first line of each specification file, GNU emacs will help indent and colour the code nicely, although it really isn't C++ code.

```
// -*- C++ -*-

#define USING_CRATE2

#ifdef USING_CRATE2
// Include the specification for crate2
#include "project_crate2.spec"
#endif

/* And then comes the rest ... */
```

Listing 4.1: (Ab)using the C preprocessor to handle comments, include files and expand macros.

4.1.1 Item blocks, SUBEVENT

The specification is "free format"[1] and each item (an event, subevent or any smaller separate entity) loosely looks like a mix of a structure and a function in C, as in Listing 4.2.

```
SUPER_TDC(slot)
{
  UINT32 value;
}

SUBEVENT(ONE_CRATE)
{
  tdc1 = SUPER_TDC(slot=5);
  tdc2 = SUPER_TDC(slot=6);
}

EVENT
{
  crate1 = ONE_CRATE(type=5);
}
```

```
class SUPER_TDC
{
  uint32 value;
};
class ONE_CRATE
  : public unpack_subevent_base
{
  SUPER_TDC tdc1;
  SUPER_TDC tdc2;
};
class unpack_event
  : public unpack_event_base
{
  ONE_CRATE crate1;
};
```

Listing 4.2: Basic structures with simple items. Corresponding generated `unpack` structures (edited for clarity).

A named item type is defined by the name together with any parameters it may take within parentheses, and a structure body:

```
IDENTIFIER(param1,param2,...)  { ... }
```

The parameters may be given default values, like `param2=5`. Subevents are different in that they are marked as such:

```
SUBEVENT(IDENTIFIER) { ... }
```

and only have implicit parameters, representing the subevent header members, in case of LMD files: `type`, `subtype`, `control`, `subcrate`, and `procid`.

An item is included into another item by stating the name it should have in the enclosing structure and, like a function call, the name of the item type and any arguments. To avoid confusion, the parameter name of each argument is stated explicitly.

```
[qualifier] member = IDENT(param1=arg1,param2=arg2,...);
```

The item may have one or more qualifiers preceding it (only valid within certain contexts):

`multi` Signifies that several instances may occur for data of several physical

[1]Columns or line breaks do not matter. Arbitrary whitespace (space, tab, and newline) is used to separate keywords and identifiers.

events of the same item. Used in a multi-event unpacker, see Section 2.2.2 and Appendix C.

`external` The structure is external (hand written) and not part of the specification, see Section 5.5.1.

`norevisit` The item may only be used once per event. Only applies within a `select several` statement.

`revisit` Only applies to subevents within the `EVENT` declaration. This removes the default limitation of only allowing each item to match one subevent per event.

4.1.2 The `EVENT` declaration

An unpacker has one `EVENT` item, giving the possible contents of the input file. As opposed to items declared in subevents and other items, the subevent items declared within the `EVENT` are not expected to come in the given order, or occur at all. It behaves like a `select several` statement (see Listing C.3 on page 92), where the subevent declaration to use to unpack each subevent is selected by the given subevent header items. Ambiguities are not allowed, and will cause run-time errors. When creating an unpacker that only has explicit interest in some subevents, the statement

```
ignore_unknown_subevent;
```

may be used to ignore unhandled ones, as in Appendix B.

4.1.3 Multiple choice: `select several`

Often, the order of data from various modules is not fixed and entire modules may be completely left out if they did not produce any data. To handle this, a kind of `C switch` statement can be specified: `select several`, see Listing 4.3. All included items are made members of the data structure, and each time the unpacker reaches this point, it will try to match the next data word versus the items. Matching is by default done by calling a special version of the unpacking routine for each item, which determines if that item can handle the data (usually some part of a bit field matching a slot number). For many common cases with bit-fields (where the member items are using e.g. `MATCH` or `RANGE` selectors), the `ucesb` program is able to generate a direct look-up table. Using the significant (i.e. differing) bits for selection by indexing into the look-up array, the time-consuming match-attempts for all declared items are avoided. In any case, if two items match simultaneously, a run-time

```
SUBEVENT(FASTBUS_CRATE)
{
  select several
  {
    adc0 = LECROY_1885(geom=10,channels=96);
    qdc[0] = LECROY_1885(geom=13,channels=48);
    qdc[1] = LECROY_1885(geom=17,channels=48);
    qdc[2] = LECROY_1885(geom=15,channels=48);
  }
}
```

```
class LECROY_1885
{
};
class FASTBUS_CRATE
  : public unpack_subevent_base
{
  LECROY_1885 adc0;
  LECROY_1885 qdc[3];
};
class unpack_event
  : public unpack_event_base
{
  FASTBUS_CRATE crate2;
};
```

Listing 4.3: Items with several components in arbitrary order. The corresponding generated code (edited for clarity) is also shown. The lower panel also has code for some specifications not shown in the upper panel. Listing 4.4 has details on `LECROY_1885`.

error occurs. If no items match, or the subevent is out of data, the loop terminates. Otherwise, the item is unpacked, and the `select several` tries to match another item, for the following data word.

Two variants are also implemented: a simple `select`, which only unpacks exactly one item, and `select optional`, which accepts zero or one items.

4.1.4 Constants, variables and expressions

At most places where a number is expected, either a numerical constant, a variable or an expression can be used. A numerical constant may be expressed as an ordinary base-10 number, hexadecimally or binary, i.e. `19`, `0x13`, or `0b10010`.

Variables may be either parameters of the item block being processed or data members of the structure. Structure members are either persistent or local unpack variables (as directed by the `NOENCODE` modifier, see below). Members of sub-structures (including bit-fields) are accessed by separating the structure instance name and item member name with a dot: `tdc1.value` or `qdc[1].value`. Please note that the `ucesb` program does not keep track of all known variable names, it just reproduces the full names in the generated code, and thus cannot check their validity or existence. That is done by the compiler. Bit-fields are for technical reasons implemented as `C union` structures. In these cases, when a name is part of a bit-field, `ucesb` does know enough to add the `union` name as a prefix.

`ucesb` can also transfer more complicated expressions to the generated code, including parentheses for evaluation order and the most common arithmetic operations: `- + * / ~ & | ^ << >> ( )`. (Variables within the unpacker are implicitly unsigned, making the right-shift operation unsigned.) Both constants and variables can be used. Type-casts can be expressed using `static_cast<type>(var)`, which effectively performs the `C`-style `(type) var` conversion, but had less parsing issues within `ucesb` itself.

4.1.5 Data items: `UINT8`, `UINT16`, `UINT32`

Each basic data item is defined by its length (type) by one of `UINT8`, `UINT16`, or `UINT32`[2] and a name:

```
TYPE identifier;
```

The name may be followed by a `NOENCODE` marker, which prevents the item from being part of the unpack data structure — it is then handled as a temporary variable in the unpacking function, and thus just unpacked and discarded. This is generally used together with specially declared data containers to handle the information payload, see `MEMBER` in Section 4.1.8.

An item may optionally be preceded by either the word `optional` or `several` in which case it may either be omitted altogether, or represent a value occurring multiple times. As the unpacking then operates in a greedy fashion (consuming as much input as possible before proceeding with the next logical item), this makes most sense together with the bit-fields as described below, since those can judge if a data-word has a fixed bit-pattern set or not. Note that `several` can only be used together with `NOENCODE`, as multiple items cannot be stored — `ucesb` only creates multi-entry items for `MEMBER` directives. This should perhaps be fixed, although one would then also need to specify a limit on the number of occurrences.

4.1.6 Bit-fields

Data delivered by digitisation modules is generally packed in 16 or 32-bit words, containing both the digitised value and the channel number of each

[2]Support for 64-bit entries is in progress.

Data word					
31 - 27	26-24	23	22 - 16		11 - 0
geom	n	r	channel		value

Table 4.1: LeCroy 1885/1875 data format [10].

non-zero conversion, see Table 4.1. To enable easy handling of these bit-fields, given a list of which bits contains what variable, `ucesb` can create the needed `C` structures[3]. The bits are numbered from 0, starting at the least significant bit, see Listing 4.4.

Each item in a bit-field begins by specifying the bits, with an underscore to represent ranges, followed by a colon and the name or a constant bit pattern. Constants are checked during unpacking, and bits not specified are implicitly expected to be zero, also verified during unpacking. Each bit may only be part of one variable[4]. The allowed values of a variable can be restricted by `MATCH` or `RANGE`.

4.1.7 Checking bits: `MATCH` and `RANGE`

Each bit-field item can be followed by an equals sign, and either a constant or a `MATCH(value)` or `RANGE(min,max)` statement, which specifies what value or range of values (inclusive) the item may have. The check is done during unpacking, and any mismatch will be reported as an error. As the arguments may be e.g. the parameters given to the enclosing unpack item, it is possible to write generic descriptions of the data from some kind of module, and make e.g. slot numbers variable.

When an unpack item is used as a part of a `select` statement, the `match`ing routine will not report errors for mismatching fields, only conclude that the data cannot be for the current module. Furthermore, when `match`ing items (of a `select` statement), only the first data member of each item is tested, unless the item's structure contains a `MATCH_END` keyword somewhere, marking the end of values to be verified. `CHECK` is mostly a synonym for `MATCH` and should perhaps be deprecated[5].

[3]Taking into account that the ordering of items is different in little and big endian machines.

[4]Without too much trouble, `ucesb` could be extended to generate multiple `unions` to allow for varying interpretations of the bits.

[5]The idea was to use `MATCH` for variables that may be matched, and `CHECK` for any other. The implementation is the same for both though.

```
BITFIELD()
{
  UINT16 data
  {
    0_11:  value;
    12_14: channel;
    15:    overflow;
  }
}

LECROY_1885(geom,channels)
{
  MEMBER(DATA12_RANGE data[96] ZERO_SUPPRESS);

  UINT32 ch_data NOENCODE
    {
      0_11:  value;
      // 12_15: 0;
      16_22: channel = RANGE(0,channels);
      23:    range;
      24_26: n = 0;
      27_31: geom = MATCH(geom);

      ENCODE(data[channel],(value=value,range=range));
    }
}

SUBEVENT(FASTBUS_CRATE) {
  select several {
    adc = LECROY_1885(geom=10,channels=64);
  }
}
```

```
class BITFIELD
{
  union
  {
    struct
    {
      uint16 value:12;
      uint16 channel:3;
      uint16 overflow:1;
    };
    uint16 u16;
  } data;
};
class LECROY_1885
{
  raw_array_zero_suppress < DATA12_RANGE,
                            DATA12_RANGE, 96 > data;
};
```

Listing 4.4: Two structures with bit-fields, followed by the generated structures. The second bit-field `ch_data` is only used during unpacking, the `value` is stored in the zero-suppressed array `data`.

4.1.8 Data members: `MEMBER`

A data member not directly corresponding to data words occurring in the on-file data format can be added to an item's `unpack` structure by using the `MEMBER` directive. This is useful to store lists of items, and required for any variables that should enjoy zero-suppression, as in Listing 4.4. The declaration contains the type and name of the member (together with any array specifications), and optionally a keyword specifying the kind of zero-suppression to be used:

```
MEMBER(TYPE identifier[m][n] ZERO_SUPPRESS);
```

Such a data member may also be sent to a sub-item for filling, in which case the sub-item should have the name as a parameter and also declare it as a `MEMBER` within the body to describe the type. It will then not be made a part of the structure, but sent as a reference to the unpack function.

Zero-suppression always acts along the innermost (last) dimension, and can be of these kinds:

`ZERO_SUPPRESS` Zero-suppressed data structure, suitable for filling in random order.

`ZERO_SUPPRESS_LIST` As above, but only to be used when the items are expected to come in order of increasing indices.

`ZERO_SUPPRESS_MULTI(n)` Two-dimensional zero-suppression. For each indexed item, up to `n` multi-hit items may occur.

`NO_INDEX_LIST` An unindexed list of items, filled as they occur. Note that as opposed to the previous specifier, the length of the list is given as the last array item of the data member itself, see Listing 4.5.

4.1.9 Encoding data members: `ENCODE`

To insert values into arrays or items created by `MEMBER` declarations, the `ENCODE` directive is used:

```
ENCODE(destination,(subitem1=var1,...));
```

The `destination` is the name (possibly with indices) of the `MEMBER` variable, and the assignments tell which destination fields should take what values. It is generally used from within a bit-field specification, where part of the bit-field gives an index, and another part a value. Each `subitem` is a member of the structure holding the `destination` variable. If the `destination` is not

```
GROUP_DATA(group)
{
  MEMBER(DATA16 data[64] NO_INDEX_LIST);

  UINT16 header NOENCODE
    {
      0_7:   group = MATCH(group);
      8_13:  item_count;
      14_15: 0b01;
    }

   list (0 <= index < header.item_count)
    {
      UINT16 value NOENCODE;

      ENCODE(data APPEND_LIST,(value=value));
    }

  if (!(header.item_count & 1))
    {
      // Padding needed to keep 32-bit alignment
      UINT16 pad NOENCODE { 0_15: 0; }
    }
}
```

Listing 4.5: Specification for a `GROUP_DATA` block of EDOC073 data [7]. Each 16-bit data item is inserted into a variable length array `data`. Discarding of the alignment padding is controlled by an `if`-statement. As all direct input items are marked `NOENCODE`, the `data` array is the only surviving result of unpacking.

a structure, but a basic type itself, use a single underscore as the `subitem` name. `var` may be a variable or an expression as detailed in Section 4.1.4.

When the array is of the `NO_INDEX_LIST` kind, the flag `APPEND_LIST` should follow the `destination`, but no index.

4.1.10 Counted number of data words: `list`

Much like a for-loop, a specified number (e.g. from a header word) of items can be unpacked using a list:

```
list(min <= index < max) { ... }
```

`index` is the loop variable, it can be used for checking and indexing data member assignments within the loop body. `min` and `max` specify the iteration. Generally, `min` will be `0` and `max` the number of items to read. The relational operators are part of the syntax, and not modifiable.

4.1.11 Conditional unpacking: `if`

Parts of an item structure unpacking can be made conditional:

```
if (expr) { ... }
```

where `expr` is some variable or expression that can evaluate to true or false (non-zero or zero). `expr` may also specify a user-written member function of the current class, by preceding the name of the function (without parentheses for the function call) with the `external` keyword. The function is to have the prototype:

```
uint32 unpack_class::identifier() const;
```

The body of the `if`-statement may be followed by an `else`-clause:

```
if (expr) { ... } else { ... }
```

or

```
if (expr1) { ... } else if (expr2) { ... } else { ... }
```

4.1.12 Checking word counts

To facilitate the verification of word counts (usually found in a trailer word of some modules, see Listing 4.6), various locations within the data stream can be marked using

```
MARK_COUNT(mark);
```

where `mark` is a unique identifier. It will be handled as a simple `void*` pointer.

Like a `MATCH` statement for the bit-field variable containing the word count,

```
CHECK_COUNT(mark1,mark2,offset,multiplier)
```

checks that the number of bytes between the positions `mark1` and `mark2` plus the `offset` (in bytes) is equal to the value times the `multiplier` (i.e. the counted word size). As an example, if a trailer which contains the word count should be included, a positive offset is to be used, since all `mark`s must be declared before the variable is verified.

```
VME_CAEN_V1190(geom)
{
  MARK_COUNT(v1190_start);

  UINT32 header
    {
      0_4:   geom = MATCH(geom);
      5_26:  event_number;
      27_31: 0b01000;
    }

  // ... Actual TDC data

  MARK_COUNT(v1190_end);

  UINT32 trailer
    {
      0_4:   geom = CHECK(geom);
      5_20:  word_count = CHECK_COUNT(v1190_start,v1190_end,4,4);
      24:    tdc_error;
      25:    buffer_overflow;
      26:    trigger_lost;
      27_31: 0b10000;
    }
}
```

Listing 4.6: The CAEN V1190/1290 family of TDCs emit the word count at the end of the data for each event.

4.2 Data mappings: SIGNAL

The mappings of data from the `unpack` to the `raw` level data structures are also part of the specification. They at the same time give the layout of the `raw` level — items are created in the structures as needed. Each mapping:

`SIGNAL(raw_name,unpack_name_C,type ["unit"]);`

describes the logical detector signal `raw_name` together with the source item `unpack_name_C` and the data `type`. The `type` must be the same as in the `unpack` structure, or a run-time error will occur upon program start-up, as the actual mappings cannot be set up. Optionally, the `unit` of the value can be specified, see Section 4.3. The source is specified using the `C` notation of structure items, while the destination uses a more `PAW`-like format[6], with underscores (or digits) separating names between structure encapsulation levels. Any series of digits is treated as an index. It is currently impossible to specify a digit to be part of a `raw` level name.

