c4Root Documentation
v0.3.1

Calum Eoin Jones

Contents

7   Writing Tasks

8   Introduction to UCESB and SPEC

9   Troubleshooting and common issues

1 About

c4Root is a FairRoot-based software for the online and nearline analysis of HISPEC/DESPEC experiments. In principle it can be extended to the analysis of other data, but it is extremely focused on the structures and setups of our collaboration.

This software is currently developed and maintained by several Postdocs and PhD students:

- Calum Eoin Jones

- Johan Emil Linnestad Larsson

- Jeroen Peter Bormans

- Nicolas James Hubbard

- Elisa Maria Gandolfo

- Kathrin Wimmer

If you would like, or need, to contribute to this software, please contact calum.e.jones@gmail.com.

2   Installation

2.1   Downloading Software

Installation of c4Root currently requires the following softwares:

- FairRoot and FairSoft (v18+)

- cmake (v13+)

- UCESB

- c4Root (v0.9+)

FairRoot and FairSoft are easily accessible through the GSI computer network: every lxpool computer can access the latest prod releases via **/cvmfs**. Additionally these computers are equipped with an up-to-date version of cmake. UCESB and c4Root are not standard, and must be downloaded:

- UCESB can be downloaded here: https://git.chalmers.se/expsubphys/ucesb or with:

    ```
    $ git clone https://git.chalmers.se/expsubphys/ucesb.git
    ```

- c4Root can be downloaded here: https://github.com/cej25/c4Root or with:

    ```
    $ git clone https://github.com/cej25/c4Root.git
    ```

- Create a **build** directory on the same directory level as UCESB and c4Root have been downloaded to.

```
drwxr-xr-x 2 cjones ks 10 Oct  9 17:31 c4Root
drwxr-xr-x 2 cjones ks 10 Oct  9 17:31 build
drwxr-xr-x 2 cjones ks 10 Oct  9 19:28 ucesb
```

2.2   Building Software

In your **.bashrc** the following paths should be set, in order to allow c4Root to find its dependencies:

```
$ export SIMPATH=/path/to/fairsoft
# likely: /cvmfs/fairsoft.gsi.de/debian10/fairsoft/nov22p1
$ export FAIRROOTPATH=/path/to/fairroot
# likely: /cvmfs/fairsoft.gsi.de/debian10/fairroot/v18.8.0_fs_nov22p1
$ export UCESB_DIR=/path/to/ucesb
```

```
export UCESB_DIR=/u/cjones/ucesb
export SIMPATH=/cvmfs/fairsoft.gsi.de/debian10/fairsoft/nov22p1
export FAIRROOTPATH=/cvmfs/fairsoft.gsi.de/debian10/fairroot/v18.8.0_fs_nov22p1
```
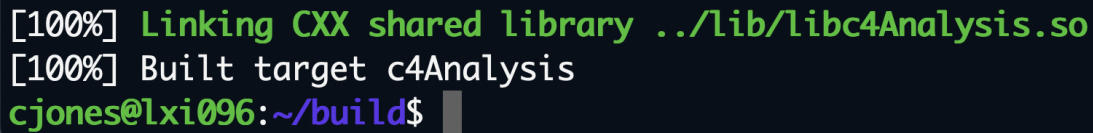
Build the empty unpacker that comes with UCESB:

```
$ cd UCESB
$ make empty −j
```

and then build the c4Root software in the build directory:

```
$ cd build
$ cmake ../c4Root
$ . ./config.sh
$ make −j
```

Assuming you reach [100%] with no red infomessages in your terminal, you have successfully built the software and it can now be used!

```
[100%] Linking CXX shared library ../lib/libc4Analysis.so
[100%] Built target c4Analysis
cjones@lxi096:~/build$
```

The final thing you should do, for your own comfort of usage, is to add the following commands to your **.bashrc**:

```
cd $SIMPATH/bin;. ./thisroot.sh;cd
cd $FAIRROOTPATH/bin;. ./FairRootConfig.sh;cd
export FAIRROOTPATH=/cvmfs/fairsoft.gsi.de/debian10/fairroot/v18.8.0_fs_nov22p1
```

As of v18 of FairRoot, there is a bug when configuring the software that unsets the FAIRROOTPATH variable, which is why it must be reset.

3 Navigating the Code Structure

The c4Root software is organised into a series of directory structures, each responsible for a different function or stage of the analysis.

3.1 Analysis

In **/analysis** tasks for plotting spectra and data analysis are contained. This directory is further subdivided into **/online**, **/offline** and **/correlations**:

- Online Tasks are used to plot spectra for live monitoring during an experiment.

- Offline Tasks are used for deeper analysis after an experiment has run its course.

- Correlation Tasks are used when analysis requires two or more subsystems. There are some online correlation tasks for when your supervisor asks, but generally these are quite intensive and can affect the efficiency of the rest of your monitoring tasks.

Analysis Tasks *generally* fill histograms with some appropriate level of data. But they may also implement some extra analysis step to the data, such as gating conditions or intra-subsystem coincidences.

3.2 Base

In **/base** classes used across multiple areas of the software are contained. These are mostly helper classes that perform a routine necessary in some stage of the analysis, but is not part of the workflow itself.

3.3 Data

The **/data** directory contains data structure classes. This directory is further subdivided, by subsystem (e.g. **/aida, /degas, /fatima**). Each subsystem may have the following data levels*:

- RawData

- CalData

- HitData

- AnlData

  **these names may vary*

Raw data is unpacked from an .LMD file using UCESB, and placed in the first data level structure (Data/UnpackData/RawData). This raw data is then calibrated and mapped if appropriate, and placed in the second data level structure (CalData). After this, further processing may be required and data may be stored in higher levels such as HitData or AnlData, depending on the subsystem. For example, in the FRS code: Raw V7X5 signals from the so-called *User* crate are placed in a **FrsMusicItem** structure. They are then converted into Calibrated MUSIC energy values and stored in a **FrsCalMusicItem** structure, and finally a **Z** value is calculated and placed in a **FrsHitItem** structure.

