LHCb-INT-2011-017

LPHE-2011-02

Configuration of TELL1 boards from XML database with PVSS

J. Luisier
École Polytechnique Fédérale de Lausanne

May 4, 2011

Abstract

Functionality of the TELL1 readout board for the LHCb Silicon Tracker is configured by downloading external files, which are not convenient to manipulate and are prone to be modified from one firmware version to the next. A C++ program has been developed to emulate the function of the TELL1 board, which can be used to extract the calibration constants. The configuration of this C++ program is provided by a XML file. The implementation of a PVSS component that allows to create a TELL1 configuration from this XML file is presented.

Contents

List of Figures

List of Tables

1 Introduction

The Silicon Tracker (ST) consists of more than 270'000 channels, read out by about 2'000 read-out chips, called Beetles. The analog data that comes out of the Beetles is digitalised neat the detectors and the digital signal is carried by optical fibres to 90 TELL1 boards[1] in an electronics hut away from the detector. The TELL1 boards perform the zero suppression on events selected by the Level 0 trigger (L0), and then send the data to the computer farm where events are reconstructed and selected by the High Level Trigger (HLT). The data processing in a TELL1 board is performed by *Field-Programmable Gate Arrays* (FPGA), referred to as PP-FPGA, which are programmable chips, allowing a parallel and fast processing. Each TELL1 board has more than 18'000 parameters that need to be set for a correct operation, parameters that will be discussed in Section 2.

Optimisation of the parameters is made using an emulator : a C++ program that has been developed in order to debug and test the TELL1 algorithms[2]. The TELL1 emulator and the TELL1 boards share about 17'000 parameters, which have to be consistently set up. The chosen solution was to create a XML database, similar to the detector description (also known as DDDB) and the detector conditions (also known as LHCBCOND). This new database is a private *slice*, which is meant to be used by people in charge of TELL1 algorithms or configuration only, and is known as COND. The emulator and the code to read and write the XML database run in the Vetra environment[3]. The XML database is discussed in Section 3.

The real TELL1 boards are usually configured with a PVSS *recipe*, which is a binary file that contains the name of the TELL1 registers and the associated value (i.e. the register content). In order to create these recipes it is presently required to configure the board and then read back all the registers. Doing this operation requires to have a direct connection to the TELL1 boards, and cannot be used when LHCb is collecting data. The initial configuration is performed by a text file, called 'config file' or 'cfg file', read by a dedicated function from the TELL1 c-code library[4]. The writing of these cfg files is not very convenient since there is no proper way to read them within PVSS, and moreover the parameters are identified by a line number, which may change when the TELL1 board firmware is updated. Therefore it was decided to find a way to use the XML database to build recipes, using the same XML file to configure both the emulator and the real TELL1 boards. The implementation of the PVSS component is presented in Section 4.

2 TELL1 parameters

The TELL1 parameters can be classified in two categories : the software-like and the hardware-related ones. The former set consists of the settings of the TELL1 algorithms, needed by both the emulator and the real TELL1 boards, and will be discussed in Section 2.1. The latter set consists of every settings that are needed for gigabit ethernet (GBE) communication, switching on/off optical receivers, controlling the throttle counters, etc. and will be discussed in Section 2.2. All of these parameters have to be written either in the PP-FPGA registers (which are 32 bits wide and 1 word deep) or PP-FPGA RAM (that are 32 bits wide and 128 words deep). A complete description of the parameters can be found in [5], here only a few examples are shown, presenting an overview of the number of parameters.

2.1 Software-like parameters

The main task of the TELL1 is to perform the zero-suppression of the data sent by the front-end electronic and to forward L0 selected data to the HLT farm. The zero-suppression is required in order to keep the needed bandwidth small for the transmission of the data to the HLT farm with a high rate of $\sim$ 2 kHz. The zero-suppression is performed in four different steps (each of them performed by a dedicated algorithm) :

- header correction,
- pedestal subtraction,
- common mode subtraction,
- clusterisation,

each of these algorithms and their key parameters are briefly described in the following. A more complete description of the algorithms is found in [6].