Listing 4.7 gives various examples of possible mapping specifications,

[6]Note that the string-like `PAW`-resembling format starts numbering at one, while the array-like `C` notation uses zero-based indices.

```
// Declarations of single detector signals
SIGNAL(DETA_1_T2, vme.tdc[1].data[6], DATA12);
SIGNAL(DETA_1_E, vme.qdc[2].data[12], DATA12);

// Create an item in the raw level without source
SIGNAL(DETB_5_T, , DATA12);

// A list of mappings/items
SIGNAL(DETC_2_FRONT_1_T, vme.tdc[2].data[0],
       DETC_2_FRONT_16_T, vme.tdc[2].data[15], DATA12);

// Make an array zero-suppressed
SIGNAL(ZERO_SUPPRESS: DETC_1_FRONT_1);

// An item with cal level entry, and units
SIGNAL(DETD_1_E, vme.qdc[3].data[0], (DATA12 "ch",float "#MeV"));

// An item which is only mapped for the last physical event in a
// multi-event unpacker
SIGNAL(LAST_EVENT: SCALER_1, vme.scaler.data[0], DATA32);
```

```
struct raw_event_DETA
{
  DATA12 T[2];
  DATA12 E;
};
struct raw_event_DETB
{
  DATA12 T;
};
struct raw_event_DETC_FRONT
{
  DATA12 T;
};
struct raw_event_DETC
{
  raw_array_zero_suppress < raw_event_DETC_FRONT,
                            raw_event_DETC_FRONT, 16 > FRONT;
};
struct raw_event_DETD
{
  DATA12 E UNIT ("ch");
};
struct raw_event
  : public raw_event_base
{
  raw_event_DETA DETA[1];
  raw_event_DETB DETB[5];
  raw_event_DETC DETC[2];
  raw_event_DETD DETD[1];
  DATA32 SCALER[1];
};
```

Listing 4.7: Signal mapping variants, with corresponding generated `raw` level structures (edited for clarity).

along with the generated data structures. By giving an entry without a source, it is possible to enforce the creation of a `raw` level item of that name, as well as enlarging an array to accommodate the given index. By giving two source and destination names, which are similar except for one index, a range of mappings can easily be described:

```
SIGNAL(raw1,unpack1_C,raw2,unpack2_C,type ["unit"]);
```

The program also supports generation of a `cal` level data structure, with a similar layout as the `raw` level. This can be used to implement simple linear conversions from the integer data delivered by the modules to floating point values representing more physical quantities. Whenever this is wanted, the data type is declared as a pair, giving the `raw_type` and `cal_type`, respectively:

```
SIGNAL(raw,unpack_C,(raw_type ["unit"],cal_type ["unit"]));
```

To make an array zero-suppressed, the name of the array (ending with the array index to be compacted) should be preceded by a keyword:

```
SIGNAL(ZERO_SUPPRESS: raw_array_name);
```

This will create an array optimised for random fill order at the `raw` level, as the mappings are expected to be complex due to e.g. anti-cross-talk cabling schemes. The `cal` level (if present) will use a representation suitable for ordered filling, since the items would come 1-by-1 from the `raw` level. The keywords `NO_INDEX_LIST` and `ZERO_SUPPRESS_MULTI` are also available, and work as for data `MEMBERS`, cf. Section 4.1.8.

Finally, an item can be marked `LAST_EVENT` (or `FIRST_EVENT`) to make it only map for the last (or first) event when each `unpack` event contains data for multiple physical events.

In addition to the creation of appropriate `raw` and `cal` level structures, the `ucesb` program also generates a text table `gen/data_mapping.hh`, containing the data mappings to be performed. That file is read by the unpacker program at startup, and parsed using the same mechanism that parses calibration parameters for the `raw` to `cal` level conversions (see below). The mappings have the format

```
SIGNAL_MAPPING(type, name, src_C, dest_C);
```

where `type` and `name` are ignored (used for information only), and `src_C` and `dest_C` are the `C`-form specifications of the source (at `unpack` level) and destination (at `raw` level) data items.

4.2.1 Calibration parameters: `raw` to `cal` level conversions

In addition to, or helping, the user functions (where arbitrary code can be inserted), the unpacker program can do simple linear data conversions while transferring data from the `raw` level structures to the `cal` level. The transformation parameters are entered in a file `calibration.hh`, which is automatically read by the unpacker at start-up, if it exists. The `--calib=FILE` option can be used to choose what parameters to load by using another file. Each parameter is given as either of

```
CALIB_PARAM(var, conversion,value1 [unit1],
                            value2 [unit2], ...);
CALIB_PARAM_C(var_C, conversion, value1 [unit], ...);
```

where `var` (or `var_C` in `C`-form) is the name of both the source (at `raw` level) and destination (at `cal` level) variable[7]. `conversion` determines the type of transformation to do along with the `values`. Correct `units` must be supplied, i.e. when the variables at `raw` or `cal` levels have units associated. The currently known operations are shown in Table 4.2.

Name	values	Operation
`SLOPE_OFFSET`	`slope`, `offset`	$cal = raw \cdot \texttt{slope} + \texttt{offset}$
`OFFSET_SLOPE`	`offset`, `slope`	$cal = (raw + \texttt{offset}) \cdot \texttt{slope}$

Table 4.2: `raw` to `cal` level transformations. The coefficients (e.g. `slope` and `offset`) are the calibration parameter values used in order.

4.3 Unit and prefix handling

To assist the user in using consistent units and powers of magnitude of variables and calibration parameters, units can be associated with `raw` and `cal` level variables. The affiliations are defined in the `SIGNAL` specifications.

A unit consists of one or several more basic units, separated by spaces or asterisks or slashes, the latter two marking the sign of the exponent of following basic units. Exponents (also negative) are given after a caret (`^`).

[7]For these conversions, the unpacker only supports mappings between the same variables. The actual mapper code would support arbitrary mappings, but the zero-suppressed data structures expect the items to come in order.

Examples: `ns`, `cm/ns`, `MeV/ch`, `cm^2`. It is allowed to give calibration parameters with other prefixes than those used in the variables, e.g. specifying a gain factor as `keV/ch` with the `cal` level variable using `MeV`. In this case, the unit of the variable definition must have all its (replaceable) prefixes preceded by hashes, e.g. `#MeV` or `#cm/#ns`. A unit definition of a variable with no prefix (i.e. unity) can still allow prefixes, by using a double-hash: `##s`. The program does not have any deeper knowledge of units, it only matches the basic unit constituents in arbitrary order, requiring exponents to add up correctly. In particular, there is no support (yet) for any conversions, e.g. $1\ \mathrm{G} = 10^{-4}\ \mathrm{T}$.

Gain parameters connecting the `raw` and `cal` levels must have the correct unit for the multiplication involved in the conversion, i.e. the unit of the `cal` level variable divided by the unit of the `raw` variable. To make the calibration parameters look sane, it is suggested to use the unit `ch` (as in channels) for `raw` level variables, where applicable.

Chapter 5

Compilation and user code

The sources making up one unpacker have various origins: code created from the data format specifications, generic unpacker sources and possibly user code. In addition, the programs that generate code are tightly coupled to the functionality provided by the generic sources, and thus frequently need to be synchronously rebuilt during development of `ucesb` itself.

5.1 Sources

Welcome to pre-processor heaven!

First, an overview of what makes up the unpacker sources (from within the `unpacker/` directory):

`eventloop/` Common sources used by all unpackers. Command line parsing, event loop and other execution dispatches. Contains the generic data structures heavily drawn upon by the generated code, as well as wrappers to instantiate various functionality from generated templates.

`common/` These sources are shared with the code generation programs `ucesb` and `psdc`.

`file_input/` The input stage and file unpack parts, replacing the `MBS event-api`. Similar to, and partially lifted from `lmd_read/` of `land02`[1] — it has gradually become better than the original.

`threading/` Mostly experimental routines, for later use with `USE_THREADING`.

`watcher/` The `ncurses` based watcher, borrowed from `land02` and enhanced.

[1]`land02` is a batch mode detector calibration and reconstruction system for experiments performed with the ALADiN-LAND setup at GSI.

`hbook/` This directory is exactly the same as in `land02` (it is an ugly hack — done via a symlink in the CVS tree — but works beautifully). Therefore, `ucesb` has the same ntuple stager and writer as `land02`.

`ucesb/` The data format specification parser and unpacker code generator.

`psdc/` A data structure parser and mirror (reflection) structure and function generator.

`spec/` Contains data format specifications for a few commonly used modules.

The many other directories contain experiment-specific sources or other examples, see Appendix A.

5.2 Compilation

To make things easy, the `Makefile` in the root `unpacker/` directory governs the building of all programs, both code generators and example unpackers. It should be enough to issue:

```
$ make
```

To build one specific program, its name is given as an argument, like `make is430_05`. Each program is usually created with the same name as its directory, e.g. `is430_05/is430_05` is the resulting executable.

The rules for building one unpacker are in `makefile_unpacker.inc`. This may in turn include additional rules using the file `makefile_additional.inc` in the specific directory of the unpacker being built.

To maintain the sources for an experiment specific unpacker outside the `unpacker/` tree, it is enough to arrange for a `Makefile` within that directory to locate and invoke `make` with the common `makefile_unpacker.inc`. An example of such a `Makefile` can be found in the `hacky/` example directory.

5.3 Compile options

As some features of the unpacker depend on external libraries that are not always available, they are enabled by compile time options, via environment flags. They can be enabled at compilation by specifying `make USE_...=1`.

`USE_PTHREAD` Compile the unpacker with support for multiple threads. Even without the multi-threaded event processing (`USE_THREADING`) some other operations will run in separate threads. Most notable, all data input (except direct `mmap`) will be buffered and the reader will be operating asynchronously relative to data processing, helping to hide most network latencies. This by avoiding the serialisation caused by processing the events of one or a few buffers, then request a new buffer and wait for it to be delivered before being able to process further events[2].

`USE_MERGING` Make the unpacker program capable of keeping several input files open simultaneously, so that events can be processed in order, even when they were recorded alternatingly to multiple files. Currently, due to some technical limitations (pointer handling within the event structures), this disables some other program functionality, notably ntuple-generation. Nevertheless, `raw` level inspection and file output is possible.

`USE_CERNLIB` With `cernlib` available, the program can be compiled to produce `hbook ntuple` files, using the `--ntuple=` command line option. It requires the environment variable `CERN_ROOT` to be correctly set[3]. This option has been superseded by the following option:

`USE_EXT_WRITER` For simplified compilation and increased performance, the `hbook ntuple` and `root` file writing as well as the `structure` server are furnished by external processes. They are automatically forked as needed by the `--ntuple=` command line option, and receive the event-wise data via shared memory or a pipe. The build system will only attempt to compile the required `cernlib` and `root` components if the necessary libraries are found.

`USE_CURSES` Enables use of the `ncurses` text terminal control library, needed for the `--watcher` and `--progress` options.

The options controlling the input file format flavour (see Section 2.4) are specified within each specific project itself, and thus are not given externally. They are `USE_LMD_INPUT`, `USE_PAX_INPUT`, `USE_EBYE_INPUT` and `USE_HLD_INPUT`. The alternative option enabling the `MBS eventapi` (also reading LMD data), `USE_MBS_INPUT`, is deprecated as it offers no advantages.

[2]Any forked decompressor (`gunzip`, `bunzip2` or `lzma`) is always a separate processes, and as such would (where possible) run in parallel with the event processing. However, in cases where the decompress programs themselves have fairly small output buffers, and the system pipe size (usually 4096 bytes on Linux, 64 kB by default on FreeBSD) is smaller than a buffer, operation would be serialised without the `USE_PTHREAD` option.

[3]This is usually performed by some login script. On a Debian system using the precompiled `cernlib` packages, setting `CERN_ROOT=/usr` is enough.

It performs worse than the common input stage and has a more cumbersome interface.

The compilation process will itself determine if a suitable[4] rfio library can be found or not.

5.4 Code generation

An unpacker also consists of source files that the ucesb and psdc programs generate, see Figure 5.1. They will be placed in the gen/ subdirectory of the particular unpacker being compiled. These are some C++ source files (the unpacker code itself) and header files which declare data structures and templates with macros for functions to iterate over the structure members. These header files are included from several files in the eventloop/ directory, which define macros necessary to turn the templates into something that actually can be compiled.

The code and structures are first generated by ucesb, into one single output .uce file, from which the various needed parts can be extracted. These are the unpack level data structure, associated unpacker code, raw and cal level data structures and listings of inter-level data mappings.

To be able to do the unpack $\rightarrow$ raw and raw $\rightarrow$ cal mappings, let the watcher run, enumerate data items for ntuple generation, and so on, we need both 'parallel' data structures (having the same layout, i.e. names and branching, but with different content) and functions that do the iterations over the structure members. These mirror objects are generated by the psdc program, which takes an ucesb-generated header-file as input and produces one output .psd file, from which, once again, the wanted parts (structures, functions) are extracted into individual files. These files are littered with macros, such that the generic code can give prefixes to names, insert conditional and loop constructs, declare temporary variables, etc., by defining the macros differently before the various inclusions of the generated files.

Examples of this practice can be found in the files eventloop/watcher.cc, eventloop/struct_fcns.cc and eventloop/struct_mapping.hh. Each of these inclusions begin with 10-15 #defines, followed by #includes for the generated files — e.g. in gen/struct_mirror.hh for creating mirror structures or gen/struct_fcncall.hh for iterating functions — and finish by #undefining the macros.

Each nesting level in a structure is handled as its own (sub)structure, and thus also has its own associated iterating functions for various tasks. The

[4]Only the GSI-specific implementation has been extensively tested so far.

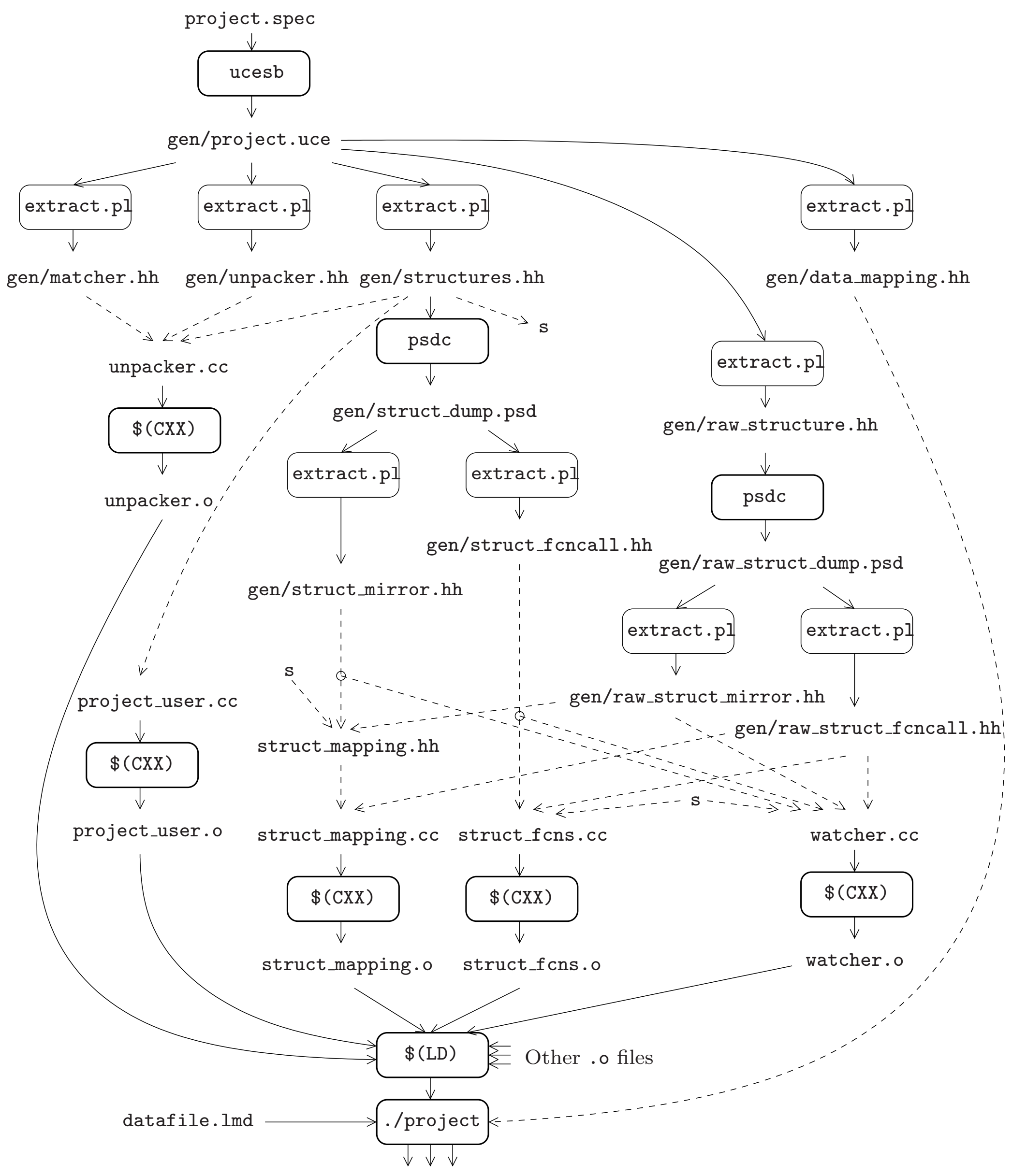


Figure 5.1: Code generation and compilation process. Dashed lines mark files used by inclusion. `project.spec` and the optional `project_user.cc` are the unpacker specific files. Files prefixed by `gen/` are generated and other `.cc` and `.hh` files are part of the generic sources (which become customised by including the generated files). The `extract.pl` script is used to let the specification parser and code generator `ucesb` and the structure reflector generator `psdc` easily create all needed output in only one invocation. `$(CXX)` and `$(LD)` are the system `C++` compiler and linker, respectively.

trick with the zero-suppressed data (i.e. members marked ZERO_SUPPRESS etc.) is that they are defined using the generic templated wrapper classes, that have equally named functions declared, which will be interwoven with the generated code.