3.4 Source

The Tasks contained in **/source** are those that process the data from Raw to Analysed, in multiple steps. This directory is further subdivided by subsystem (e.g. **/aida, /degas, /fatima**). The data processing flow is shown in the figure below - generally:

- {Subsystem}**Reader** takes unpacked data from UCESB and places it in the first level data structure in c4Root

- {Subsystem}**Raw2Cal** takes raw data from the first level data structure and performs calibrations and mappings.

- {Subsystem}**Cal2Hit** takes calibrated data and further processes into analysed "hits" if required.
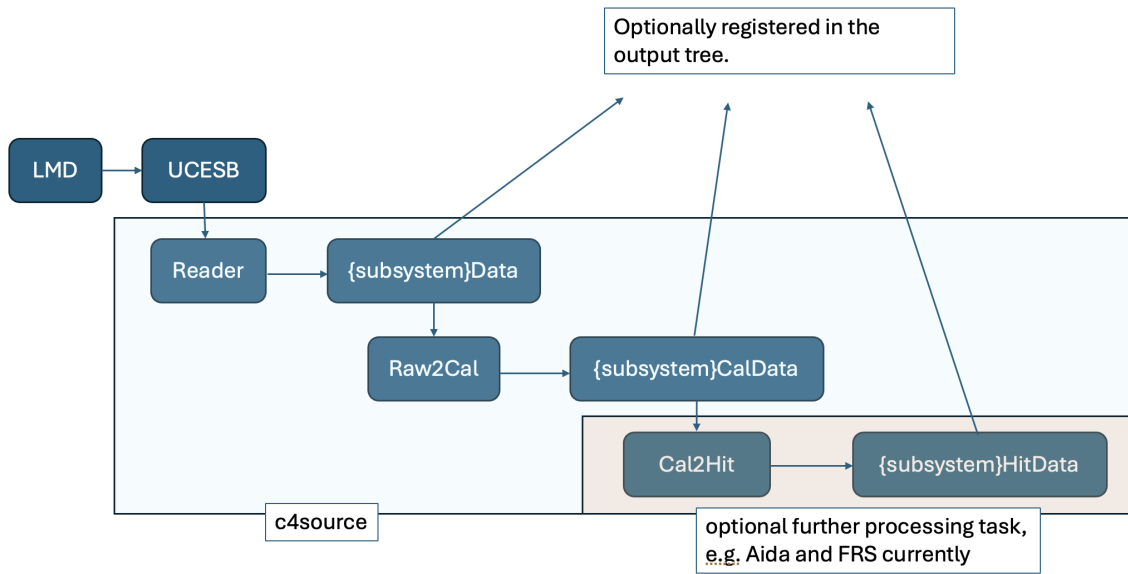
3.5 Unpack

The **/unpack** directory contains the so-called "unpackers", written in the Spec language of UCESB. A guide to 'Spec' is given further in this document.

3.5.1 Exps

In **/exps** are unpacking codes specific to a particular experiment. These generally call upon some combination of module unpackers contained in **/common**.

3.5.2 Common

The **/common** directory contains unpacking code for individual electronics modules, to be used for many subsystems across different experiments.

3.5.3 Scripts

The **/scripts** directory contains useful scripts pertaining to UCESB specifically. Currently the following exist:

- **gen_struct_header.py** - this script generates the header structure file that c4Root and UCESB require to communicate, for a given subsystem.

3.6 Macros

In **/macros** are a series of ROOT macros that use c4Root to perform some analysis.

3.6.1 Despec

In the DESPEC section are the online macros for a particular experiment. Once the experiment is concluded they are moved to **/old** for reference.

3.6.2 Test

This directory is likely a mess, with macros used for code-testing and debugging purposes.

3.6.3 Util

There are a selection of so-called "Utility" macros available, pre-written and designed for a specific single purpose.

4   Using c4Root

Using the software comes with a bit of a learning curve, but becomes simple with a little experience.

In general a ROOT macro, hereby referred to as a "Steering" Macro is used to invoke many Tasks that are part of the c4Root software. Steering Macros are smaller pieces of code dedicated to a particular experiment or smaller analyses. Each will follow the same basic template, however there may be differences depending on whether the analysis is online or offline.

4.1   Steering Macro Template

For an experiment with many subsystems, the macro begins with a series of definitions that allow particular systems to be switched on and off. If used, these affect how different tasks should be added to the macro, so take care.

```
#define FATIMA_ON 1
#define FATIMA_VME_ON 1
#define AIDA_ON 1
#define BPLAST_ON 1
#define GERMANIUM_ON 1
#define BGO_ON 1
#define FRS_ON 1
#define TIME_MACHINE_ON 1
#define BEAMMONITOR_ON 0
#define WHITE_RABBIT_CORS 1
#define BB7_ON 1
```

When the FRS is used, a setup file is included to access parameters used in the analysis. This is included using the relative filepath, pointing to the **/config/{experiment_name}** directory.

```
extern "C"
{
    #include "../../config/s100/frs/setup_s100_010_2024_conv.C"
}
```

The unpack structures, generated by UCESB, for each subsystem used in the analysis are then set in a structure as below:

```
typedef struct EXT_STR_h101_t
{
    EXT_STR_h101_unpack_t eventheaders;
    EXT_STR_h101_fatima_onion_t fatima;
    EXT_STR_h101_fatimavme_onion_t fatimavme;
    EXT_STR_h101_aida_onion_t aida;
    EXT_STR_h101_bplast_onion_t bplast;
    EXT_STR_h101_germanium_onion_t germanium;
    EXT_STR_h101_frsmain_onion_t frsmain;
    EXT_STR_h101_frstpc_onion_t frstpc;
    EXT_STR_h101_frsuser_onion_t frsuser;
    EXT_STR_h101_frstpat_onion_t frstpat;
    EXT_STR_h101_beammonitor_onion_t beammonitor;
    EXT_STR_h101_bgo_onion_t bgo;
    EXT_STR_h101_bb7vme_onion_t bb7vme;
} EXT_STR_h101;
```