2.1.1 Header correction

It is observed that for a Beetle chip the noise of first channels of an analog port[1] is higher than the average value of the other channels. This is understood as a crosstalk from the header sent by the Beetle chip just before the data. The header consists of four pseudo bits, pseudo meaning that they are interpreted as '0' or '1', even if they are analog values. A way of correcting this effect was developed and correction values are extracted from data for each channel that needs to be corrected. These values are added to the received ADC for the corresponding channel. For each of the 96 analog ports, the first six channels are corrected, according to the value of the header. There are eight possible

[1] One TELL1 board reads out 24 Beetle chips, each of those having 4 analog ports, consisting in 32 channels. Thus 96 analog ports are read by a TELL1 board.

configuration for the header, therefore the number of header correction values is 4608 per TELL1 board, to which two thresholds have to be added. These thresholds are used to determine if the received pseudo-bit is a '0' or a '1'. The two thresholds are 8 bits integer, stored in one register per PP-FPGA, and the header corrections are 4 bits signed integer (from -8 to 7) and are written in a 32 bits wide and 16 words deep RAM blocks (twelve blocks per PP-FPGA).

2.1.2 Pedestal subtraction

The channel gain is adjusted such that the mean value of the received ADC value is around 128 (mid point of the ST dynamic range, which goes from 0 to 255). This mean value is known as *pedestal value*. The pedestal subtraction is done by subtracting an 8-bits number to the received ADC value, before any other processing is done, and results in an ADC distribution centred at zero (with valid values between -128 and 127). The pedestal values can be fixed or updated event by event. An outlier rejection is used when updating the pedestal value, to avoid an overestimation in presence of signal. Problematic channels, such as dead or noisy channels, have to be disabled in the data (or *masked*), in order to avoid fake clusters. This means that the ADC value for those channels will always be set to zero after pedestal subtraction. Each channel can be disabled by a single bit value, called *pedestal mask*. There is one pedestal value and one mask per channel, which makes 6144 values per TELL1 board. Both the mask (1 bit) and the value (8 bits unsigned integer) are stored in the same RAM block. The used RAM blocks are 64 words deep and 32 bits wide, and there are twelve of them per PP-FPGA.

2.1.3 Common mode subtraction

The noise pattern in a Beetle chip is known to look like a banana : the noise[2] is higher for the channels near the edges of the chip. Common mode subtraction is used to reduce this effect and results in a flat noise pattern, by using linear correction on the received ADC values. For each event, an average ADC value is computed and subtracted for each port (i.e. a set of 32 consecutive channels). A second average is then computed, from which the possible hits are excluded (i.e. the ADC is set to zero). Assuming a linear distribution of the remaining ADC values in the port is linear against the channel number a slope is computed (using least square method and setting to zero the ADC values of possible hits). The ADC values are then modified, so that a slope computed in the exact same way would be zero. The detection of possible hits is done using a 8 bits threshold per channel, called CMS threshold, above which the readout value is considered to be signal and set to zero when computing the second average value and the slope correction. Rejecting signal is important because in presence of signal the average would be overestimated, resulting in a lower signal value, and the slope would be either over- or underestimated, depending on the location (i.e. the channel number) of the signal. The 3072 values per TELL1 board are written in twelve RAM blocks per PP-FPGA.

[2]The *noise* is referring to the width of the ADC distribution in absence of signal, whilst the mean of the distribution is called *pedestal*, as mentioned in Section 2.1.2.