5.4.1 The `ucesb` unpacker generator

After pre-processing by the `C` pre-processor (to discard comments and include additional files), the contents of the `.spec` files are parsed by the `ucesb` program, building an internal representation of the specifications. When parsing is complete, some consistency checks are performed on the full information. The data mappings are inserted into list and tree structures.

For unpacking of the individual data structures, code is recursively generated. For each data structure, one structure declaration and one function is created to do the unpacking of the input data, filling the accompanying structure. A second, non-writing version of the unpacker (matcher) is also generated, to be used for `select` statements.

Each data mapping pair has two entries, one which defines the target name at `raw` level, and one identifying the source item at `unpack` level. The source items have names already corresponding to the described data format. From all the `raw` level destination names, a tree is built, with branches for each sub-name and arrays for the indices. When completed, structures are generated for each level in the tree.

5.4.2 The `psdc` reflector creator

For many of the tasks that the unpacker program can perform — generating ntuples of various subsets of the variables available, creating correlation plots and “watching” selected items, as well as mapping data from one level to another in a dynamic way — it needs to actively have some knowledge about its own data structures, more than just having them compiled. Reflection is the process by which a computer program can observe and modify its own structure and behaviour.

The generated unpacker programs do not have a full-featured reflection system by which they can create arbitrary items etc., but with some help from the `psdc` program (that can parse a subset of `C` structures[5]) the needed functionality is obtained. For this two things are required: Firstly, a way to iterate over and operate on all members of a structure, including substructures. Secondly, structures parallel to the data containers which can store

[5]One limitation being that all substructures must be defined separately, i.e. not nested, but before the structure using them.

some associated information for each member of the original structure, like mapping destinations or cumulative results for many events (as done by the watcher).

Both these ends are achieved by `psdc`, which creates, for each structure, both a function that operates on all items and a mirror structure[6]. After code generation, examples can be found in the various unpacker directories as `gen/*_fcncall.hh` and `gen/*_mirror.hh`. Both functions and structures are riddled with macros. The generated files are then included from other sources, where the macros have been appropriately defined, such that the compiler can create executable code with the wanted functionality.

5.5 User functions

To allow arbitrary processing of the data, user-specified functions can be called at various stages of the event-wise transformations, see Figure 2.1. A user function is enabled by defining its name in the file `control.hh`:

```
#define CAL_EVENT_USER_FUNCTION project_user_function
```

and adding the object file to the project's `makefile_additional.inc`:

```
OBJS += project_user.o
```

as well as enabling the use of the `control.hh` file:

```
CXXFLAGS += -DCONTROL_INCLUDE
```

The prototypes of the functions are given in the generic file `user.hh`:

`void INIT_USER_FUNCTION()`

Used to make any needed initialisations of data structures declared and used in the file(s) with the user functions. Please note: there is currently no really comfortable way to access the various event data structures. It would at times be convenient to store also some cumulative data directly inside some (external, i.e. hand written) unpacker classes. However, as this will require careful handling for threaded operation, a more permanent and suitable interface would be needed, but has not been designed yet.

`void EXIT_USER_FUNCTION()`

Called after all events have been processed, to do any cleanup or presentation of any cumulative results. Please note: this function is intentionally called also if the program is aborted, e.g. by pressing

[6]Mirror structures — that's what is to be expected from a reflector. . . ?

ctrl+C, as the TERMinate signal is caught and just causes event processing to cease.

```
int UNPACK_EVENT_USER_FUNCTION(unpack_event *event)
```

Called when the data of one event has been unpacked. Can be used to perform additional consistency checks, e.g. verify event counter synchronisation between different modules. For multi-event unpackers, it must perform the necessary assignments of the data in the unpack structure to physical events, and return the total number of events contained. A return value of 0 prevents further processing of the event.

```
void RAW_EVENT_USER_FUNCTION(unpack_event *event,
                             raw_event *raw_event
                             MAP_MEMBERS_PARAM)
```

Called after data has been mapped from the `unpack` to the `raw` level structure. The `MAP_MEMBERS_PARAM` is only used when handling multi-event data, to tell which physical event is processed.

```
void CAL_EVENT_USER_FUNCTION(unpack_event *event,
                             raw_event *raw_event,
                             cal_event *cal_event,
                             USER_STRUCT *user_event,
                             MAP_MEMBERS_PARAM)
```

Called after the data has been mapped into the `cal` level data structure. If the unpacker has defined its own user level data structure (`USER_STRUCT`, see Section 5.5.2), this is also available for filling here.

```
void WATCHER_EVENT_INFO_USER_FUNCTION(
          watcher_event_info *info,
          unpack_event *event)
```

To enhance the displayed information, the watcher must know which class of triggers each event belongs to (given by the `_type` member of `watcher_event_info`). It also needs to know when it is a good time to update the display, e.g. at end-of-spill events (directed by the `_display` member). See `eventloop/watcher_event_info.hh` for further details and options.

```
bool CORRELATION_EVENT_INFO_USER_FUNCTION(
               unpack_event *event)
```

This function is used to tell which events are suitable for inclusion in correlation plots.

```
bool COPY_OUTPUT_FILE_EVENT(lmd_event_out *event_out,
                            FILE_INPUT_EVENT *file_event,
                            unpack_event *event,
                            const select_event *select)
```

Additional subevents can be inserted into any output data stream using this function. The destination is `event_out`, while `file_event` holds the original event data and `event` has the unpacked information, from which the new subevents can be constructed. The user's command line wishes of which subevents to include are handled with `select`.

```
void HANDLE_COMMAND_LINE_OPTION(const char *arg)
```

Command-line options to control aspects of the user code are handled by this function being called for all options not recognised by the common code. It should return `true` for successfully handled options. See the `is430_05` unpacker for an example. Note that this interface may be modified when (and if) the recognition of command-line options is changed to use `getopt()`.

```
void USAGE_COMMAND_LINE_OPTIONS()
```

Print short help messages for the additional options supported by the above function. Called in response to `--help`.

5.5.1 External (hand-written) unpackers

When the data format (of a subsystem) is too complicated to be described by the specification language, the data structures holding the data as well as the associated unpacker itself can be written manually and seamlessly invoked by the generated code.

With hand-written code, the possibilities are virtually unlimited. This section will only deal with the basic inclusion of such a structure. For further reference, some of the examples in Appendix A make use of external data structures and unpackers.

Listing 5.1 shows how the external class `EXT_COMPLEX_ITEM` must first be declared (with any arguments, each of type `uint32`), such that the code generator `ucesb` knows the calling sequence, and then used with the `external` keyword, such that it does not to try to find a real declaration.

To make the declaration of the class available when compiling the generated unpacker code, the name of a header (`.hh`) file including it should be specified with the `USER_EXTERNAL_UNPACK_STRUCT_FILE` macro in the file `control.hh`:

```
external EXT_COMPLEX_ITEM(id);

SUBEVENT(COMPLEX_SUBEVENT)
{
  select several
    {
      external item1 = EXT_COMPLEX_ITEM(id=3);
      external item2 = EXT_COMPLEX_ITEM(id=5);
    }
}
```

Listing 5.1: Including an external structure and unpacker from a data format specification.

```
#define USER_EXTERNAL_UNPACK_STRUCT_FILE "project_external.hh"
```

The details of the construction of an external unpacker is outside the scope of this write-up, thus the following description is only a broad outline. When creating a new external unpacker, it is suggested to investigate the existing ones in the example directories, and use a suitable one as template.

Class declaration

Listing 5.2 shows the outline of a class declaration. First, some common declarations must be included, e.g. the type of the data source pointer (`__buffer`) which is used by the unpacker function:

```
#include "external_data.hh"
```

This file also contains some convenient macros for reading the input data, as well as some small external unpackers, which could also be used as templates:

`EXTERNAL_DATA_SKIP` Advances the input data pointer to the end of the subevent, effectively discarding the data.

`EXTERNAL_DATA16` Retains a pointer to the input buffer and the length of the remaining subevent data. The input pointer is advanced to the subevent end. The retained pointer and length can be used to handle the data in a user function instead. This is possible as the file input buffer is never released until the event has been completely processed and retired, see Section 7.1.

`EXTERNAL_DATA32` Similar as above.

A natural ingredient is the data members that should hold the event-wise unpacked data. For performance reasons, excessive memory allocation and

```
#include "external_data.hh"

class EXT_COMPLEX_ITEM
{
public:
  EXT_COMPLEX_ITEM() { }

public:
  /* Data members... */

public:
  void __clean();
  EXT_DECL_UNPACK_ARG(uint32 id);
  EXT_DECL_MATCH_ARG(uint32 id); // If part of a select statement

public:
  DUMMY_EXTERNAL_DUMP(EXT_COMPLEX_ITEM);
  DUMMY_EXTERNAL_MAP_MEMBERS(EXT_COMPLEX_ITEM);
  /* more dummy functions... */
};

DUMMY_EXTERNAL_MAP_STRUCT(EXT_COMPLEX_ITEM);
/* more dummy structures... */
```

Listing 5.2: Declaration of an external data structure.

deallocation should be avoided. Memory should either be statically reserved for the worst case item size, or reallocated as needed when an event requires more space then was previously needed. Special care may be needed when using multi-threaded operation, as that will have several hundred events in-flight at the same time, possibly causing very large reservations.

In addition to the data members, the unpacker class must have some member functions:

`void __clean()`

Called before every event for each instance of the item to reset the data structure members to an empty state. Often, when using some sort of zero-suppression, this only means to reset a counter or bit-field. Touching all allocated memory should generally be avoided, lest performance shall suffer.

`template<typename __data_src_t>`
`void __unpack(__data_src_t &__buffer,...)`

Called to perform the unpacking of one item, i.e. reading the input data using the `__buffer` data source pointer and filling the data members as needed. As the `__data_src_t` is a `templated` type (the actual data source handling is compile time generated in several versions

```
#include "external.hh"
#include "error.hh"

void EXT_COMPLEX_ITEM::__clean()
{
  /* Reset the data members for a new event... */
}

EXT_DECL_DATA_SRC_FCN_ARG(
  void,EXT_COMPLEX_ITEM::__unpack,uint32 id)
{
  uint32 header;
  GET_BUFFER_UINT32(header);
  if (header != id)
    ERROR("Header (0x%08x) mismatch id (0x%08x).",header,id);

  while (!__buffer.empty())
    {
      uint32 w1;
      GET_BUFFER_UINT32(w1);
      if (!(w1 & 0x80000000))
        break;

      /* Insert the data words into the structure... */
    }
}
EXT_FORCE_IMPL_DATA_SRC_FCN_ARG(
  void,EXT_COMPLEX_ITEM::__unpack,uint32 id);

// The __match() function is needed when the structure is part of a
// select statement.

EXT_DECL_DATA_SRC_FCN_ARG(
  bool,EXT_COMPLEX_ITEM::__match,uint32 id)
{
  if (__buffer.left() < sizeof(uint32))
    return false; // not enough space for entire header

  uint32 header;
  GET_BUFFER_UINT32(header);
  return header == id;
}
EXT_FORCE_IMPL_DATA_SRC_FCN_ARG(
  bool,EXT_COMPLEX_ITEM::__match,uint32 id)
```

Listing 5.3: Minimal functions needed for an external unpacker. The data consists of a `header` word with the instance `id`, followed by data words, each marked with the high bit set. Note the liberal use of the `ERROR` macro whenever any irregularity in the data is encountered, in line with the `ucesb` motto: “Unpack and check every single bit.”

to handle byte-swapping and unscrambling), the function definition is normally done using the convenience-macro `EXT_DECL_UNPACK()`, or `EXT_DECL_UNPACK_ARG(...)` if the item has some parameters.

```
template<typename __data_src_t>
static bool __match(__data_src_t &__buffer,...)
```

This function is only needed when the structure is part of a `select` statement, in which case it should determine if the data at the current point comes from the current item instance. Usually the external unpacker would be declared with some parameters to be used at the identification. The function is declared `static` as it must not perform any unpacking, this is left for the `__unpack` function upon a successful match.

To make the data structures available for use by all the features of the unpacker, many provided directly at the command line (e.g. text dumps of the data, member lists, mapping of data to `raw` level structures), several further functions are needed. To not make their presence mandatory, `DUMMY_...` macros are provided to declare them, such that compilation of the generic code, that expects them to exist, works.

Function implementation

Listing 5.3 shows some small `__unpack` and `__match` functions for a simple data format. The convenience-macro

```
GET_BUFFER_UINT16|32(destination)
```

is used to retrieve the data words from the input `__buffer`. It wraps the functions that handle any needed byte-swapping, as well as throwing an error if the subevent unexpectedly runs out of data.

Once again, the `EXT_DECL_DATA_SRC_FCN[_ARG]` macros are used to hide the `template` character of the `__buffer`. To allow the function bodies to be placed in a separate compilation unit (`.cc` file) and not require their presence along with the prototypes (in the `.hh` file), the needed `template` versions are instantiated by the `EXT_FORCE_IMPL_DATA_SRC_FCN[_ARG]` macros.

5.5.2 Custom `user` event and `calibration` structures

Variables derived during reconstruction in the `CAL_EVENT_USER_FUNCTION` can be made available for dumping in ntuples etc. as a `user` level by including them in a user-defined structure, named in the file `control.hh` by:

```
#define USER_STRUCT project_user_struct
```

The structure must be declared or included from a header file whose name is given in the project-specific makefile `makefile_additional.inc`:

```
USER_STRUCT_FILE = project_user_struct.hh
```

An illustration of this practice can be found in the `hacky` unpacker example. As these structures must be parsed by the `psdc` reflector generator in order for the data members to be known to the generated internal routines (to be considered for inclusion in ntuple output as well as being externally assignable, see below), certain restrictions on their layout and members apply — see Section 5.4.2. Each structure must contain the macro `USER_STRUCT_FCNS_DECL`, as exemplified in Listing 5.4.

```
struct detector_data
{
  float hit_x  UNIT("#cm");
  float hit_t  UNIT("#ns");
  float speed UNIT("#cm/#ns");

  uint8 nhits;

  USER_STRUCT_FCNS_DECL;
};

struct my_user_struct
{
  detector_data dets[2][2];

  USER_STRUCT_FCNS_DECL;
};
```

```
struct detector_calib
{
  float t_common   UNIT("#ns");
  float t_offset[16] UNIT("#ns");
  float t_slope     UNIT("#ns/ch");

  float wire_spacing UNIT("#cm");

  CALIB_STRUCT_FCNS_DECL;
};

struct my_calib_struct
{
  detector_calib dets[2][2];

  CALIB_STRUCT_FCNS_DECL;
};
```

Listing 5.4: Structures for `user` level data and `calib`ration parameters, for a setup with 2×2 detectors of the same kind. Units and prefixes are used generously.

It is also possible to declare a structure for holding calibration parameters which can be read and assigned during program startup (see Section 4.2.1). The structure is declared analogously to the `user` level data structure, with `USER` replaced by `CALIB`. It is accessible as the global variable `_calib`. To take advantage of the unit checking system on input, each structure member can be associated with a `UNIT`, cf. Section 4.3. Note that the unit and prefix handling is a matter internal to the generic code of a UCESB unpacker, the user-supplied analysis routines deal with the variables in a normal manner.

Chapter 6

Output and Monitoring

Without any options given for output, the unpacker program will only unpack the data, verifying the integrity of the input files. While already this may be enough to identify problems, it is not too exciting. . .

6.1 Ntuple generation

An “ntuple” contains the data items of a (possibly large) structure stored for many events, arranged to easily process the information event-wise [9]. In this respect, also the input files (`.lmd`, `.pax`, `.hld`. . .) would qualify, but the key concept of an ntuple is the ability to access one or a few members within many or all of the events easily, without having to bother about all other items. Imagining the events as rows and each structure item as a column, the storage is thus actually more column-oriented than the stricter row-wise orientation of the input data files. One or more ntuples (as well as some other objects, e.g. histograms) can be stored in `cernlib` `hbook` files, for use with e.g. the PAW [1] interactive system. With this, single and multi-dimensional plots and histograms of the variables and correlations between them from many events can be made. A `root` tree is the equivalent (more advanced) form of ntuples for the ROOT [2] data analysis framework.

`hbook` ntuples come in two flavours: row-wise and column-wise. The row-wise ntuples only support a fixed list of items per event (which are always present) and each item is a floating point variable. These are not efficient (or convenient), so although supported by the unpacker program, they will not be considered further. The column-wise ntuples support both integer and floating point data types, as well as giving ranges for the items. This is especially useful for the integers, whereby also the number of bits used to store the items will be kept to a minimum. A limited-range integer may also be used to define the event-wise array size of other items, such that zero-suppression can be achieved. Multiple levels of zero-suppression are not supported, and there is no support for multi-level (nested) structures — all variables have to be flattened to the top level.

To create a column-wise `hbook` `ntuple`, `cernlib` provides three functions:

`HBNT(id,title,options)`

Used once to create the ntuple and give it a name.