After this preamble, the main macro function begins. Everything associated with a particular experiment will be contained in directories named after the experiment code, e.g. "s100" (this is not a given, but rather the convention chosen for this software and its usage). For this reason, the first thing we do in the macro is define a string with the name of the experiment:

```
TString fExpName = "s100";
```

After which a series of useful paths are defined:

```
// Define important paths.
TString c4Root_path = "/u/cjones/c4Root";
TString ucesb_path = c4Root_path + "/unpack/exps/" + fExpName
                     + "/" + fExpName + " --debug --input-buffer=200Mi --event-sizes";
ucesb_path.ReplaceAll("//","/");

std::string config_path = std::string(c4Root_path.Data()) + "/config/"
                          + std::string(fExpName.Data());
```

At this point, the input "file" should be declared. An output file is also declared for nearline analysis, or if the histograms produced during online monitoring wish to be saved to a file.

```
TString filename = "trans://lxg1257"; // transport server on lxg127
TString outputpath = "online_output";
TString outputFileName = outputpath + ".root";
```

The FairRoot Online run is then created, and a **THttpServer** is activated for viewing histograms that can be filled with live data.

```
Int_t port = 5000; // firewall exception on port 5000
Int_t refresh = 10; // 10s refresh rate for online server

FairRunOnline* run = new FairRunOnline();
EventHeader* EvtHead = new EventHeader();
run->SetEventHeader(EvtHead);
run->SetRunId(1);
//run->SetSink(new FairRootFileSink(outputFileName));
run->ActivateHttpServer(refresh, port);
TFolder* histograms = new TFolder("Histograms", "Histograms");
FairRootManager::Instance()->Register("Histograms", "Histogram Folder", histograms, false);
run->AddObject(histograms);
```

With the FairRun created, a **UcesbSource** is defined and added to the run. More details about the UcesbSource class are given in another section.

```
EXT_STR_h101 ucesb_struct;
TString ntuple_options = "UNPACK, RAW"; // Level of data to unpack
UcesbSource* source = new UcesbSource(filename, ntuple_options,
                      ucesb_path, &ucesb_struct, sizeof(ucesb_struct));
source->SetMaxEvents(nev);
run->SetSource(source);
```

The UcesbSource class connects UCESB with c4Root, unpacking raw data for the software to process. Data is processed in a series of **FairTasks**, allowing for versatile control over which level of data is analysed and displayed.

```cpp
// EventHeader - should always be done
UnpackReader* unpackheader =
    new UnpackReader((EXT_STR_h101_unpack*)&ucesb_struct.eventheaders,
    offsetof(EXT_STR_h101, eventheaders));

source->AddReader(unpackheader);

if (FATIMA_ON)
{
    FatimaReader* unpackfatima =
        new FatimaReader((EXT_STR_h101_fatima_onion*)&ucesb_struct.fatima,
        offsetof(EXT_STR_h101, fatima));
    //unpackfatima->DoFineTimeCalOnline(config_path +
    //      "/fatima/fine_time_s181_8June.root", 1 000 000);
    unpackfatima->SetInputFileFineTimeHistos(config_path + "/fatima/fine_time_s181_8June.root");

    unpackfatima->SetOnline(true);
    source->AddReader(unpackfatima);
}

if (AIDA_ON)
{
    AidaReader* unpackaida =
        new AidaReader((EXT_STR_h101_aida_onion*)&ucesb_struct.aida,
        offsetof(EXT_STR_h101, aida));

    unpackaida->SetOnline(true);
    source->AddReader(unpackaida);
}
```

4.2 Online Analysis

Online macros are generally designed to read a live data-stream from a transport server while an experiment is ongoing. In principle however, it is possible to read an .LMD file and watch spectra fill as if it were "online". The 'filename' variable should reflect your choice of data source:

```cpp
TString filename = "trans://lxg1257"; // timesorter.
```

For HISPEC/DESPEC experiments, all used subsystems are first connected in a "timesorter" computer. Events are ordered by absolute time (White Rabbit), and then time-stitched into *Events*. If more than one subsystem is being analysed, this time-stitching is critical, so ensure that your data source is the computer server time-stitched events (**lxg1257** currently) or a stitched .LMD file.

Running the macro is simple - assuming c4Root, FairRoot and FairSoft have been configured and an experimental Unpacker (ref) has been built, the following command can be run:

$ root −b {macro_name}.C

A series of [INFO] statements will be printed, and then data processing will begin.

```
Processing s100_tests.C...
[INFO] FairRunOnline constructed at 0x561a1dffba90
Info in <THttpEngine::Create>: Starting HTTP server on port 7070
Focus distance S4: 3700
Setup done
[INFO] FatimaTwinpeaks Allocation coefficients File: /u/cjones/c4Root/config/s100/fatima/fatima_alloc_new.txt
[INFO] Germanium Configuration File: /u/cjones/c4Root/config/s100/germanium/ge_alloc_apr15.txt
[INFO] Germanium Calibration File: /u/cjones/c4Root/config/s100/germanium/ge_uncal_apr15.txt
[INFO] FrsGate:FrsGate: PID gate Gd164 loaded with cuts on Z/AoQ = 1, Z/Z2 = 0, x2/AoQ = 0, x4/AoQ = 0, dEdeg/
Z = 0
[INFO] FairRunOnline::Init
[INFO] UcesbSource:Init: EventHeader. was defined properly - success!
```

If the data source is found, you should see a "Processed: N {events} " line in the terminal while the
program runs:

```
Processed: 354.6k (38.1k/s)   354.6k (38.1k/s) (0 errors) /
```

4.3   Nearline / Offline Analysis

Nearline and Offline analysis is performed after a run or a full experiment has concluded; hence data
must be read from a *time-stitched* .LMD file. Nearline/Offline analysis is generally broken into two
parts, with a macros responsible for each half:

- **Tree-maker** - this macro reads a data file and creates ROOT trees.