2.1.4 Clusterisation

When consecutive channels have signal, they are bundled together and form a *cluster*, which was likely to be generated by a single track. The clusterisation process relies on two thresholds : an inclusion threshold (called Hit threshold) and the Confirmation threshold. The algorithm is quite simple : it loops over the channels and start to build a cluster candidate if the ADC value of the channel is above the Hit threshold. A cluster comes to end when the ADC of the channel becomes below the Hit threshold or when the cluster already contains four channels. In order to decrease the rate of fake cluster due to noise, the total cluster charge (i.e. the sum of the ADC values) is required to be above the Confirmation threshold. One Hit threshold per channel and one Confirmation threshold per every set of 64 consecutive strips (called *processing channel*) is used. This makes in total 3168 values, including 48 SpillOver threshold values (again one per processing channel). This last threshold is used to set the SpillOver flag, used online to discriminate between real clusters and cluster from the previous bunch crossing (called SpillOver) : the SpillOver flag is set to 1 if the cluster charge is bigger than the SpillOver threshold. The 8 bits Hit threshold values are stored in twelve RAM blocks per PP-FPGA, and both Confirmation and SpillOver threshold values (8 bits each) are written in the same register (twelve per PP-FPGA).

This adds up to 16'994 needed constants. On top of these, there are other needed parameters for a TELL1 board to produce useful results[3], such as 'process info' settings, each of them enabling one of the four TELL1 algorithms.

2.2 Hardware-related parameters

Only the most important settings will be presented here, as there are many of them. An important subset of the hardware-related parameters is the collection of settings for the GBE ports of a TELL1 board. As an illustration of the complexity of writing into RAM blocks, the 16 words concerning the configuration of the first GBE port are given in Table 1. There are four ports, and the presented pattern is written four times (i.e. 64 words in total) in the same RAM block. Some very important parameters are present in there, such as source and destination MAC or IP addresses.

Another example are the bank class numbers. There is a number that identifies the bank class (error bank, pedestal bank, no zero suppressed bank, etc.) per detector type (such as IT, TT, VeLo, etc.). If these numbers are not set correctly, the event builder won't be able to decode the information.

The trigger settings are also controlled by the hardware-related parameters : the source of the trigger has to be specified. TELL1 boards can receive triggers either from 'Timing, Trigger and Control' (TTC), used for standard data taking (in global or stand-alone mode), or 'Experiment Control System' (ECS), which only allows stand-alone data taking.

[3]Even if the pedestal values and thresholds are incorrectly set, a TELL1 board will still send data, however it cannot be used for analysis.

	31	23	15	7
0	Dest. addr. [47:16]			
1	Dest. addr. [15:0]		Src. addr. [47:32]	
2	Src. addr. [31:8]			Src. addr. [7:0]
3	Ethernet type		IP vers. IHL	Svc type
4	Total length		Identification	
5	Flag	Frag. offset	TTL	Protocol
6	Header checksum		IP src. addr. [31:16]	
7	IP src. addr. [15:0]		IP dest. addr. [31:16]	
8	IP dest. addr. [15:8]	IP dest. addr. [7:0]	*Reserved*	
9-15	*Reserved*			

Table 1: GBE parameters, the blue field is set according to the port number (0, . . . , 3), the magenta ones are overwritten once the configuration is done, and are thus set to zero. Only settings for the first GBE port are shown in the table. The reserved words are not used and thus set to zero. The numbers between square brackets indicates which bits of the data field are written where.

If ECS trigger is used, the trigger type also has to be set, which will then determine the type of sent data (zero suppressed or non zero suppressed).

Checks on the received and processed data are performed by the TELL1 board, which require many thresholds. For instance the height of the header pseudo bits is not within an acceptable range, the TELL1 board sends an error bank together with the data.

The main difficulty with the hardware-related parameters is that a non negligible part is absent from the cfg file, but hardcoded in the tell1lib function that reads the cfg file and applies the configuration. This appeared to be a major problem when developing the component, since the only way to find these 'hidden settings' is to read every register once the TELL1 board is correctly configured, and compare the readings with the XML database. A good example of this consists in the discovery of the last missing setting : the ORxCtrlReg register (one per PP-FPGA) contains 8 bits that need to be set to 1 in order to enable the optical receivers. Since all the previous debugging was performed using the data generator, meaning that the TELL1 board processes a fake event and is actually not reading any data, this couldn't be found before testing *in situ*.