`HBNAME(id,block_name,pointer,description)`

Called (several times) at startup to add variables to the ntuple. For each set of variables, the names, types and ranges must be provided, along with a pointer to a structure in memory that has the layout described. (Variable sized arrays have the worst-case size as given by their size-controlling variable's maximum limit).

`HFNT(id)`

This function is invoked once per event. The `hbook` ntuple filling mechanism will use the pointers provided in the `HBNAME` calls to retrieve the values of all items.

Although the working principle of filling the ntuples from a fixed-location existing data structure (`Fortran` `common` block) is straight-forward and generally appealing, it is not suitable for the generic unpacker. The data structure layout used by the unpacker cannot be described in this scheme. Bit-mask versions of zero-suppression are unheard of, and also the multi-level handling of items that logically belong together, must be flattened before they can be used by `hbook`[1].

To cope with this situation, each event to be written to the ntuple will be pre-staged to a fixed "virtual" structure in memory, with a layout suitable for `HBNAME` description. The structure is virtual as there exists no description of it as a `C` structure, its memory layout is calculated within the program at startup when handling the `--ntuple` option.

The unpacker program internally has a list of pointers to all variables (every item within an array separately) together with its name (handled as a path within the data structure tree). These are set up at startup by letting generated functions iterate over the structures. These pointers and name pairs are among others also used when setting up `unpack` $\rightarrow$ `raw` maps, and `raw` $\rightarrow$ `cal` transformations.

[1]The same arguments also hold for `land02` — the ntuple creation code was actually created there.

The handling of the `--ntuple=` option selects all data members that are wanted (as specified on the command line). The list of names and pointers is then sent to the ntuple-stager[2]. The stager sorts the entries based on names, to arrange "similar" items together, basically such that indexed variables with the same name come in order. This way, flattening of the data structures is performed, in some sense reordering the entries along another index than is used internally in the unpacker program.

The number of needed variables in the worst case, i.e. for a full event, is counted and memory space is allocated. Finally, the stager runs over the "virtual" array, producing the necessary strings and pointers and invokes `HBNAME` to make `hbook` believe that such an array exists.

For each event, the ntuple writer will iterate through a list of pairs of source and destination pointers to copy the data from the internal data structures to the virtual array. It also does the conversions necessary to handle bit-mask protected zero-suppressed arrays and creates index variables in the array to mark which index each entry originates from. As zero-suppression is strictly obeyed, the iteration time is proportional to the number of ejected items. Finally `HFNT` is called to let `hbook` store the data to file.

The capability of the ntuple stager and writer to create `root` trees is invoked by giving a `.root` file extension, or by using the `ROOT` option. The trees have the same layout as the `hbook` ntuples, i.e. the stager essentially behaves as if `h2root` had been used. Creation of more advanced trees is a future development. Only `root` files can be read with the `--in-tuple` option. Pass `hbook` files through `h2root` first to avoid indigestion.

6.1.1 Examples

To create a `file.ntu` ntuple with all `unpack` level variables, use the option:

```
--ntuple=UNPACK,file.ntu
```

To get the data organised by detector instead, fetch the `raw` level:

```
--ntuple=RAW,file.ntu
```

Many levels can be requested at the same time:

```
--ntuple=UNPACK,RAW,file.ntu
```

By explicitly naming some variables, only those will be included in the output file (the detector names in the following examples apply to the LAND setup, use the `--show-members` option to get a list of available variables):

```
--ntuple=RAW,POS,N,TFW,TOF,file.ntu
```

[2]The same information (names and pointers) is collected by land02, but in a different way, and sent to the same stager.

Or at the `unpack` level:

```
--ntuple=UNPACK,fastbus,camac,file.ntu
```

The writer can also be reversed, becoming a reader for input:

```
--in-tuple=UNPACK,file.root
```

6.2 External structure server

Along with the internal partition of the actual ntuple writing into a separate process (see Section 7.4) and the simultaneous cleaning of the interface for set-up and data transfer to the needed `hbook`/`root` libraries, an opportunity appeared to also deliver the virtual structure directly to other external programs. The idea is simple: for any requested ntuple output, a `C` header-file with a declaration of a structure corresponding to the virtual array can be generated. This can be included into any `C` program, allowing it to use the members in a normal fashion, see Listing 6.1. By calling a few functions provided by one lean library (based on a single source file in plain `C`), the external program connects to a `tcp/ip` server operated by a running UCESB unpacker, and fetches the data event-by-event into the structure. The program is free to do whatever it likes with the structure contents. It can even be a ROOT script or program.[3] This interface is most suitable for on-line monitoring or testing of ideas when one for some reason does not want to perform that task in a user-function of the unpacker itself.

6.2.1 Example

First the header file with the structure layout must be created:

```
--ntuple=UNPACK,STRUCT_HH,ext_struct.hh
```

The server is started by:

```
--ntuple=UNPACK,STRUCT,dummy
```

Following this (and compilation of the external program), start processing:

```
./my_external_program localhost
```

where `localhost` would be replaced by the `hostname` of the server machine. The protocol is unidirectional, thus also suitable for pipe-line operation:

```
... --ntuple=UNPACK,STRUCT,- | ./my_external_program -
```

The pipeline functionality could be used by another analysis system wishing

[3]The possibility to run a similar server with ROOT intrinsics have been investigated, but an easy and efficient solution supporting multiple clients was not found. Being independent also have other advantages.

```
typedef struct EXT_STR_h101_t
{
  // UNPACK
  uint32_t  TRIGGER;
  int32_t   EVENTNO;
  // RAW
  uint32_t  POS1_1E;
  uint32_t  POS1_1T;
  uint32_t  POS1_2E;
  uint32_t  POS1_2T;
  /* 10 items omitted ... */
  uint32_t  POS2_4E;
  uint32_t  POS2_4T;
  uint32_t  TFW;
  uint32_t  TFWI[32 /* TFW */];
  uint32_t  TFW1E[32 /* TFW */];
  uint32_t  TFW1T[32 /* TFW */];
  uint32_t  TFW2E[32 /* TFW */];
  uint32_t  TFW2T[32 /* TFW */];
} EXT_STR_h101;
```

```
typedef struct
  EXT_STR_h101_onion_t
{
  // UNPACK
  uint32_t  TRIGGER;
  int32_t   EVENTNO;
  // RAW
  struct {
    struct {
      uint32_t  E;
      uint32_t  T;
    } _[4];
  } POS[2];
  uint32_t  TFW;
  uint32_t  TFWI[32 /* TFW */];
  struct {
    uint32_t  E[32 /* TFW */];
    uint32_t  T[32 /* TFW */];
  } TFW_[2];
} EXT_STR_h101_onion;
```

Listing 6.1: Example of structure layout for use by external programs. The two structures have the same organisation in memory. The left has names directly corresponding to the `hbook`/`root` ntuple, while the right is generated with heuristics to combine named indices into structured arrays, for looping over the items.

to use the unpacker just to read raw data files, perhaps as a `fork`ed process, hiding the pipe-line.

The header file can also be generated without direct access to the program of a running server, by connecting with the `struct_writer` program (which is the same as runs the server):

```
$UCESB/hbook/struct_writer hostname --header=ext_struct.hh
```

It resides in the `hbook/` directory of the unpacker sources (denoted by `$UCESB`). It can also be used to run a proxy-server:

```
$UCESB/hbook/struct_writer hostname --server
```

The events from a server can be stored in `hbook` or `root` files by using the writers since all necessary information to set them up is available in the protocol data stream:

```
$UCESB/hbook/root_writer hostname --outfile=mytree.root
```

Finally, if the external program instead have been constructed to generate data, this can be passed as input to the UCESB process:

```
./my_external_generator | ...  --in-tuple=UNPACK,STRUCT,-
```

6.2.2 Implementation notes

For the user of the external program, the event-wise filling of the structure behaves as if the new data is simply copied into it each time. The exception is that zero-suppressed arrays are not touched except for the number of valid items given by the controlling variables in each event. Internally, the amount of data transferred over the network is additionally reduced by employing a byte-packaging favouring short encodings for small values, zeros and `NaN`s[4], at the same time handling the byte-swapping issue due to machines having different endianess.

The server handles multiple client connections efficiently by, analogously to Figure 7.1, employing a common buffering of outgoing data, although no separate thread is used in this case. The packed data for multiple events are stored into buffers of approximately the same size before being sent over the network to allow the bulk transmissions to handle larger chunks than individual events may provide. As the total amount of buffering is limited (currently 16 MiB), client connections that are too slow to handle all events will eventually skip chunks of events as some buffers are re-used before being transmitted to those clients. No losses occur when the connection is made as a `pipe`.

The server listens on two `tcp/ip` ports. The main socket only act as a port-mapper to the second sacrificial socket, bound to a random port number, and handling all data transmission. This way, there are rarely any connections open on the main port, allowing for easy restart of the server without having to wait for the dreaded “socket in use” due to the previous server having just torn down the connections on the main port. Works like a charm.

6.2.3 Interface library for the external program

The functions needed in an external program to either fetch or produce event-wise data through the generated structure are collected in a client library with prototypes in `$UCESB/hbook/ext_data_client.h`. This header is to be included by client programs:

```
#include "ext_data_client.h"
```

For the compiler (`$CC`) to find it, add the `hbook/` directory to the include file search path, usually:

```
$CC ...  -I $UCESB/hbook/
```

[4]`NaN`s are used to mark unset values of floating point variables and are therefore frequent at `cal` level (and higher).

And link (`$LD`) the object file of the client library with the program:

```
$LD ...  $UCESB/hbook/ext_data_client.o
```

Alternatively, include the source file in the compilation directly. In any case, it is not recommended to copy the header or source files of the client library directly into another directory, as they may be changed due to protocol extensions. Mismatches will be detected at run-time. Listing 6.2 shows a simple external program. More templates are in `$UCESB/hbook/example/`.

The following functions are available:

```
struct ext_data_client;
```

Forward declaration of a structure used as handle for a connection.

```
struct ext_data_client *ext_data_connect(const char *server);
```

Create and return a client context connected to `server`. Use `-` for `stdin`. Returns `NULL` on failure.

```
int ext_data_setup(struct ext_data_client *client,
                   const void *struct_layout_info,
                   size_t size_info,size_t size_buf);
```

Validate that the server provides data suitable for the intended structure (of size `size_buf`) by comparing with information in an ancillary structure also declared in the generated header. See Listing 6.2 for more details on usage. Returns 0 on success.

```
int ext_data_fetch_event(struct ext_data_client *client,
                         void *buf,size_t size);
```

Fetch one event into the structure pointed to by `buf`. Its `size` is only given for an extra check. Returns 1 for one fetched event, 0 on end-of-data and negative on error.

```
int ext_data_close(struct ext_data_client *client);
```

Close the connection and delete the context. Returns 0 on success.

```
void ext_data_rand_fill(void *buf,size_t size);
```

To help identify possible bugs in the user-code management of zero-suppressed arrays (reading invalid items), this function can be called to fill the entire structure with random bits before fetching events, making use of invalid data more likely to stand out and be noticed.

```
const char *ext_data_last_error(struct ext_data_client *client);
```

Returns a pointer to a string describing the most recent error (if any).

```
#include <stdlib.h>
#include <stdio.h>
#include "ext_data_client.h"

/* Change these, here or replace in the code. */

#define EXT_EVENT_STRUCT            EXT_STR_h101
#define EXT_EVENT_STRUCT_LAYOUT     EXT_STR_h101_layout
#define EXT_EVENT_STRUCT_LAYOUT_INIT EXT_STR_h101_LAYOUT_INIT

#include "ext_h101.h" /* The generated header. */

int main(int argc,char *argv[])
{
  struct ext_data_client *client;

  EXT_EVENT_STRUCT event;
  EXT_EVENT_STRUCT_LAYOUT event_layout =
    EXT_EVENT_STRUCT_LAYOUT_INIT;

  if (argc < 2) {
    fprintf (stderr,"No server name given, usage: %s SERVER\n",argv[0]);
    exit(1);
  }

  client = ext_data_connect_stderr(argv[1]); /* Connect. */

  if (client == NULL)
    exit(1);

  if (ext_data_setup_stderr(client,
                            &event_layout,sizeof(event_layout),
                            sizeof(event)))
    {
      for ( ; ; ) /* Handle events. */
        {
#ifdef BUGGY_CODE
          ext_data_rand_fill(&event,sizeof(event));
#endif
          /* Fetch the event. */

          if (!ext_data_fetch_event_stderr(client,&event,sizeof(event)))
            break;

          /* Do whatever is wanted with the data. */

          printf ("%10d: %2d\n",event.EVENTNO,event.TRIGGER);

          /* ... */
        }
    }

  ext_data_close_stderr(client);
  return 0;
}
```

Listing 6.2: Example of an external program for processing events fetched from a `struct` server.

The following functions are used when generating data:

```
struct ext_data_client *ext_data_open_out();
```

Create a client context for data output on `stdout`. To be followed by a call to `ext_data_setup`.

```
int ext_data_clear_event(struct ext_data_client *client,
                         void *buf,size_t size,
                         int clear_zzp_lists);
```

This function can be used to clear the structure `buf` before filling a new event. With non-zero argument `clear_zzp_lists` all items in zero-suppressed arrays are also cleared — a performance killer.

```
void ext_data_clear_zzp_lists(struct ext_data_client *client,
                              void *buf,void *item);
```

Instead of clearing all zero-suppressed arrays with the previous function, this routine can be used to clear the valid items of arrays associated with a controlling `item`, after having set it, but before (sparsely[5]) filling the array members.

```
int ext_data_write_event(struct ext_data_client *client,
                         void *buf,size_t size);
```

Pack the data from the structure `buf` for transmission.

```
int ext_data_flush_buffer(struct ext_data_client *client);
```

Data is normally sent when the internal transmit buffer becomes full. Immediate transmission can be forced with this function. Useful if the producer knows that it will need to wait for further input before generating the next event.

More detailed documentation is available in `hbook/ext_data_client.h`. These functions return error codes in `errno` and never litter `stderr` with any messages. To simplify writing of external programs where this is not a problem, easier-to-use wrappers with `_stderr` added to their names are also available. Those are used in Listing 6.2. The functions are also directly usable by `C++` programs, but wrapper `class`es do exist, along with shared library files useful for ROOT scripts.

[5]If all valid members of a structure or array are set in each event, clearing is not needed.

6.3 Watcher – the DAQscope

The unpacker provides a way to from a text-terminal quickly see if a bunch of channels have some data, and where within the range the data lies, see Figure 6.1. The display is made using the `ncurses` library and controlled by a command line option, making it easy to use even from remote locations. By using it with an on-line data source, it can be used to help approximately adjust delays and amplifications. Running it off-line, it can show the development of the channels during an entire experiment, like a movie, played in fast forward!

For each selected channel, the value of every processed event is added to a histogram. When the display condition occurs (usually due to an off-spill event) the histogram is rendered as one line of numbers representing the $\log_2$ of the counts in each bin, and dots representing empty bins. When the `WATCHER_EVENT_INFO_USER_FUNCTION` has provided trigger type information, each bin will be colour coded by the trigger dominating the bin content.

6.3.1 Examples

To invoke the DAQscope, specify some modules or detectors at the `unpack` and/or `raw` level to show:

```
--watcher=POS,PIN,PSP 2> /dev/null
```

The redirection `2> /dev/null` prevents cluttering of the display by the normal output of `ucesb`, including error messages. In case the watcher does not start at all, remove the redirection to inspect any error messages that may explain the cause.