- **Histogram-writer** - this macro reads a ROOT tree and creates a ROOT file containing histograms.

The tree-making macro is almost identical to a large portion of the online macro. The real difference
comes with the histogram-writer - reading ROOT trees requires a different FairRoot process, and
in order to structure the output file correctly (such that many can be combined), slightly altered
/analysis Tasks are required.

```cpp
if (FRS_ON)
{
    FrsOnlineSpectra* onlinefrs = new FrsOnlineSpectra(frsgates);
    run->AddTask(onlinefrs);
}
```

```cpp
if (FRS_ON)
{
    FrsNearlineSpectra* nearlinefrs = new FrsNearlineSpectra();
    run->AddTask(nearlinefrs);
}
```

Each macro can be run from the terminal in the same way as the online macro:

```
$ root -b {tree_maker}.C
$ root -b {histogram_writer}.C
```

While the tree-maker will provide a progress bar as the online macro does, the histogram-writer will not do this, and simply quit ROOT when complete. This is because the progress bar comes from the UCESB unpacking procedure.

4.3.1   Cluster Submission

Analysing large data-sets near/offline would be a slow process file-by-file in the terminal. Instead, for many files, you should submit an analysis job to the GSI/FAIR supercomputer cluster. Cluster submission will require a slightly modified version of the nearline/offline macros, as well as bash scripts to perform the job submission. It also requires that you have access to virgo, so check that first!

First, when in the virgo computers, you must build an installation of c4Root as described in Section 2. You will also have to change the paths pointing to UCESB_DIR, FAIRROOTPATH and SIMPATH, so it is recommended to set up a **.bashrc** script that exclusively runs when logging on to the virgo computers, as to not interfere with working on the lx-pool computers. Once c4Root is downloaded and built, make sure that an experimental unpacker is also built.

Next, set up your workspace. You should already have **c4Root**, **build** and **ucesb** directories. It is recommended to create the following directories:

- **/cluster** to contain the cluster submission scripts. Inside **/cluster** make a **/logs** directory to store the SLURM output log files.

- **/trees** to contain generated ROOT trees.

- **/histograms** to contain generated ROOT histograms.

In **/cluster**, you will then need the following*:

- **tree_maker.C** - this is a version of the nearline tree-making macro, slightly altered for cluster submission.

- **submit_trees_to_cluster.sh** - this is a bash script that sets up the SLURM job, and executes the launcher script.

- **tree_launcher.sh** - this is a bash script that defines the software and library paths, reads the file list and executes the tree-maker.

- **file_list_for_trees.txt** - this is a list of files from which trees should be made.

- **histogram_writer.C** - this is a version of the nearline histogram-writing macro, slightly altered for cluster submission.

- **submit_histograms_to_cluster.sh** - this is a bash script that sets up the SLURM job, and executes the launcher script.

- **histogram_launcher.sh** - this is a bash script that defines the software and library paths, reads the file list and executes the histogram-writer.

- **file_list_for_histograms.txt** - this is a list of tree files from which histograms should be made.

  name the scripts however you like, of course

The tree-maker and histogram-writer should be altered at the very least such that it can take in a filepath as an argument. Relative filepaths for the output file, calibration files, etc... should all be carefully edited. An example of this submission kit, with comments, is available in the **/cluster** directory of c4Root. These scripts are prepared to work with the build contained in **/c4Root_cluster_demo** in **/lustre/gamma**. You should make your own workspace and change the filepaths if you wish to test yourself, though.

To submit a job, simply run the command:

```
$ ./submit_trees_to_cluster.sh
```

You can watch the progress of the job with:

```
$ watch squeue -u {username}

e.g.
$ watch squeue -u cjones
```

Take care when attempting to make trees. For experiments using subsystems such as FATIMA, fine-time calibrations are required. You can make these calibrations yourself (see Section **??**), or you can use previously made calibrations. However, these calibrations are stored in ROOT files, and ROOT files are *not* uploaded to **git**, meaning when you clone the code from github, you will not have these particular calibration files. The same is true for FRS gates (see Section **??**). If these files exist, copy them into your workspace version of c4Root in the correct path, otherwise the steering macro will not be able to find the file, and the cluster job will not complete.

5   Helper Scripts and Macros

There are a host of scripts available to automate parts of writing and using the software.

- convert_setup.py is a script that converts the 'setup' configuration files from the FRS Go4 into a format that can be used by this software.
  Location: /macros/util/convert_setup.py
  Usage: >python convert_setup.py {name_of_setup_file}.C

  Several old parameters will be renamed by this script, which is essential if analysing experiment s181 or older!

6 Implemented Subsystems

This section will describe the Tasks and Classes of each of the currently implemented DESPEC subsystems. They generally have a similar format with a similar method structure, the descriptions can be expected to be extremely similar, especially for subsystems using the same electronics. If you're interested in developing another subsystem, see instead Section 7.

6.1 EventHeader

The EventHeader data class contains a series of metadata items about each event, useful for events that contain numerous subsystems and subevents with multiple hits.

6.2 FATIMA

FATIMA is unique in that it has run for many experiments with two separate DAQs, at GSI, while TAMEX was being developed and tested. Therefore, there are two sets of event processing tasks. Those named with simply the *Fatima* prefix refer to the TAMEX DAQ, while those with the *FatimaVme* prefix refer to the FATIMA DAQ using VME CAEN modules. FATIMA with VME may be obsolete after experiment **s181** however these classes are still needed in order to retroactively analyse this data.

FATIMA has two primary event processing FairTasks:

- **FatimaReader** - This takes unpacked data from UCESB and stores it in the **FatimaTwinpeaksData** structure. The Fine Time calibration is applied in this task.