To enforce a different sorting of the detectors than their order in the internal structures, a colon can be used to separate more specific detector requests:

```
--watcher=POS1-2_1-4T:POS1-2_1-4E
```

The histogram can be expanded around a certain interval:

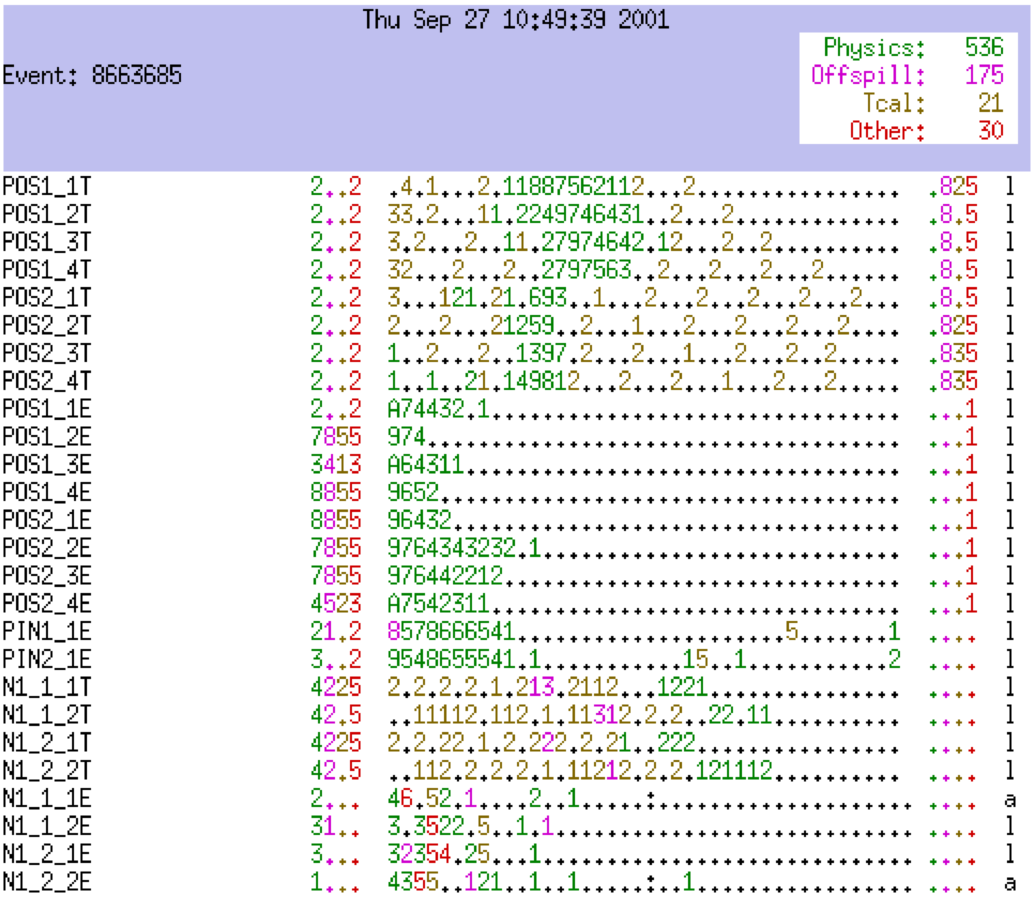
```
--watcher=MIN=0,MAX=1000,POS1-2_1-4E
```

To easily adjust the delays of e.g. time calibrator signals, the watcher can help by showing the currently “illuminated” range (the user code must provide the event-wise sampled value):

```
--watcher=RANGE,POS1_1-4T,N1_1-5_1-2T
```

For cases when no spill synchronisation is available, the maximum number of events or timeout values can be specified (overriding the large defaults):

```
--watcher=COUNT=1,TIMEOUT=2,vme_adc1
```



```
                           Thu Sep 27 10:49:39 2001
                                                        Physics:    536
Event: 8663685                                        Offspill:    175
                                                             Tcal:     21
                                                            Other:     30

POS1_1T                 2..2  .4.1...2.11887562112...2................  .825  l
POS1_2T                 2..2  33.2...11.2249746431..2...2.............  .8.5  l
POS1_3T                 2..2  3.2...2..11.27974642.12...2..2..........  .8.5  l
POS1_4T                 2..2  32...2...2..2797563..2...2...2...2......  .8.5  l
POS2_1T                 2..2  3...121.21.693..1...2...2...2...2...2...  .8.5  l
POS2_2T                 2..2  2...2...21259..2...1...2...2...2...2....  .825  l
POS2_3T                 2..2  1..2...2..1397.2...2...1...2...2..2.....  .835  l
POS2_4T                 2..2  1..1..21.149812...2...2...1...2...2.....  .835  l
POS1_1E                 2..2  A74432.1................................  ...1  l
POS1_2E                 7855  974.....................................  ...1  l
POS1_3E                 3413  A64311..................................  ...1  l
POS1_4E                 8855  9652....................................  ...1  l
POS2_1E                 8855  96432...................................  ...1  l
POS2_2E                 7855  9764343232.1............................  ...1  l
POS2_3E                 7855  976442212...............................  ...1  l
POS2_4E                 4523  A7542311................................  ...1  l
PIN1_1E                 21.2  8578666541.....................5.......1  ....  l
PIN2_1E                 3..2  9548655541.1..........15..1............2  ....  l
N1_1_1T                 4225  2.2.2.2.1.213.2112...1221...............  ....  l
N1_1_2T                 42.5  ..11112.112.1.11312.2.2..22.11..........  ....  l
N1_2_1T                 4225  2.2.22.1.2.222.2.21..222................  ....  l
N1_2_2T                 42.5  ..112.2.2.2.1.11212.2.2.121112..........  ....  l
N1_1_1E                 2...  46.52.1....2..1.....:....................  ....  a
N1_1_2E                 31..  3.3522.5..1.1............................  ....  l
N1_2_1E                 3...  32354.25...1.............................  ....  l
N1_2_2E                 1...  4355..121..1..1.....:..1.................  ....  a
```

Figure 6.1: Watcher showing data from the S245 experiment. Channels selected by: `--watcher=POS1-2_1-4T:POS1-2_1-4E:PIN1-2_1E:N1_1-2_1-2T:N1_1-2_1-2E`.

To the left, each signal's name is given. The first column shows the number of underflow conversions (per different event type), in the middle the full range of the channel is displayed, with the colour selected by the dominating contribution, and overflow is shown in the right column. The rightmost letter, `l`, `h`, `a`, gives the range in use (low, high, or auto).

The time calibrator (yellow) only generated signals at 11 locations (with 10 ns spacing) as can be seen in the POS time channels. The cosmics (offspill, magenta) and physics (green) in LAND (N) are not overlapping in the time range, which will require a good TDC calibration, as the timing is synchronised by cosmic muons. Preventing such things from passing unnoticed during data-taking is one of the watcher's reasons for being.

6.4 Correlation plots

A `ucesb` correlation plot is a 2-dimensional grey-scale triangular picture showing how often each pair of channels have signals in the same event. Each detector channel, times and amplitudes individually, occupy one index on both the x and y axis. The intensity of every pixel in the upper triangle shows the number of times the two channels that represent its intersecting indices have had data in the same event — the darker the more often. The typical use is for detectors where neighbouring channels are expected to sometimes (or quite often) give a signal for the same particle passing through the detector.

An example is shown in Figure 6.2, where both the correlations between the two PM tubes of the same paddle can be seen as dots near the diagonal, as well as the correlation between TDC and ADC values for the same PM tube channel as a diagonal in the upper right square of the picture. Even a rough picture of the hit pattern over the detector is seen in the horizontal vs. vertical TDC correlations (t_x vs t_y).

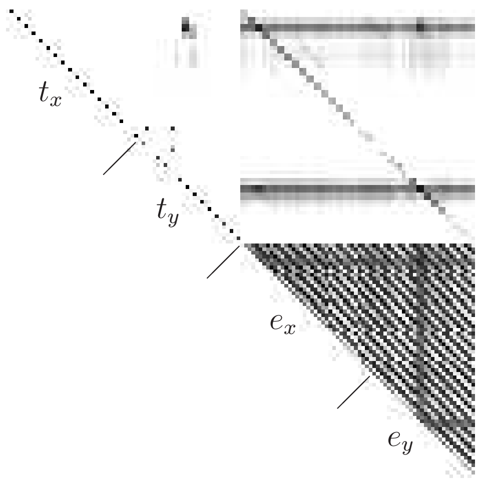


Figure 6.2: TFW detector internal correlations. Each paddle has two PM tubes, here ordered pairwise.

Figure 6.3 shows an example with more channels (the entire LAND detector). While often rather easy to spot when some mapping is wrong (see inset of Figure 6.3), it is harder to figure out how to correct it, in which case the documentation on how the detector was mounted and cabled has to be consulted to correct the problem.

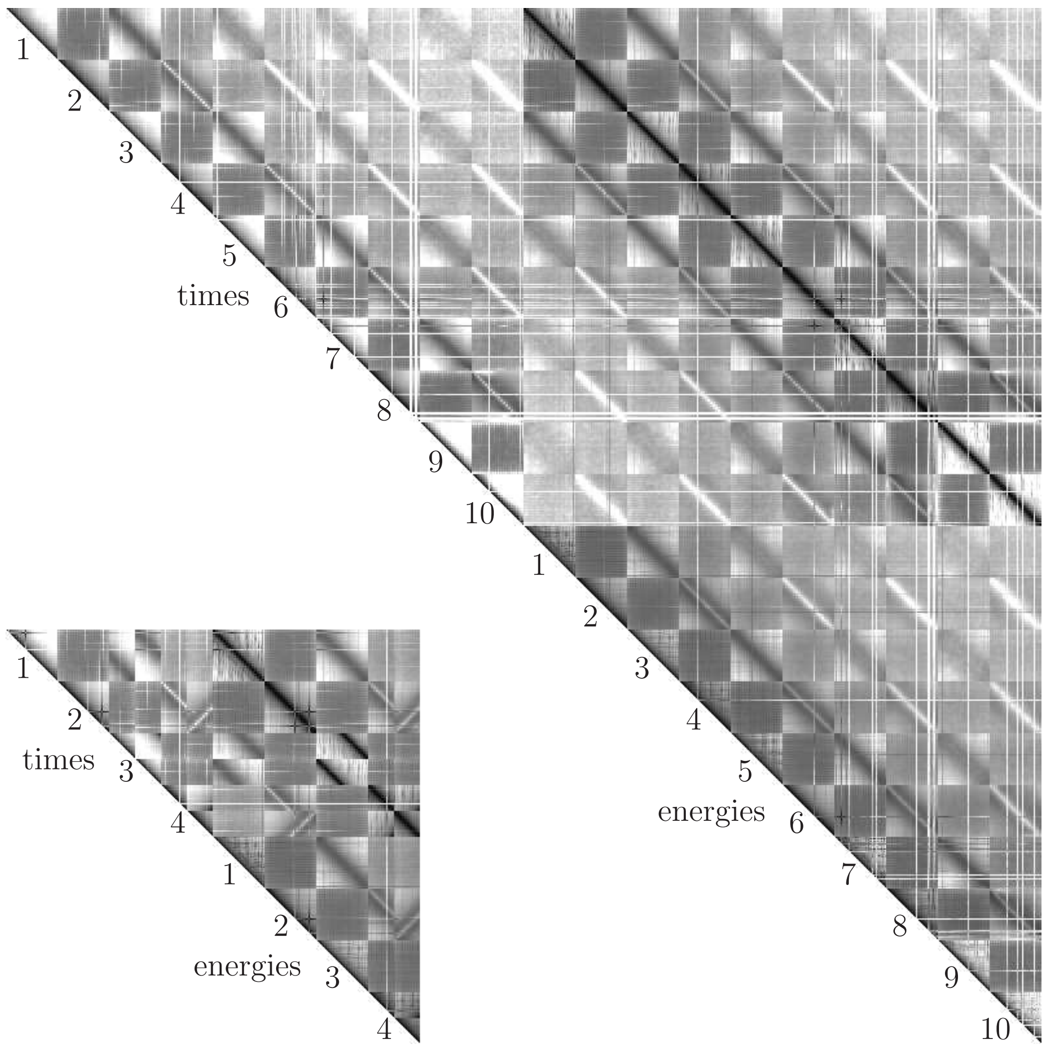
6.4.1 Examples

The TFW correlations in Figure 6.2 were made using:

```
--corr=TFW1-32_1-2T:TFW1-32_1-2E,tfw.png
```

and the picture of LAND cosmics in Figure 6.3 was created by:

```
--corr=N1-10_1-20_1-2T:N1-10_1-20_1-2T,n_cosm.png
```



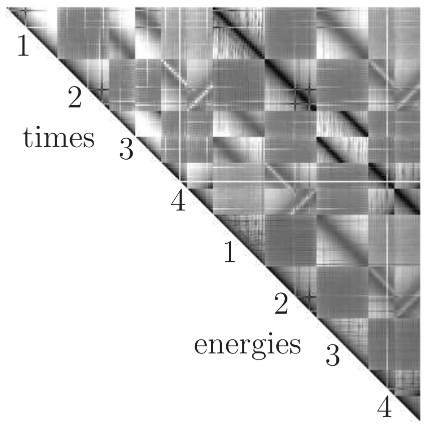


Figure 6.3: Coincidences between the read-out channels of the LAND detector for a cosmic muon calibration run. The even numbered planes have horizontal paddles. The lack of correlated data (diagonals) between those planes comes from the fact that such correlations would require cosmic muons with almost horizontal tracks, which are rare as they originate in the sky. The missing vertical and horizontal lines are not printing artefacts — they represent broken channels.

The smaller picture shows the first four planes with some intentional channel mapping errors. The first plane has had two time channels exchanged, the second plane has had two PM tubes (time and energy) swapped. The first and second halves of third plane have been exchanged, while the second half of the fourth plane is reversed.

6.4.2 Implementation

Each channel to be shown in a plot is given an index, becoming the channel's row and column number. For each event, a list with the indices of channels with non-zero data is constructed. Then, making a double loop with i and j, $j > i$, over the list of indices, the counter of correlations of each location, $s_{i,j}$, is increased.

When all events have been processed, the matrix is converted into a picture. The intensity of each pixel $I_{i,j}$ is the logarithm of the counts $s_{i,j}$, normalised to the total number of hits s_i and s_j in the contributing channels, and an overall normalisation based on the total number of correlations S, the number of channels n and the number of events E,

$$I_{i,j} = \ln\left(\frac{s_{i,j}}{\sqrt{s_i s_j}}\right) / \ln\left(\frac{S}{nE}\right). \tag{6.1}$$

To prevent the swamping of the real (physical) correlations by background noise, it is important for energy channels to be properly zero-suppressed, i.e. not include any pedestal or time calibrator events when creating the plots. The pictures are generated as a raw bitmap, piped to a forked `convert` process, that creates a file in any common graphics file format.

The class handling correlation plots (`corr_plot_dense`) can also be used from user code to make more elaborate selections. A version suitable for sparse channel occupation (`corr_plot`) is more efficient in cases where the channel numbers in use are selected from large ranges, but are grouped with many unused numbers in-between. This is used in the `esn` program, see Appendix A.

The correlation plots have been used successfully to verify that the cable mappings of detectors with several thousand channels are correct. The biggest being a RICH detector with 4096 channels (64 pixels squared), where each scan lines gives neighbour correlations, and also between closely neighbouring lines. The plots have also been used with the EuroSuperNova wire chamber detectors, having a few thousand wires. Here, the sparse mode came to good use, as the data format itself allowed for 64k wires, out of which some 4000 are in use, sparsely using some regions of the possible wire number encodings.

As noted above, obtaining reasonable correlation plots is sometimes difficult. ADC channels require a reasonable threshold cut value to be usable in the correlation plots. It could for future development be advantageous to couple a pedestal determination routine to provide cut values as a pre-filter to the correlation plot, such that only non-noise readings are counted.

Chapter 7

Internals

Some of the internal workings of the program may not be of general interest, but some ideas that have been used deserve to be explained. It should be noted that most of these ideas were not included in the design from the beginning, but have matured into existence.

The primary goal is to keep the code size down — avoiding code duplication at almost any cost — letting the compiler and helper scripts and programs do the job instead! The second ambition is to use as efficient data storage schemes (and thereby algorithms) as possible and feasible. That the program is *fast* is not by chance — it comes from hard work and discipline. This fortunately only seldom interferes with the primary goal.

One thing has led to the other. Seemingly random cleanup and consolidation actions have paved the way for entirely new capabilities.

7.1 Input stage

One example of evolving developments is the input buffering stage. The file format reader originally did its own I/O, even opening the files. An additional layer was introduced, separating the file handling and I/O, making it possible to also easily read data from a pipe, filled by a separate decompressor process. This separation lowered the threshold for implementing readers for more file formats considerably.

The practice of by default reading data directly from `mmap`ped file buffers in itself gives a quite noticeable speed boost for some usage cases. Emulating it with internal buffers when necessary[1] in addition enabled decoupling of forked decompressor processes[2], allowing them to work at full speed

[1] A `pipe` (or network socket) cannot be `mmap`ped.

[2] The `fork` system call creates a new process, which in turn can `exec`ute a decompression

on multi-processor systems. These buffering mechanisms were then re-used when implementing TCP transports reading data directly from the DAQ.

7.1.1 Zero copy approach

The analysis processing of experimental data can quite often be rather quick, such that any I/O overhead will be responsible for a large part of the total computer time spent.

Instead of reading files with the `read()` system call, one may directly map the kernel memory representing (part of) a file into the process virtual memory with `mmap()`, in read-only mode. In this case, it will not be necessary for the kernel to copy the data to user-space, saving one loop over it. This is particularly useful in cases where events are skipped. The memory-mapped window of the file is moved as the processing of the data advances[3].

7.1.2 Emulating `mmap`

By using the `mmap` approach, the individual file format interpreters are simplified, as they no longer have to worry about how to fetch the data into buffers or deal with errors. They only have to call two functions:

`int map_range(off_t start,off_t end,buf_chunk chunks[2])`

> Which ensures that the data in the specified range of the file is available in memory somewhere, and return a pointer to the start of it. For `mmap`able cases, this translates directly.
>
> Since several memory map regions can be active simultaneously or the circular internal buffer could wrap in the middle of the requested range, it is not enough to just return one pointer. Instead, `map_range` will fill out `buf_chunk` entries with each fragment (pointer and length) and return the number of fragments used, `0` indicating failure. Normally one fragment is needed, and two are enough in the worst case, since the mapped regions or the size of the internal circular buffer are much larger than any individual record requested.

`void release_to(off_t done)`

> Is called to allow the input subsystem to reclaim (or re-use) any memory associated with data up to the given point in the input file.

program, passing the data to the original program via a `pipe`.

[3]The entire data-files cannot be mapped into virtual memory at once. On a 32-bit machine, addressable memory is only 4 GB. In cases when several GB large input files are open simultaneously, e.g. for merging, this would not suffice.

For the cases when a native `mmap` is not possible (i.e. all situations except a direct `file://` input), an internal buffer will be employed. Thus any I/O-related error handling can be located at one place. Reading compressed files (e.g. `.gz` or `.bz2`) is also done once for all different supported data file formats.

7.1.3 Subevent copying

The subevent unpacking routines use pointers directly into the `mmap`ped areas or buffers whenever possible. To not have to deal with pointers to subevent data regions that are 'split', whenever a subevent is fragmented over several buffers of the input data, the subevent data will be copied to one contiguous memory area. As only the fragmented subevent(s) within a fragmented event are given this treatment, and byte-swapping is made on the fly, much copying is avoided.

7.2 MBS-like TCP server, stream/transport

The unpacker can not only read, but also propagate raw data to other analysis clients by using the (simple) MBS stream and transport network protocols.

Although data would usually be read over TCP from a running DAQ and relayed, the program could equally well serve data from files. The `empty` program (see Appendix B) is especially useful as a data relay, as it does not care about the contents of the subevents and just passes them along. The major use of this feature is as an event server during a running experiment. The program (running on a computer with much network bandwidth) connects as the only client to the DAQ, and other analysis clients in turn connect to it, leaving the DAQ alone. The unpacker can also do some event rewriting or selection if necessary.

By decoupling both the input and output stages in separate threads from the main event processing, the network can be used optimally (to the limit of serialisation induced by the protocols in use). The CPU load is small for all the threads, at least when running as the `empty` program, i.e. without data processing.

7.2.1 Server in one thread using `select`

The data server runs in one separate thread, driven by the `select()` I/O multiplexer, and is responsible to accept client connections and serve them data. New data blocks are provided by the (normal) processing thread. To

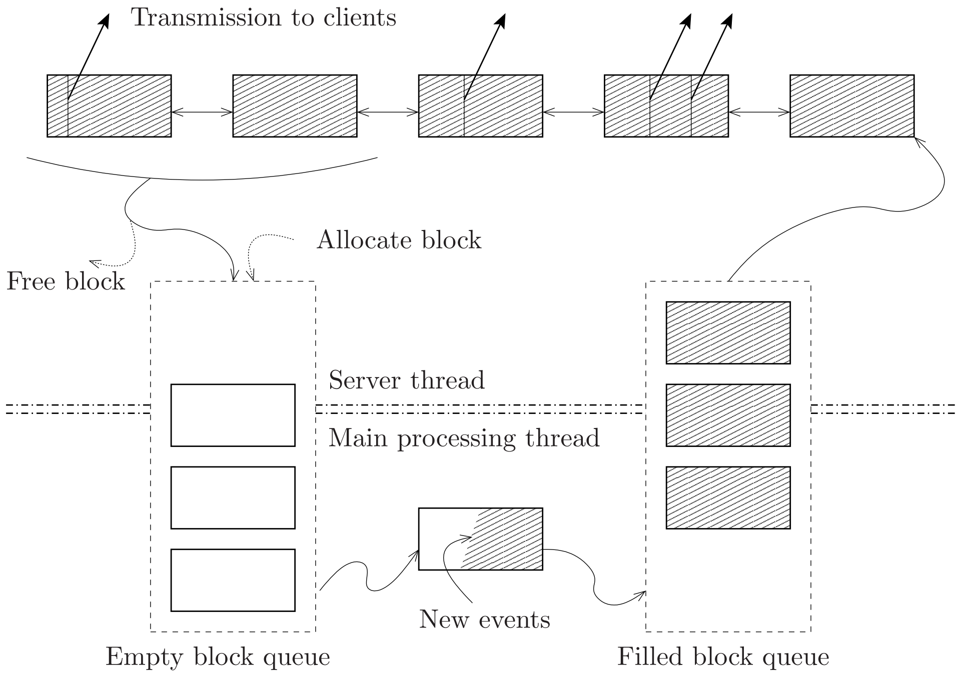


Figure 7.1: Buffer re-cycling between the server and producer threads. The main processing thread fills empty buffers with data and inserts them into filled block queue. The server thread inserts the filled buffers into a list, through which each client connection proceeds individually. Blocks are allocated or freed by the server thread as the empty block queue becomes empty or full.

avoid reallocations[4], old blocks are recirculated, much in the same way as — and inspired by — the MBS. Figure 7.1 sketches the data flow.

The empty and freshly filled data blocks are transported between the two threads using two circular queues, and the only blocking[5] required between the two threads occur when the queues become either full or empty. For simplicity, all memory management, in particular allocation of new blocks is done within the server. This way, only that thread needs to ensure that the maximum amount of memory used does not exceed the specified limit. Each time the processing thread needs a new block, if memory usage is still below the limit, it will be freshly allocated. If not, the oldest block in the send

[4]Bad, mostly because for these large sizes memory may easily be returned to the OS, and then new, freshly zeroes pages are required again, but also since memory management has to be synchronous between the threads.

[5]Multiple threads that may execute concurrently must carefully regulate (lock) the access to shared resources, in this case to not dequeue from an empty queue.

chain that is currently not used as a source is removed from the chain and reused. That block of data is then lost for all clients that currently receive data from even older blocks, but this just constitutes the dropping needed, to make those (slow) clients catch up.

Each newly filled block is put at the end of the send block list. It remains on the list until the last client has gotten the data (or it is prematurely reclaimed due to shortage of blocks). Each client connection only needs to keep track of which block of the send chain, together with an offset within that block, it is currently sending from. Each time `select` tells that an outgoing socket is ready to accept further data, more is written. As the network sockets are operated non-blocking, the server thread is never stalled talking to any client. When a client is done with one block, it moves to the next in the linked list. If it reaches the front, it will wait until another block becomes available from the processing thread. Other communication with the clients (other protocol overhead) is also performed in non-blocking mode.

When a block is no longer needed by any client, it is recycled onto the free block queue, for eventual use by the processing thread, or deallocated, if that queue is full.

Performance considerations: stream vs transport server

The stream and transport server/client protocol has only small differences. While the stream server always deliver full streams (i.e. a bunch of buffers, above called blocks), the transport server does the delivery in (non-empty) buffers. The advantage of only transmitting non-empty buffers would surface only if periodic flushing would be implemented. However, the protocol for the stream server also requires a token from the client before each new stream is sent. Depending on if the client itself sends a request for another stream before it has consumed the previous or not[6], this may incur a serialisation, such that for each stream delivered, one network packet round-trip time is lost.

It is recommended to use the transport protocol when reading from the `ucesb` data server, as the stream protocol communication offers no advantage. As a relay server, the `ucesb` program should however itself read data from the DAQ using the stream protocol, as that is non-blocking at the DAQ side, ensuring that a malfunction cannot halt the DAQ.

[6]The `ucesb` stream client does not send the request in advance.

Missing feature: periodic flushing

In cases where the data rate is small, or when the DAQ delivers the data using delayed event building, i.e. in bunches after each spill, it would be nice to ensure that no data lingers around for too long in any link of the transport chain. The MBS can ensure this with data flush timers.

The same could be done in the `ucesb` data server, i.e. when no buffer has been emitted for the last n seconds, a flush could be requested. However, due to the nature of how data is "pushed" from the processing thread to the server thread, the flush would have to originate in the processing thread. As that is completely data (event) driven — generally waiting deep down in the event reading code for new data at times when flushing is needed (i.e. when no events are arriving) — such a timer would currently not insert cleanly into the code. Flush functionality should come for free as threaded processing of events (see Section 7.3.2 below) becomes functional.

7.3 Multi-threaded operation

The use of several threads processing data can speed up the operation, as seen earlier even on a single-processor CPU by decoupling data in- and output. The following describes work-in-progress towards making further parallelisation possible.

7.3.1 Input system

When reading data over the network when a separate read thread is not employed, the input will be (partially) serialised with the analysis, as the buffers only will request more data when they become empty. Depending on the protocols involved, this usually means that at most one partial buffer is outstanding at a time, waiting in the OS network buffers. The next buffer will not even start to arrive before the request has propagated through the network round-trip time. These delays add directly to the execution time.

7.3.2 Event processing

With the advent of general availability of multi-core processors[7], it is in some cases beneficial to make the analysis tools able to process events in parallel.

[7]Computing power is since a few years growing horizontally — expanding the number of processing cores — instead of as before vertically — increasing the clock speed of the processors.

This line of operation is often far from necessary though. In most cases, one may just as well run several instances of the program in parallel, each processing one part of the data, and then combine the results at the end. The results of analysis, like histograms, are well suited for this.

Minimal impact and memory allocations

The goal is to provide access to (and make use of) the parallel capabilities of modern processors (more cores but generally lower clock speeds), while not requiring the user code to be more knowledgeable of how to deal with the associated constraints than absolutely necessary.

The most important aspect of such a “small-scale” multi-processing approach is the management of memory. As all threads run in the same process, and use the same virtual memory space, any memory allocation must by necessity be locking between the different executing threads. As long as the occasions to do memory allocations are few, this is no problem. If invoking programming primitives that require process memory allocations several times per event, the number of usefully executing threads would be few, since they would spend their time waiting for each other.

The idea is to make it possible to let the various processing stages of Figure 2.1 operate mostly independently.

Circular memory pools

To prevent memory management from becoming a bottle-neck, each processing thread employs a circular memory pool, from which only it can allocate memory. When the processing it performs (either unpacking or some later reconstruction) requires an array of memory that is not part of the statically allocated parts of an event structure, the memory will be retrieved from the end of the circular pool. Memory is only returned to the pool by the event retirement mechanism. As all events are processed in order both globally (i.e. read from file, and eventually retired) and within each thread, it is enough for the retirement mechanism to advance the counter that marks the end of free memory in the pool where it was allocated to release it. When an event is retired, all pointers to such dynamically allocated pool memory within it become invalid.

In many ways, this overall scheme of processing events (reading/decoding the events, performing the actual tasks associated with them, and retiring them) is similar to how out-of-order processor cores execute instructions, as found in high-end x86 processors since a decade.

Current status

With multi-threaded processing enabled, the code compiles and can run the unpacking stage. Mapping of data to the `raw` level, as well as ntuple generation or other analysis is not yet functional. The missing functionality lies in that with multi-threaded event processing, the program will have many events "in-flight" simultaneously, which requires several instances of the data structure holding an event. To cope with that, the numerous companion structures holding various pointers and offsets to within the event structure must be equipped with an additional base offset.

The separate threads used for the input stage and output stages (event server) are fully operational. They handle data before and after it is considered event-wise, respectively.

7.4 Data multiplexer

By using the various input and output options — controllable directly with the command-line — a UCESB unpacker is easily used as remote event server during experiments and for various data stream surgery, on-line as well as off-line: picking them apart and merging data flows from multiple event builders. Figure 7.2 shows the available data-paths entering and leaving UCESB. Of interest during (software) experiment preparations is the emulation of a running DAQ by the ability to feed data from raw data files into other on-line analysis tools using the built-in data server[8].

Handling of `hbook/root/struct` writing and reading is implemented as separate processes. These separate programs, transparently invoked, share the same source. In addition to the external library glue, they also implement a light-weight protocol to communicate with UCESB and LAND02 using a shared memory segment[9] . The biggest advantage is that only this source needs to be compiled together with either the `cernlib` or `root` libraries. The other advantage is that the bulk of the CPU load due to ntuple writing is off-loaded to another thread of execution.

[8]Functionality to limit the delivered data rates in this scenario is still missing. This would help mimic realistic time-structures of an actual experiment, with spill-pauses etc.

[9]With transparent fall-back to using `pipe`s when `shm` setup fails.

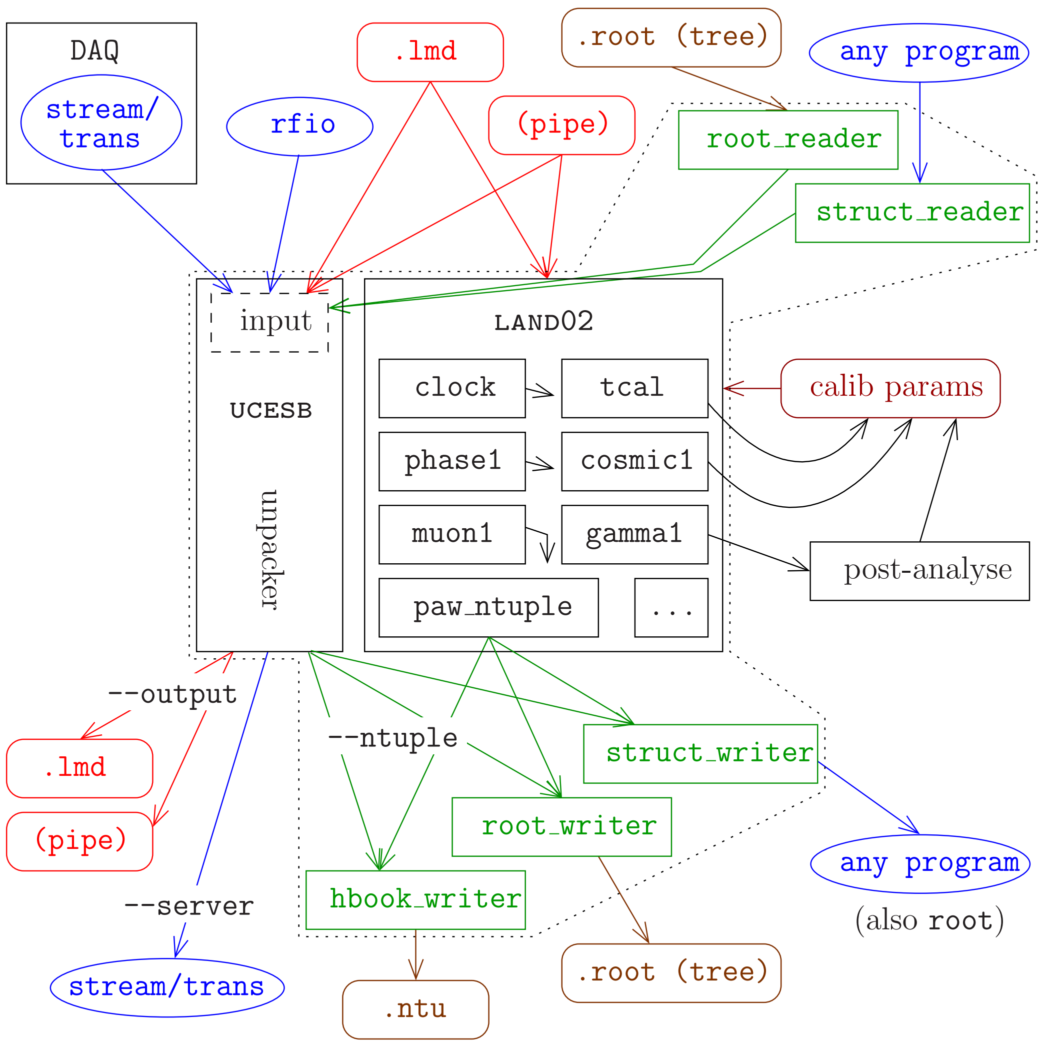


Figure 7.2: Octopus data-hauling routes available with UCESB unpackers. The DAQ connections (stream and transport protocols) are currently only available with LMD data formats. (pipe) denotes the capability to build command-line pipe-lines through `stdin` and `stdout`. The schematic includes LAND02 as this shares the code for the `ntuple`/`root`/`struct` output and input stages. It also serves to show the flow of calibration information when analysis is performed. The input capability for `root`/`struct` data will currently only be useful for programs doing some sort of analysis (like LAND02), but may become interesting once the unpacking process can be reversed (see note in next Chapter).

Chapter 8

Outlook

To say that a computer program is finished is only rarely true, as there is always one more bug to fix, and some new little feature to add.

The `ucesb` unpacker has reached a point where it spans the gap intended, that of unpacking data from raw data files and inspect it as well as generating ntuples.

It supports the creation of various user functions to do preliminary investigations of data collected in an experiment. Although one could even think of writing an entire analysis (from raw data to physical momenta) within the unpacker, this would be outside the intended scope. The support for handling calibration parameters is limited, and none at all is provided for handling parameters that vary during the course of an experiment.

Nonetheless, the program has successfully been used in LAND experiments[1] for monitoring and some on-line data transport. It is also used as a preprocessor for the raw data of some experiments, in order to avoid having the real analysis program deal with unnecessarily verbose formats. It has unpacked data from a few experiments at ISOLDE, as well as assisted the author in the (mostly private) playing with some data from the beautiful BBS-ESN detector system at KVI.

The set of items that could be improved is however not empty:

- Support for more file formats. Requests?
- Functional multi-threaded operation.
- Consider changing the command-line option handling to use `getopt()`.

[1]A heart-warming rumour has reached the author in that something like "What did we do before this existed?" was expressed during one beam time, probably related to the `watcher` functionality.

- Labelling of (groups of) channels in the correlation plots.
- Option for pedestal/threshold determination (on behalf of the correlation plots).
- Support a limited form of re-cabling during experiment, i.e. time-dependent mappings, unpack specifications and parameters[2].
- Work is in progress to allow "reversal" of the unpacking step, i.e. pack raw data files from simulated data. This would be useful for development and testing of analysis software by letting data flow from the reversed `ntuple` writer (reader) to the output stage in Figure 2.1.
- With trigger-less DAQ systems beginning to become popular, it would be interesting to apply the techniques for simple generation of apt and fast code for unpacking and mapping to the sorting and coincidence-building needed in on- and off-line event selection routines (software triggers) eating time-stamped data.