- **FatimaRaw2Cal** - This takes raw data from **FatimaTwinpeaksData** and performs a mapping, calibrations and stores the data in the **FatimaTwinpeaksCalData** structure.

There are also a number of Tasks associated with analysis and plotting histograms:

- **FatimaOnlineSpectra**
- **FatimaNearlineSpectra**
- **FatimaSinglesAnalysis**
- **FatimaTimingAnalysis**
- Correlations...

6.2.1 FatimaReader

The **FatimaReader** Task has the following methods:

- **Init** - This method is called once at the beginning of processing. The raw data branch is registered into the tree here, and the fine time coefficients are initialised if applicable.

- **Read** - This method is called once per event. Unpacked data is processed; relevant data is extracted and stored in FatimaTwinpeaksData structure.

- **Reset** - This method is called once per event, after **Read()**. Arrays used in the processing of data are cleared here to make way for the next event.

- **PrintStatistics** - This method prints statistics pertaining to the TAMEX boards read, typically used once a file has been processed.

- **SetOnline** - This method allocates whether data is written to the tree or not. **True** means *Online* mode, and the data is not written to the tree. **False** means the data is written to the tree. This is used for online vs. nearline analysis.

- **DoFineTimeCalOnline** - This method allows a user to do a fine time calibration while processing live data.

- **DoFineTimeCalibration** - This method performs a fine time calibration using an allocated number of hits. Typically 100,000+ events should be used for a good calibration.

- **WriteFineTimeHistosToFile** - Once a fine time calibration is performed, this method stores the calibration in an allocated filepath, in a ROOT histogram.

- **ReadTimeFineHistosFromFile** - This method reads in a fine time calibration histogram from an allocated filepath.

- **GetFineTime** - This method retrieves the fine time from the histogram.

- **SetInputFileFineTimeHistos** - This method allocates the filepath to read in an already existing fine time calibration.

Some are needed in the Task itself, other methods may be useful to access from the steering macro, for example to perform a fine time calibration:

```
FatimaReader* unpackfatima = new FatimaReader((EXT_STR_h101_fatima_onion*)&ucesb_struct.fatima, offsetof(EXT_STR_h101, fatima));
unpackfatima->DoFineTimeCalOnline(config_path + "/fatima/fine_time_s181_8June.root", 1 000 000);
//unpackfatima->SetInputFileFineTimeHistos(config_path + "/fatima/fine_time_s181_8June.root");
```

Note: if the **DoFineTimeCalOnline** method is being used no histograms will be filled! This is true for FATIMA and all other subsystems that use the similar fine time calibration methods. You should perform the fine time calibration, if needed, and then restart the macro after commenting/removing this line!

6.2.2 FatimaRaw2Cal

The **FatimaRaw2Cal** Task has the following methods:

I think we can be a bit more specific about what some tasks do, especially Exec

- **Init** - This method is called once when the Task is initialised. The **FatimaTwinpeakData** branch is read in, and the **FatimaTwinpeaksCalData** branch is registered to the tree here. The **TimeMachinedData** branch is also registered at this point, as TimeMachine hits are processed at the Raw2Cal stage.

- **Exec** - This method is called for every event. FATIMA hits are mapped to a detector, calibrations are applied and data is stored in the **FatimaTwinpeaksCalData** structure here. TimeMachine hits are extracted and placed in the **TimeMachineData** class.

- **FinishEvent** - This method is called once per event, after **Exec()**. Here arrays used in the event processing are cleared.

- **FinishTask** - This method is called after a Task has finished executing completely, i.e. when there are no more events to read.

- **SetOnline** - This method allocates whether or not the **FatimaTwinpeaksCalData** is branch is written to the tree. **True** means *Online* mode, and the branch is not written. **False** means the branch is written to the tree.

- **PrintDetectorMap** - This method prints the detector mapping to terminal, before event processing begins.

- **PrintDetectorCal** - This method prints the detector calibration coefficients to terminal, before event processing begins.

6.3 DEGAS

DEGAS classes are in fact labelled with the *Germanium* prefix. DEGAS data is processed in several steps:

- **GermaniumReader**

- **GermaniumRaw2Cal**

- **GermaniumCal2Anl**

There are also a number of analysis and plotting Tasks:

- **GermaniumOnlineSpectra**

- **GermaniumNearlineSpectra**

- **GermaniumResolution**

6.4   bPlast

bPlast classes have the **bPlast** prefix. bPlast uses the TAMEX4 DAQ with TwinPeaks, and so the FairTasks processing event data are remarkably similar to those of FATIMA in many ways. That being said, one looks for different analyses in these subsystems, so there are differences.
bPlast has two primary event processing FairTasks:

- **bPlastReader** - This takes unpacked data from UCESB and stores it in the **bPlastTwinpeaksData** structure. The Fine Time calibration is applied in this task.

- **bPlastRaw2Cal** - This takes raw data from **bPlastTwinpeaksData** and performs a mapping, calibrations and stores the data in the **bPlastTwinpeaksCalData** structure.

There are also a number of Tasks associated with analysis and plotting histograms:

- **bPlastOnlineSpectra**

- **bPlastNearlineSpectra**

- Correlations...

6.5 AIDA

AIDA classes have the *Aida* prefix.

Aida has three main processing FairTasks:

- **AidaReader** - this.
- **AidaUnpack2Cal** - this.
- **AidaCal2Hit** - this.

6.6 FIMP

FIMP classes have the *Fimp* prefix.

FIMP has two primary event processing FairTasks:

- **FimpReader** - This takes unpacked data from UCESB and stores it in the **FimpItem** structure. The Fine Time calibration is applied in this task.

- **FimpRaw2Cal** - This takes raw data from **FimpItem** and performs a mapping, calibrations and stores the data in the **FimpCalItem** structure.

There are also a number of Tasks associated with analysis and plotting histograms:

- **FimpOnlineSpectra**

- **FimpNearlineSpectra**

6.7 BGO

BGO classes have the *Bgo* prefix. BGO also uses the TAMEX4 with Twinpeaks DAQ, and thus like bPlast has similar event processing Tasks. The analysis Tasks however are dependent on DEGAS, since the BGOs act as a veto on the Germanium detectors.

bPlast has two primary event processing FairTasks:

- **BGOReader** - This takes unpacked data from UCESB and stores it in the **BGOTwinpeaksData** structure. The Fine Time calibration is applied in this task.

- **BGORaw2Cal** - This takes raw data from **BGOTwinpeaksData** and performs a mapping, calibrations and stores the data in the **BGOTwinpeaksCalData** structure.

There are also a number of Tasks associated with analysis and plotting histograms:

- **Online/Correlation with Ge**

6.8   LISA

LISA classes have the *Lisa* prefix.

LISA has Reader, Raw2Ana, Ana2Cal

I'll let Elisa explain this bit... ;-)

6.9 FRS

The FRS is a complex "subsystem" to get to grips with, as there are many moving parts to the hardware that change from experiment to experiment (unlike the DESPEC systems generally, where the electronics often remain the same). This software is designed to be general enough to analyse *any* experiment, so writing something that can handle these electronics crate + module changes, without changes to the source code (on an experiment-by-experiment basis) is ~~borderline impossible~~ a big challenge. Nevertheless, we persevere.

The FRS detectors are typically connected to digitizer modules that are connected to a series of crates, labelled: **Main**, **User** (also **FRS** but "FRS FRS Crate" felt silly to write), **TPC** and **TPAT**. A "travelling MUSIC" detector has it's own separate crate, as does the **VFTX** system, which is currently non-operational. As mentioned however, care should be taken since modules can and will be moved around from year to year. The organisation of modules into crates is mapped in UCESB itself, using the **mapping.hh** file. A discussion of how UCESB mapping works is given in Section 8.2. This takes FRS data from the 'UNPACK' level to the 'RAW' level, so the ntuple_options in the Steering Macro string should be set to account for this when the FRS is used:

```
TString ntuple_options = "UNPACK,RAW";
```

There are a number of Tasks written for the processing of FRS data:

- **FrsReader** - The Reader takes 'RAW' level data from UCESB and places it into several data structure classes corresponding to its detector system. It is broken into several methods for this purpose:

  - The White Rabbit timestamp and TPAT data are placed into an **FrsTpatItem** structure.
  - **ReadScalers**: Scaler data is placed into an **FrsScalerItem** structure.
  - **ReadScintillators** Scintillator data is placed into an **FrsSciItem** structure.
  - **ReadMusic**: MUSIC data is placed in an **FrsMusicItem** structure.
  - **ReadTpcs**: TPC data is placed in an **FrsTpcItem** structure.

- **FrsRaw2Cal** - This task maps and calibrates data from the Scintillators and TPCs using two methods:

  - **ProcessScintillators**: Data (dE, dT and T) from the **FrsSciItem** is mapped to specific scintillators (e.g. SciData[channel0] → sci_de_41L). These mapped data are placed in an **FrsCalSciItem** structure.
  - **ProcessTpcs**: TPC data is mapped and checked against checksums and window conditions to determine positions and angles along the FRS. Data such as tpc_x_s4 is placed in an **FrsCalTpcItem** structure.
  - Other data does not require an inbetween step and is thus not processed here. It is possible some rearrangement of the processing may occur in the future - e.g. dE from MUSIC could be calculated in this Task, but for now this is our setup.

- **FrsCal2Hit** - This Task takes calculates what is referred to as the 'ID' or 'Hit' Data (e.g. Z, A/Q). Data is processed in a series of steps, broken down into several methods:

  - **ProcessScalers:** This Task makes some calculations with the FRS scalers in order to plot them later. This analysis may well be moved to a plotting **/analysis** Task in the near future.

- **ProcessScintillators:** This Task calculates position and time-of-flights from the TAC Scintillator data.
- **ProcessScintillator_MHTDC:** This Task does the same as above, but for the MHTDC channels.
- **ProcessMUSIC:** This Task calculates energy loss in the MUSIC detectors.
- **ProcessIDs:** This Task calculates values such as **beta**, **Z** and **AoQ** using MUSIC, Scintillator and TPC data.
- **ProcessIDs_MHTDC:** This Task does the same as above, for the MHTDC channels.
- **ProcessDriftCorrections:** This Task corrects ID values Z and AoQ to account for the drift over the course of the experiment.

Calculated values are placed in an **FrsHitItem** structure.

Data from multiple systems is required for 'Hit' analysis (e.g. Z requires the Scintillators for Time-of-Flight and MUSIC for energy loss), however in principle the code could be set up in a way to allow one or more of these methods to be switched on or off. As of c4v1.0, this has not been implemented for simplicity.

6.9.1 Usage

There are several essential components required for using FRS Tasks. Without them, including them in a Macro will cause the program to quit early.

Crate Map
Analysis of the FRS requires a "crate_map.txt" file, which outlines which channel number Scintillators are plugged into (both MHTDC and TAC channels). In the near future this will include TPC mappings. The crate map file should be set in the Steering Macro:

```
TFrsConfiguration::SetCrateMapFile(config_path + "/frs/crate_map.txt");
```

Setup File
Also required is the **setup.C** file. These should be taken from the FRS team and converted to a format readable by c4. This can only be done *easily* with the script **convert_setup.py** which can be found in **/macros/util** - several references to Go4 must be removed, and several parameters must be renamed if you are analysing an older experiment (s181 or earlier). To use this script see scripts in section 3.6.3. The path to this setup file should be included at the top of the macro, as shown in Section 4.1. Your steering macro should then contain the following (before any FRS Tasks are declared):

```
TFRSParameter* frs = new TFRSParameter();
TMWParameter* mw = new TMWParameter();
TTPCParameter* tpc = new TTPCParameter();
TMUSICParameter* music = new TMUSICParameter();
TLABRParameter* labr = new TLABRParameter();
TSCIParameter* sci = new TSCIParameter();
TIDParameter* id = new TIDParameter();
TSIParameter* si = new TSIParameter();
TMRTOFMSParameter* mrtof = new TMRTOFMSParameter();
TRangeParameter* range = new TRangeParameter();
setup(frs,mw,tpc,music,labr,sci,id,si,mrtof,range); // Function defined in frs setup.C macro
TFrsConfiguration::SetParameters(frs,mw,tpc,music,labr,sci,id,si,mrtof,range);
```

6.10 Beammonitor

The Beam Monitor subsystem essentially analyses the structure and timing of ions in each spill, and determines the 'Quality' of the beam. It is used for diagnostic purposes on the DESPEC side to alert shifters if there is a dramatic decrease in the quality of the beam.