8.1 `land02` relationship

Having been developed almost simultaneously with, often in tandem, and performing many similar tasks, it is not surprising that the `ucesb` unpacker uses many similar designs as the more specialised `land02` software, as well as directly sharing some code:

- The ntuple dumper is common. In neither case, `land02` nor `ucesb`, it knows about the data structures themselves — it instead uses a lot of pointers, but it does know enough on how to handle the relevant control variables of zero-suppressed arrays: bit-fields, counts and indices.
- The input stage is inherited from `land02`, and actually has become more powerful. As such, it should be adapted for common use again.
- The same applies for the watcher, although it is more dubious if the watcher functionality inside `land02` is needed at all any longer, given the capabilities of the complementary `ucesb` based unpackers set up for DAQ monitoring of each experiment.

[2]A reasonable limit would be that the structures must stay the same (a superset of the parts needed for various file sets), but unpack functions and mappings may vary.

Glossary

ADC	Amplitude-to-digital converter. Measures the maximum amplitude of detector signals during a gate. Commonly used for semiconductor detectors, e.g. silicon diodes.
Channel	The word has two ambiguous usages. a) One detector or DAM channel = a read-out channel. b) The digital channels of one DAM channel (usually 4096), corresponding to the bins of a histogram of that read-out channel.

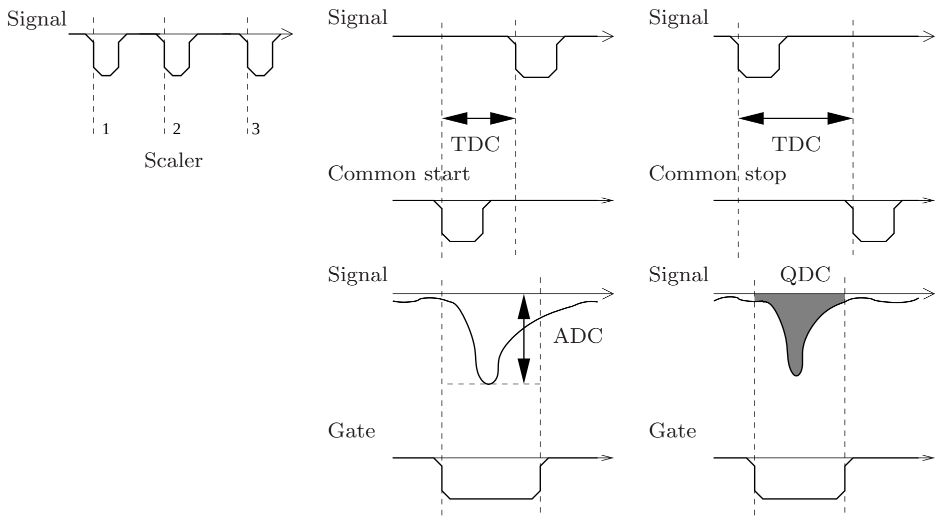


Figure 8.1: Basic DAM modules. From left to right on the first line: a scaler counting the number of logical signals and two TDCs operating in common start and common stop mode, measuring time interval between logical signals. On the second line, an ADC measuring the highest pulse amplitude (negative signal) during a gate, and a QDC integrating the signal during a gate.

CVS	Concurrent versions system. Revision control system.
DAM	Digital acquisition module. See Figure 8.1 for examples.
DAQ	Data acquisition. The scoping of this word varies wildly — between only the program controlling the data read-out, to the entire read-out electronics, hardware as well as software.
Dead-time	Time during which another event cannot be accepted, due to one or more parts of the DAQ being busy, usually processing a previous event.
DSSSD	Double-sided silicon strip detector.
ESN	Eurosupernova, spectrometer detector setup at KVI, among others, measuring the (d,^{2}He) reaction on stable targets.
Event	The occurrence of each ion passing through a setup (possibly reacting), together with the data recorded from the detectors, is an event.
Gate	Signal determining when the inputs of a DAM module are open. Usually derived from the master start signal.
GSI	Gesellschaft für Schwerionenforschung.
HADES	High Acceptance Di-Electron Spectrometer, experimental setup at GSI investigating hadron properties inside nuclear matter.
IS430	Experiment to perform elastic and inelastic scattering of ^{11}Be on deuterium at REX-ISOLDE, CERN, August-September 2005.
IS446	Experiment at REX-ISOLDE, investigating the $^8\mathrm{Li(d,p)}^9\mathrm{Li}$ reaction, August-September 2006.
ISOL	Isotope Separator On-Line. Method for producing exotic isotopes by bombarding a heavy target, thereby fragmenting it and then extracting the produced isotopes by diffusion into an ion source. The low-energy beams of radioactive nuclei are mass separated by a magnetic spectrometer.
ISOLDE	ISOL facility located at CERN, *on* the Swiss-French border, with the PS-booster bombarding the exotic isotope production targets with 1.4 GeV protons.
KVI	Kernfysisch Versneller Instituut, research laboratory situated north of Groningen, the Netherlands.
LAND	Large Area Neutron Detector. $2 \times 2 \times 1$ m^3 of sandwiched iron and scintillator, arranged in 10 planes with 20 paddles each, alternatingly oriented vertically and horizontally.
MBS	Multi Branch System, a general DAQ framework actively developed at GSI.

Parser	Function that reads and interprets text input, generating an internal representation of the contents.
PM tube	Photo-multiplier tube. Detects photons (usually produced in a scintillator) by converting them into electrons and amplifies the electrical output signal by an avalanche of repeatedly multiplying the electrons.
QDC	Charge-to-digital converter. Integrates and measures the current of detector signals during a gate. Commonly used for signals from PM tubes.
REX-ISOLDE	Radioactive Beam EXperiment at ISOLDE. Post-accelerates exotic isotopes after production and separation.
RFIO	Remote file input/output. Yet another protocol for accessing files over a network. Designed to allow for efficient streaming of large data files.
RICH	Ring imaging Čerenkov detector. Uses the effect analogous to a sonic boom, whereby charged particles traversing a material faster than the effective speed of light generates a photonic shock wave. The photons are emitted in a cone, whose opening angle is determined by the particle velocity.
S287	Experiment at the ALADiN-LAND setup in Cave C at GSI, April-May 2005, investigating the dipole strength in Ni isotopes.
Scaler	Counts the number of pulses. Used to see how often a detector channel or other signal fires.
Scintillator	Organic or inorganic compound which has a high probability of emitting photons (around the visible spectrum) after being excited by a passing ion.
Slow control	Computer assisted adjustment and recording of the parameters controlling an experimental setup. Replacement for potentiometers and screw-drivers.
SPALLADIN	Exclusive measurement spallation experiment setup around ALADiN.
TCP	Transmission Control Protocol. Layer in a network communications protocol stack ensuring correct delivery of data.
TDC	Time-to-digital converter. Measures the time between a common start or stop signal and individual detector signals.
TFW	Time of flight wall, a large segmented plastic scintillator detector. Part of the LAND setup.
Trigger logic	Part of the experimental electronics that, based on fast coincidences from the detectors, decide when an event has happened

	and if it should be recorded. It is also assuring the dead-time blocking.
VME	VERSA module eurocard, a databus, commonly used in industry for computing and control applications.
`land02`	Suite of programs to calibrate and reconstruct data collected with the ALADiN-LAND setup.

Bibliography

[1] *Physics Analysis Workstation*
`http://wwwasd.web.cern.ch/wwwasd/paw/` (2008-05-04)

[2] *The ROOT system homepage* `http://root.cern.ch` (2008-05-04)

[3] *CERN Program Library*
`http://cernlib.web.cern.ch/cernlib/` (2008-05-04)

[4] H.T. Johansson, *The DAQ always runs*, Licentiate Thesis, Göteborg, 2007. `http://fy.chalmers.se/~f96hajo/old/lic/home.html` (2008-01-30).

[5] `http://www-wnt.gsi.de/daq` (2008-04-07)

[6] *Tape Server — data block format*
`http://ns.ph.liv.ac.uk/MTsort-manual/TSformat.html`
(2008-04-08)

[7] *NSF Data Acquisition System, Event format [Sorting + Storage]*, Liverpool University, June 2001.
`http://ns.ph.liv.ac.uk/MTsort-manual/eventformat.ps`
(2008-04-08)

[8] B.W. Kolb, M. Münch, *HADES raw data format*, 2004-02-23.
`http://www-hades.gsi.de/daq/raw_data_format16.html` (2008-04-08)

[9] *HBOOK - H1Ntuples*
`http://wwwasdoc.web.cern.ch/wwwasdoc/hbook/H1Ntuples.html`
(2010-01-09)

[10] `http://web.archive.org/web/20070807005108rn_1/www.lecroy.com/lrs/`
(2010-01-05)

[11] `http://www.caen.it/nuclear/product.php?mod=V775` $\rightarrow$ Manual
(2008-04-08)

Appendix A

Working unpacker examples

Within the source distribution, a few different unpackers can be found. Most of them were created to make some quick investigation of some aspect of the data produced by an experiment or test setup. They are *not* full-fledged analysis tools, but may serve as examples of the various techniques described in this report.

`empty` Any experiment (LMD data).

Minimal unpacker, does no data checking. Useful to setup an event server.

`cros3` LAND setup (proton drift chambers).

Uses a hand-written (external) unpacker and specialised data structures. The actual unpacking is even performed by wrapped code, intended to run within the DAQ itself to condense the data. Also shows how to add a subevent to an output data stream (created by the data condenser).

`esn` KVI – EuroSuperNova detector.

Full-fledged external unpacker and external data structures, reads from PAX format files. The detectors contain several thousand channels, some of them with multi-hit capabilities. Specially crafted data structures are used to accommodate this. Uses the correlation plot classes in the user function to check wire map ordering. (Playground for testing an alternative (unfinished) VDC track recognition code).

`hades` Only unpacking data from the RICH detector of said setup.

`hacky` –

Minimal external unpacker, just keeping data pointers from the unpacking for later inspection in a user function.

`hirich` Spalladin RICH detector.

Testing compression of data from the HIRICH detector.

`i123` IGISOL decay setup.

Data collected with Daresbury MIDAS, stored in EBYE format.

`is430_05` REX-ISOLDE reaction setup.

Experiment was performed with multi-event data collection.

`is446` REX-ISOLDE reaction setup.

Correlation plots used in user function. See Appendix D.

`labbet1` Test setup with Daresbury MIDAS.

Unpacking of data in EBYE format.

`land` LAND – S245 experiment.

Use of external functions to control conditional unpacking within the generated unpacker code. Function to classify events for the watcher.

`madrid` Some setup at CSIC, Madrid.

Unpacking of data in EBYE format. TDCs with multi-hit capability in use.

`rpc2006` RPC test setup.

`siderem01` LAND setup (silicon micro-strip detectors).

Large user function and other routines comprising a test site for learning how to handle the data from the many-channel (640+384) silicon micro-strip detectors. Pedestal determination, common-mode noise rejection (baseline variation), Huffman compression. . .

`dets` –

Not an unpacker in itself. The directory contains common code (also external unpackers) for the `cros3` and `siderem01` programs, which is also shared with unpackers created for specific experiments.

Appendix B

A minimal unpacker: `empty`

This example shows a minimal unpacker, which only uses the capabilities to handle the events in raw data files, as no actual unpacking is performed. It can be useful for creating a simple event relay server, together with merging capabilities (based on the event numbers in the event headers) — it will not bother about unpack errors, only file packaging troubles.

The kind of input file format to be handled is declared as a compile option:

Listing B.1: `empty/makefile_additional.inc`

```
# -*- Makefile -*-

USE_LMD_INPUT=1
```

The data format specification is simple, as all subevents are silently ignored:

Listing B.2: `empty/empty.spec`

```
// -*- C++ -*-

EVENT
{
  ignore_unknown_subevent;
}
```

That's it! Agreed, this was a bit cheating, since this unpacker comes from within the source tree and is made with the main `Makefile`. The `hacky/` example directory has a `Makefile` showing what is needed to build outside the `unpacker/` tree.

Appendix C

Multi-event unpacker: `is430_05`

This example highlights parts of an unpacker for an experiment using a multi-event read-out (see [4], Front page and Section 5.5). The experiment used two telescopes of thin double-sided silicon strip detectors with perpendicular segmentation at the front and back sides for ΔE and position detection, followed by a thicker silicon E-detector. The telescope detectors were located off the beam axis to detect the light particles from the inverse kinematics reactions of ^{11}Be impinging on a deuterated polyethylene target. Furthermore, three single-channel silicon detectors were used for beam monitoring.

Listing C.1: `is_430_05/control.hh`

```
#define UNPACK_EVENT_USER_FUNCTION is430_05_user_function_multi

#define USING_MULTI_EVENTS 1
```

To make the unpacker cope with multi-event data, the name of the user-defined function to help associate the data for each physical event must be defined. This is done in the `control.hh` file above, which will be included from the generic sources as needed, according to the `-DCONTROL_INCLUDE` statement in the `makefile_additional.inc` below. Compilation and inclusion of a user source file is also requested.

Listing C.2: `is_430_05/makefile_additional.inc`

```
USE_LMD_INPUT=1

CXXFLAGS += -DCONTROL_INCLUDE

OBJS += is430_05_user.o
```

C.1 Raw data format – read-out modules

The description of the raw data format is broken down into smaller and smaller entities, until finally the data bits themselves are reached. The structures are presented here in a top-down approach. To simplify re-use, the various parts have been declared in smaller files, combined using the `C` pre-processor `#include` directive.

Listing C.3: `is_430_05/is_430_05.spec` (part I: event structure)

```
#include "spec/spec.spec"

#include "is430_05_vme.spec"

EVENT
{
  vme = IS430_05_VME(type=10,subtype=1);
}
```

Each multi-event is stored in one subevent, referenced from the `EVENT` description.

Listing C.4: `is_430_05/is_430_05_vme.spec`

```
#include "spec/land_std_vme.spec"

SUBEVENT(IS430_05_VME)
{
  header = LAND_STD_VME();

  select several
    {
      multi scaler0 = VME_CAEN_V830(geom=30);

      multi tdc[0] = VME_CAEN_V775(geom=16,crate=129);
      multi tdc[1] = VME_CAEN_V775(geom=17,crate=130);
      multi tdc[2] = VME_CAEN_V775(geom=18,crate=131);

      multi adc[0] = VME_CAEN_V785(geom=0,crate=1);
      multi adc[1] = VME_CAEN_V785(geom=1,crate=2);
      multi adc[2] = VME_CAEN_V785(geom=2,crate=3);
      multi adc[3] = VME_CAEN_V785(geom=3,crate=4);
      multi adc[4] = VME_CAEN_V785(geom=4,crate=5);
    }
}
```

The subevent begins with a header[1] shown and described below. The header is followed by an arbitrary number of data chunks from the various

[1]This is not the LMD/MBS subevent packaging header.

modules used in the experiment. As each subevent can contain data from several physical events, data for the same module may appear many times (`multi`).

Listing C.5: `spec/land_std_vme.spec`

```
LAND_STD_VME()
{
  UINT32 failure
    {
      0:  fail_general;
      1:  fail_data_corrupt;
      2:  fail_data_missing;
      3:  fail_data_too_much;
      4:  fail_event_counter_mismatch;
      5:  fail_readout_error_driver;
      6:  fail_unexpected_trigger;

      27: has_no_zero_suppression;
      28: has_multi_adctdc_counter0;
      29: has_multi_scaler_counter0;
      30: has_multi_event;
      31: has_time_stamp;
    }
  if (failure.has_time_stamp) {
    UINT32 time_stamp;
  }
  if (failure.has_multi_event) {
    UINT32 multi_events;
  }
  if (failure.has_multi_scaler_counter0) {
    UINT32 multi_scaler_counter0;
  }
  if (failure.has_multi_adctdc_counter0) {
    UINT32 multi_adctdc_counter0;
  }
}
```

The general header contains various informations regarding both the validity of the event and numbers needed for disentangling multi-event data. It begins with a 32-bit data word. Some bits mark that some failure condition was detected by the DAQ during read-out. (The DAQ would have taken corrective action, and subsequent events are not affected. The data event should be handled with proper suspicion or be discarded). Some bits indicate whether some special features were enabled during digitisation of the event, e.g. zero-suppression disabled, to allow later pedestal determination. Other bits denote that further data words will follow the header.

With the DAQ running in multi-event mode, it performs several checks on the event counter consistency between the modules in use. To allow modules with no data at all for some events, it is necessary to know the starting values of the modules' event counters for the first physical event of each multi-event. Furthermore, as the DAQ performs soft-triggering of the scaler for certain triggers, its event counter may drift relative to the others, and is recorded separately. The total number of physical events is also stored.