6.11 BB7

While the BB7 layer of AIDA is still in the testing/commissioning phase(s), 'BB7' is considered a separate subsystem.

7   Writing Tasks

Writing a FairTask is fairly simple; it can be broken down into several steps:

1. Write the Task. There are only a few main methods needed to ensure proper event processing. Additionally, with the Task generator script (reference), this is made even easier.

- **Init()**
- **Exec()**
- **FinishEvent()**
- **FinishTask()**

2. Add the Task to the relevant LinkDef and CMake files. Most sections of the software (**/source**, **/data**, **/analysis**) each have {Name}LinkDef.h header and CMakeList.txt files.

– image –

3. Invoke the Task in a steering macro and add it to the **FairRun**.

– image –

8 Introduction to UCESB and SPEC

UCESB is written in a so-called 'specification' language, inherently linked to C-structures. It is an extremely efficient 'unpacking' software, used to convert raw binary data (in .LMD files) into accessible C objects. To analyse any experiment, you will need a UCESB unpacker. Many have already been written that can be copied and slightly altered, and many DAQ module unpackers have been written in **/common**. There may be cause to write an unpacker for a new module however, and this section will attempt to detail how to do so.
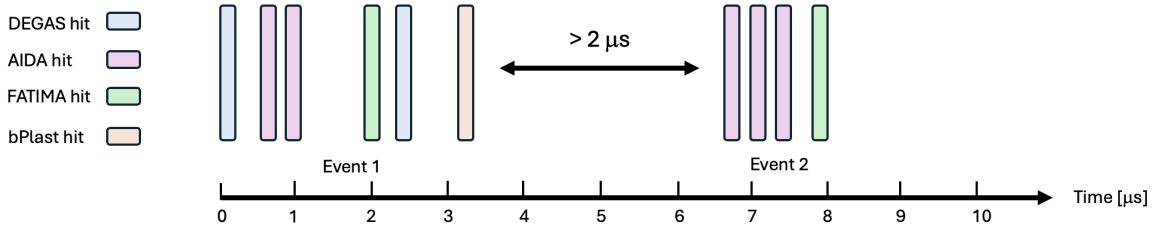
Time-sorted data is time-stitched into *Events* using the White Rabbit timestamp, in a process that searches for gaps of two microseconds. Once it finds a gap of this size or more, the previous event is closed and a new event window is opened. Any subevent found within 2 microseconds is considered to be part of the the previous event. There is nothing stopping you, in your analysis, from looking at subevents across multiple events - but subevents within the same stitched event from this process are the ones considered to be **time-correlated** and therefore used for coincidence analysis.

The main block of an unpacker is the spec file named **{exp}.spec**. In this file you will define all the details of your experiment configuration (which detectors, which electronics they use, etc). This file has two main sections: the **EVENT** block and the **SUBEVENT** block:

- The **EVENT** block is used to tell UCESB which subsystems it should look out for in the datastream and how to find them. Here the ProcID values (and other discriminators, such as Type, Crate ID, Control) are provided to make a subevent selection.

- The **SUBEVENT** block defines how a particular subevent should be unpacked. Each subsystem has its own **SUBEVENT** block with a set of precise rules for unpacking data bit-by-bit.

29

In this experiment example, we are using AIDA, DEGAS (Germaniums), FATIMA (with both DAQs), bPlast, FRS and the Beam Monitor. We can select each subevent using the ProcID; some other discriminators have been added for demonstration. Each subevent has its own **SUBEVENT** block, an example of which is given below. In each event, UCESB will attempt to match only one subevent to a given structure, unless the keyword **revisit** is given (as with AIDA). We choose to use **ignore_unknown_subevent** because there is BGO data in our stream, but we don't want UCESB to complain and throw errors.

Taking FATIMA (with TAMEX) as an example, the above shows how a **SUBEVENT** block is defined. The name given to the subevent structure is in the brackets, and is the name that should be called in the **EVENT** block. Inside this block, the precise data structure is detailed. Each subevent data block begins with the five **White Rabbit** words. This structure is defined in **/common/whiterabbit.spec**; the only thing to take care of is that **whiterabbit.spec** is included (much like a header file in C) at the top of your **{exp}.spec** file. TAMEX is also a common module (**gsi_tamex4.spec**), and thus by including this functions associated with the TAMEX data structure can be used.

8.1 Setting up an Experiment

There are a few steps to setting up a new experiment, but in many cases most of the work should simply be copying bits and pieces from other experiment unpackers, as the systems will often have been used before. The first step is to create the experimental unpacker directory. In the following example, a hypothetical experiment 's999' will be created. In **/unpack/exps**:

```
$ mkdir s999
```

In **/exps** there should be a list of directories for each experiment, now including yours, as well as a file called 'Makefile'. At the top of this file, you should see a variable EXPS being assigned a list of experiment codes - s999 should now be added to this list.

Next, you will need to set up your experiment directory. The easiest and recommended way to do this is to *start* by copying a previous example. For example, from **/exps**:

```
$ cd s100
$ make clean
$ cp * ../s999/.
$ cp .gitignore ../s999/.
```

Now, in **/exps/s999** you should have the following files:

- **Makefile**
  In this file there is a variable named 'TARGETS', currently assigned to 's100'. Change this to your experiment code, s999.

- **makefile_additional.inc**
  If you are not using AIDA in your experiment, in this file you should remove the line 'OBJS += unpack_aida.o'

- **s100.spec**
  This is the primary file of your unpacker, where you will define the structure of your experiment. Go ahead and rename this with your experiment code - s999.spec.

- **control.hh**
  The control header is required by UCESB, although it may not serve much of a purpose unless AIDA and/or other systems that require 'external' unpackers are used. **You can pretty safely not worry about this file.**

- **ext_unpacking.hh**
  In this header, remove the line referencing AIDA if you are not using it. Otherwise it can be ignored.

- **mapping.hh**
  This file defines the mapping of FRS subevents to their relevant detectors. Must be carefully set up on an experiment-by-experiment basis if the FRS is being used!

- **unpack_structures.hh**
  This file defines some useful structures for external unpacking; it can be safely ignored.

- **fatima_vme.spec**
  This file can be removed if FATIMA is not being used, with the VME DAQ.

- **unpack_aida.cc**
  This file can be removed if AIDA is not being used.

- **unpack_aida.hh**
  This file can be removed if AIDA is not being used.

This unpacker could now be built! Although, of course, it will be built to unpack data in the structure of s100, which may not match yours. If you do want to build it just to test, in **/exps** you can do:

    $ make s999 −j

or in **/exps/s999** you can do:

    $ make −j

8.2 FRS Mapping

The FRS is mapping in two stages: in UCESB itself modules are mapped to a physical detector name, while c4Root handles only the specific channel mappings where needed. This is to ensure the software is flexible enough to handle experiments spanning months and years, between which digitizers and crates may swap around and change configuration. The first stage of mapping occurs in the file **mapping.hh**. In general, mapping follows the pattern:

```
SIGNAL(RAW_NAME, UNPACK_STRUCTURE, DATA_TYPE);
```

where the 'unpack_structure' maps to some 'raw' structure and is of the type 'DATA_TYPE'. An example of the s100 mapping is shown below:

```
29    // ::::::: TPC Crate ::::::::::::::::::::::
30    // order preserved from FRS Go4
31    // 0: TPC21, 1: TPC22, 2: TPC23, 3: TPC24, 4: TPC24
32    // 7: TPC31
33    // 5: TPC41, 6: TPC42
34    SIGNAL(ZERO_SUPPRESS: TPC_ADC7_32); // 0 = S2, 1 = S3, 2 = S4
35    SIGNAL(TPC_ADC1_1, frstpc.data.v775.data[0], TPC_ADC1_8, frstpc.data.v775.data[7], DATA24);
36    SIGNAL(TPC_ADC2_1, frstpc.data.v775.data[8], TPC_ADC2_8, frstpc.data.v775.data[15], DATA24);
37    SIGNAL(TPC_ADC3_1, frstpc.data.v775.data[16], TPC_ADC3_8, frstpc.data.v775.data[23], DATA24);
38    SIGNAL(TPC_ADC4_1, frstpc.data.v775.data[24], TPC_ADC4_8, frstpc.data.v775.data[31], DATA24);
39    SIGNAL(TPC_ADC5_1, frstpc.data.v785.data[0], TPC_ADC5_8, frstpc.data.v785.data[7], DATA24);
40    SIGNAL(TPC_ADC6_1, frstpc.data.v785.data[8], TPC_ADC6_8, frstpc.data.v785.data[15], DATA24);
41    SIGNAL(TPC_ADC7_1, frstpc.data.v785.data[16], TPC_ADC7_8, frstpc.data.v785.data[23], DATA24);
42
```

Here data from some v775/v785 digitizers in one of the crates (the 'TPC crate') is mapped to structures called 'TPC_ADC'. In the first line, the TPC_ADC structure is defined as a 7 by 8 element array - it is also zero-suppressed. In the next line, the elements TPC_ADC[1][1] to TPC_ADC[1][8] are mapped to the first eight channels of the first v775 module. On the next line the elements TPC_ADC[2][1] to TPC_ADC[2][8] are mapped to the next eight channels; so on and so forth.

c4Root will then look for a structure named TPC_ADC, which is something that should not change between experiments, allowing for the flexibility to analyse multiple experiments as desired.

8.3   Structure Headers

The so-called 'structure header files' are those named in the form 'ext_h101_{subsystem}.h', located in **/c4source/{subsystem}**. These headers define the C-like structure that data from each system will be stored in after being unpacked by UCESB, in order to be interpreted by c4Root. If you change your unpacker in such a way that the data structures change, **you must regenerate the structure header**. This can be done via command or script. The script **gen_struct_header.py** in **/unpack-/scripts** can be used as: >python gen_struct_header.py {exp} {subsystem}. Otherwise, this script simply invokes the command:

>exp/exp --ntuple=UNPACK:{subsystem},NOTRIGEVENTNO,STRUCT_HH,
id=h101_{subsystem},ext_h101_{subsystem}.h

However, this script also copies the header to the correct **/c4source/{subsystem}** directory (so make sure everything is named correctly and consistently!). **NOTE: For now, please do not use this script until it is fixed.**

Using the FRS and mapping to a raw structure means that you must read and write 'RAW' level data with UCESB! In your Steering Macro the variable 'ntuple_options' must include RAW, and the FRS structure header must be generated with the option '--ntuple=RAW:frs'.

9   Troubleshooting and common issues

When compiling c4, sometimes you may encounter the error "UCESB not found, you will not be able to unpack lmd files". This will also not allow for a full compilation with 'make'.

This is because you have recently 'make clean'd UCESB and the necessary files are missing. Compile a project to fix this, and c4 will compile.
Do not use std::string type variables. ROOT trees cannot handle this - instead convert to TString to avoid invalid pointer errors when drawing from Trees.