Listing C.6: `spec/vme_caen_v775.spec`

```
#define VME_CAEN_V792 VME_CAEN_V775 // Other modules using the same
#define VME_CAEN_V785 VME_CAEN_V775 // data format

VME_CAEN_V775(geom,crate)
{
  MEMBER(DATA12_OVERFLOW data[32] ZERO_SUPPRESS);

  UINT32 header NOENCODE
    {
      8_13:  count;
      16_23: crate = MATCH(crate);
      24_26: 0b010;
      27_31: geom = MATCH(geom);
    }

  list (0 <= index < header.count)
    {
      UINT32 ch_data NOENCODE
        {
          0_11:  value;
          12:    overflow;
          13:    underflow;
          14:    valid;

          16_20: channel;

          24_26: 0b000;
          27_31: geom = CHECK(geom);

          ENCODE(data[channel],(value=value,overflow=overflow));
        }
    }

  UINT32 eob
    {
      0_23:  event_number;
      24_26: 0b100;
      27_31: geom = CHECK(geom);
    }
}
```

The unpacker specification for data from a CAEN V775 TDC module, see Table C.1, is given in Listing C.6. The `header` data word contains two numbers, `geom` and `crate`, programmable in the module, that are used to identify data from each particular instance. They are used by the `select`

Header

31 - 27	26-24	23 - 16		13 - 8	
geom	0 1 0	crate		count	

Data word

31 - 27	26-24		20 - 16		14	13	12	11 - 0
geom	0 0 0		channel		V	U	O	value

End marker

31 - 27	26-24	23 - 0
geom	1 0 0	event_number

Table C.1: CAEN V775 data format [11].

`several` statement in the `IS430_05_VME` structure. The header also gives the number of payload data words.

Each payload data word is marked with the `channel` it corresponds to, and the actual `value`, along with `overflow`, `underflow`, and `valid` markers. The `channel` is used as an index into the zero-suppressed `data` member array, holding the unpacked data. One may note that this 32-channel TDC module has two ADC chips, converting the first and second half of the channels in parallel. The data of non-zero channels will therefore appear intermixed — a reason for the zero-suppressed data container to allow quick random insertion. As all of the information from the header and payload data is used up by the unpacking, they are marked `NOENCODE` and are not stored in the module's `unpack` level data structure.

Finally, the module may emit an end marker `eob`, containing the event number. This is in particular needed by a multi-event-unpacker to be able to assign each chunk of data to the correct physical event.

C.2 Data mappings – physical detector names

To be able to access the event-wise data by detector name, a mapping between DAM channels and physical detector names is also given in the specification file, see Listing C.7. The generated `raw` level data structure is shown in Listing C.8. Note that the zero-suppressed containers can hold structures, not only single items, thereby combining data which is related, e.g. the time and amplitude measurement of individual detector strips.

Listing C.7: `is_430_05/is_430_05.spec` (part II: data mappings)

```
// Enforce zero suppression at strip level
SIGNAL(ZERO_SUPPRESS: DSSSD2_F_32);
SIGNAL(ZERO_SUPPRESS: DSSSD2_B_32);

// ADC channels
SIGNAL(BACK1_E, vme.adc[0].data[0], (DATA12, float));
SIGNAL(BACK2_E, vme.adc[0].data[1], (DATA12, float));

SIGNAL(MONE_E, vme.adc[0].data[17],(DATA12, float));
SIGNAL(MONDE_E, vme.adc[0].data[18],(DATA12, float));
SIGNAL(MONTGT_E, vme.adc[0].data[22],(DATA12, float));

SIGNAL(DSSSD1_F_1_E, vme.adc[1].data[0],
       DSSSD1_F_32_E,vme.adc[1].data[31],(DATA12, float));
SIGNAL(DSSSD1_B_17_E,vme.adc[2].data[0],
       DSSSD1_B_32_E,vme.adc[2].data[15],(DATA12, float));
SIGNAL(DSSSD1_B_1_E, vme.adc[2].data[16],
       DSSSD1_B_16_E,vme.adc[2].data[31],(DATA12, float));
SIGNAL(DSSSD2_F_1_E, vme.adc[3].data[0],
       DSSSD2_F_32_E,vme.adc[3].data[31],(DATA12, float));
SIGNAL(DSSSD2_B_1_E, vme.adc[4].data[0],
       DSSSD2_B_32_E,vme.adc[4].data[31],(DATA12, float));

// TDC channels
SIGNAL(DSSSD1_FT, vme.tdc[0].data[0],(DATA12, float));
SIGNAL(DSSSD1_BT, vme.tdc[0].data[1],(DATA12, float));
SIGNAL(DSSSD2_FT, vme.tdc[0].data[2],(DATA12, float));
SIGNAL(DSSSD2_BT, vme.tdc[0].data[3],(DATA12, float));
SIGNAL(TMON, vme.tdc[0].data[4],(DATA12, float));
SIGNAL(TBACK, vme.tdc[0].data[5],(DATA12, float));

SIGNAL(DSSSD2_B_1_T, vme.tdc[1].data[0],
       DSSSD2_B_32_T,vme.tdc[1].data[31],(DATA12, float));
SIGNAL(DSSSD2_F_1_T, vme.tdc[2].data[0],
       DSSSD2_F_32_T,vme.tdc[2].data[31],(DATA12, float));

// Scaler channels
SIGNAL(CLOCK, vme.scaler0.data[0],DATA32);
SIGNAL(CNTPROTONS, vme.scaler0.data[1],DATA32);
SIGNAL(CNTEBIS, vme.scaler0.data[2],DATA32);

// Values that will be calculated in the user function
SIGNAL(TEBIS,                 ,(DATA32, float));
SIGNAL(TSHORT,                ,(DATA32, float));

// Value only occurs once, and valid for last multi-event
SIGNAL(LAST_EVENT: TIME, vme.header.time_stamp,uint32);
```

Listing C.8: Generated `raw` level data structure, edited for readability.

```
struct raw_event_BACK { DATA12 E; };
struct raw_event_MONE { DATA12 E; };
struct raw_event_MONDE { DATA12 E; };
struct raw_event_MONTGT { DATA12 E; };
struct raw_event_DSSSD_F
{
  DATA12 E;
  DATA12 T;
};
struct raw_event_DSSSD_B
{
  DATA12 E;
  DATA12 T;
};
struct raw_event_DSSSD
{
  raw_array_zero_suppress<raw_event_DSSSD_F,...,32> F;
  raw_array_zero_suppress<raw_event_DSSSD_B,...,32> B;
  DATA12 FT;
  DATA12 BT;
};
struct raw_event : public raw_event_base
{
  raw_event_BACK BACK[2];
  raw_event_MONE MONE;
  raw_event_MONDE MONDE;
  raw_event_MONTGT MONTGT;
  raw_event_DSSSD DSSSD[2];
  DATA12 TMON;
  DATA12 TBACK;
  DATA32 CLOCK;
  DATA32 CNTPROTONS;
  DATA32 CNTEBIS;
  DATA32 TEBIS;
  DATA32 TSHORT;
  uint32 TIME;
};
```

C.3 User functions

Many sub-structures (modules) at the `unpack` level contain data for several physical events (the `multi` qualifier used in Listing C.5). Before the event-wise data mapping of each physical event, it is necessary to determine to which physical event each chunk of data belongs. After unpacking, a user defined function, `UNPACK_EVENT_USER_FUNCTION`, is called. For each `unpack`-level data member this function shall assign each contained piece of data, e.g. using the `map_multi_events` helper function, and finally also return the number of physical events present in the current multi-event.

Listing C.9: `is_430_05/is_430_05_user.cc`

```
#include "structures.hh"
#include "user.hh"

#include "multi_chunk_fcn.hh"

int is430_05_user_function_multi(unpack_event *event)
{
  // Since the code is built to handle multi-event, the mapping of the
  // modules must be done, even if the event only has 1 physical event!

  uint32 multi_events, scaler_counter0, adctdc_counter0;

  if (event->vme.header.failure.has_multi_event)
    {
      // When running in multi-event mode, the start value for the
      // event counters for the modules must be known

      if (!event->vme.header.failure.has_multi_scaler_counter0)
        ERROR("Event counter for scaler at start unknown.");

      if (!event->vme.header.failure.has_multi_adctdc_counter0)
        ERROR("Event counter for adc/tdc at start unknown.");

      scaler_counter0 = event->vme.header.multi_scaler_counter0;
      adctdc_counter0 = event->vme.header.multi_adctdc_counter0;
      multi_events = event->vme.header.multi_events;
    }
  else // Single-event data
    {
      // Code to recover the values for scaler_counter0... omitted
      multi_events = 1;
    }

  // Loop over all the modules which are multi-event, mapping their
  // physical events using the event counters

  map_multi_events(event->vme.multi_scaler0,
                   scaler_counter0,multi_events);

  for (unsigned int i = 0; i < countof(event->vme.multi_adc); i++)
    map_multi_events(event->vme.multi_adc[i],
                     adctdc_counter0,multi_events);

  for (unsigned int i = 0; i < countof(event->vme.multi_tdc); i++)
    map_multi_events(event->vme.multi_tdc[i],
                     adctdc_counter0,multi_events);

  return multi_events;
}
```

The source also contains a user function called for each physical event after data mapping, and a function for handling unpacker-specific command-line options. These are not reproduced here.

Appendix D

Using a correlation plot to uncover event mixing

Event mixing is a technique used to get a handle on the uncorrelated background (from unrelated particles) that can be part of a spectrum when several particles are combined in the analysis of an outgoing reaction channel. In this realm it is a powerful tool, which only requires reconstructed particles from different events to be combined within the analysis when calculating e.g. invariant mass or angular correlations.

When already the DAQ mixes up data from different events into one event, it is on the other hand unwanted and very bad. It is usually caused by an unclean trigger logic in combination with a too naïve programming of the read-out — with too lax or absent continuous system integrity checks.

Letter to the editor: Debunking cargo-cult[1] DAQ rumours

After the author had received a copy of the data from a run at the IS446 experiment and started to unpack it using an `ucesb` based unpacker, it soon became clear that something in connection with the digitisation modules' event counters was strange — sometimes their increment was not synchronous between events. Normally, the event counters in the modules used[2] should count each trigger seen. The (limited) information on the trigger logics obtained also indicated that each module should see each trigger — once!

[1]Compare cargo-cult science, see "Surely you're joking, Mr. Feynman!": adventures of a curious character by Richard P. Feynman as told to Ralph Leighton; ed. by Edward Hutchings, New York, W.W. Norton, 1985.

[2]CAEN V775 TDCs and CAEN V785 ADCs.

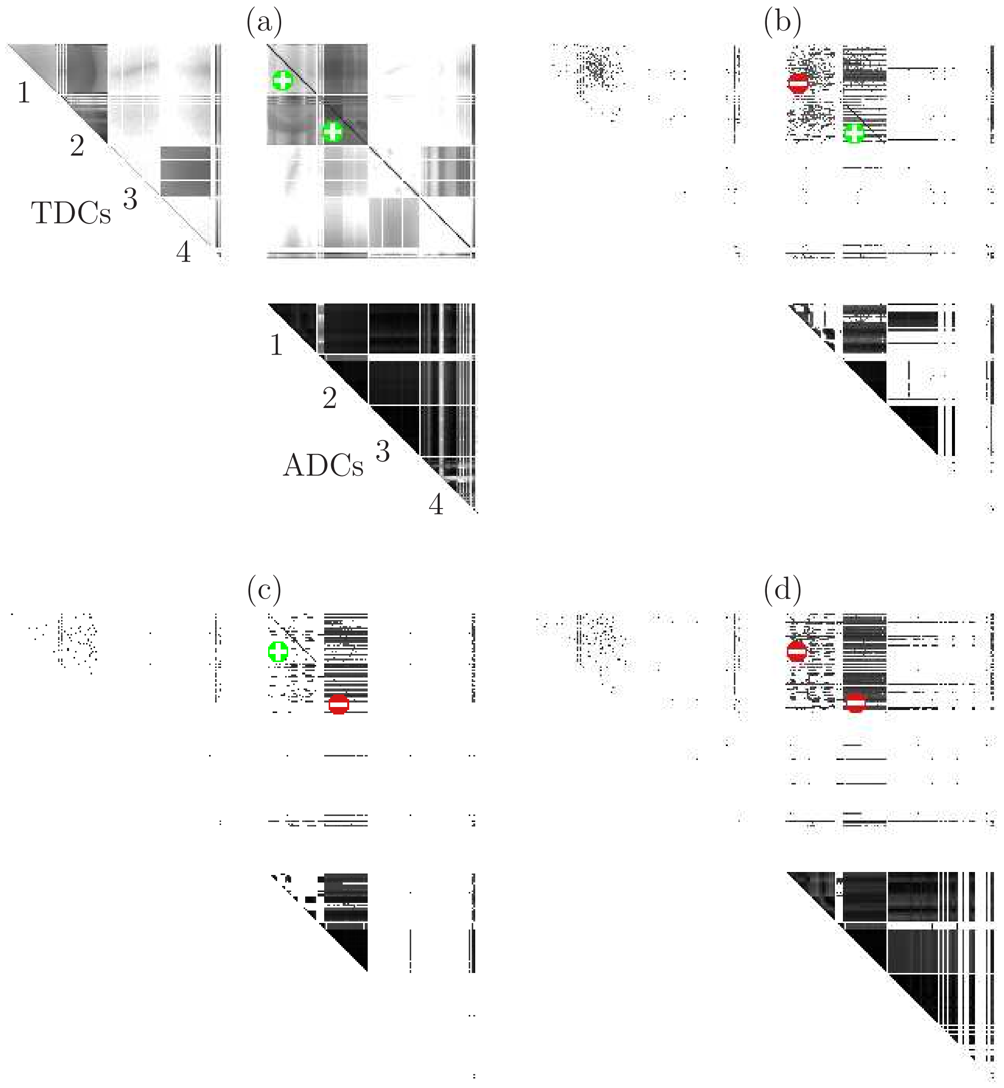


Figure D.1: Correlation plots for data from the IS446 experiment. Picture (a) shows all events. In pictures (b)-(d) (with much fewer events), the events have been chosen using the modules' events counters such that some TDC and ADC pair(s) digitising the same channels are desynchronised, marked by circles with minus signs. Mixed events are then clearly seen as the diagonal TDC-ADC correlations are missing for those events. The correlations are present for modules where the event counters did not necessarily indicate event mixing, as marked with circles with plus signs.

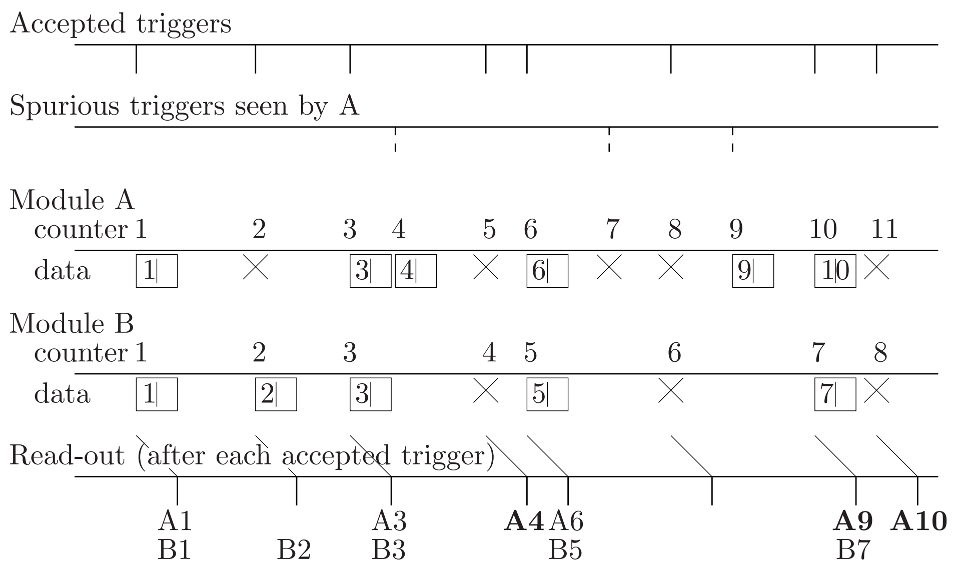


Figure D.2: Each module increases its local event counter for each seen trigger. If any channel has data above threshold (crosses mark completely zero-suppressed events), the module creates a data item for the event, which also contains the current event counter value. The data item is added to the module's multi-event output data buffer. Data from spurious triggers will be read out with the next real trigger, e.g. **A4** and **A9**. Worse, since the read-out only retrieved data for at most one event per invocation, event data could pile up in the output buffer, e.g. **A10** belongs to B7.

On further inquires, the claim from the experiment perpetrators was then that the module event counters are broken and cannot be trusted. . .

That is an interesting allegation, since the modules would then be unusable in multi-event scenarios — as in the previous Appendix! If the claim is correct, the point of this investigation would be moot, but the correlation plots of Figure D.1 tells another story:

By selecting events suspected to be desynchronised, using only the event counters, various sets of data could be obtained, each of them completely lacking the correlation between TDC and ADC channels that digitise signals for the same detector strips. The correlations are present for all other detector channels, given enough statistics.

Although the indirect cause of letting the event mixing pass unnoticed is the incomplete DAQ handling of the modules' multi-event output buffers, see

Figure D.2, it should be noted that the direct cause of the spurious triggers has not been uncovered. Unfortunately, it seems a daunting task to devise a selection technique whereby all (possibly) affected events can be discarded. The set of rather complex criteria used in these investigations so far only finds a smaller set of events that are guaranteed to be out of sync.