3 The XML database

```
<xml version="1.0" encoding="ISO-8859-1">
<!DOCTYPE DDDB SYSTEM "conddb:/DTD/structure.dtd">
<DDDB>
  <condition classID="9105" name="TELL1Board0">
    <!-- ... -->
  </condition>
  <condition classID="9105" name="TELL1Board1">
    <!-- ... -->
  </condition>
  <!-- ... -->
</DDDB>
```

Listing 1: Structure of the XML database file.

The XML database has the same structure as LHCBCOND : the file contains one *element node*[4] per TELL1 board, called **condition**, as shown on Listing 1. The **condition** *element node* has 2 mandatory *attribute nodes*, the **classID** and the **name** fields. The **name** consists of the string 'TELL1Board' to which the TELL1 board source ID is appended. The **classID** is used when the XML file is read within GAUDI. The allowed *children* for a **condition** *node* are **param** and **paramVector**, depending on how many values are stored in their *data text node*, i.e. if the child has only one value it's a **param**, otherwise it's a **paramVector**. The tags **param** and **paramVector** have mandatory *attributes* : **name**, **type** and **comment**. The **type** *attribute* can be 'int' for integer values or 'string' for character strings or integer given in hexadecimal representation. The **comment** *attribute* is present to help the user to understand what is the information stored in the *text node*. A few examples are given in Listing 2, a full example is given in Appendix A.

```
<condition classID="9105" name="TELL1Board0">
  <param name="Tell_name" type="string" comment="Name of the TELL1"> ittell01 </param>
  <paramVector name="ip_dest_addr" type="int" comment="Destination IP">
    192 168 196 131
  </paramVector>
  <paramVector name="mac_dest_addr" type="string" comment="Destination MAC address (hex)">
    0x00 0x01 0xE8 0x5D 0xE7 0x20
  </paramVector>
  <param name="TTC_dest_ip_available" type="int" comment="1=TTC, 0=ECS"> 1 </param>
</condition>
```

Listing 2: Example of the four use cases of **param** and **paramVector**, with 'int' or 'string' type.

The file is organised in 'sections', in which similar settings are grouped. The first section concerns miscellaneous parameters, such as the hostname of the TELL1 board and information about the TELL1 libraries (version and dates). The next section groups the communication parameters. The third part sets which processes are enabled, controls TTC, which set the expected trigger source and type, and forces or disables error

[4]For information about the XML terminology, please refer to W3schools website. In this document, xml terminology appear in *oblique type*, while name of element nodes or attributes are in **bold**.

banks sending, etc. The next section deals with ST specific parameters, this is where the thresholds, pedestal values and header correction values are stored. Disabling of the optical links and analog ports are also set here. The last part concerns the data generator : the TELL1 board can process and send data even if it is not connected front-end electronics. There are dedicated RAM blocks where a complete event is stored for this purpose. This last part of the XML database contains thus the ADC values of the 'fake' event.

Modifications of values in XML database file are mainly concerning the software-like parameters (discussed in Section 2.1), and new values are easily extracted from the data, using the STVetraAnalysis package (running in the VETRA environment). Most of the hardware-like parameters (see Section 2.2) are meant to remain the same.

Modification to the structure of the XML database file have to be performed very carefully, since there are several ways to produce and use such a file :

1. the TELL1 board itself can export its current configuration into XML,
2. the STVetraAnalysis package contains a script able to produce a XML database (with dummy values for software-like parameters),
3. the XML database can be read and written by classes from the VetraKernel package (in VETRA environment),
4. the PVSS component reads the XML database file to produce a recipe.

At any time the expected structure of the file has to be the same in the two C++ packages, the PVSS component and the tell1lib library.

4 Tell1XML framework component

4.1 Choosing an implementation

A TELL1 board had 341 used registers and RAM blocks, which might contain different values, whilst the XML database file had around 270 *element nodes*, and a way to relate each *element nodes* to the corresponding register(s) had to be found. When developing the component, finding a way to get this complicated mapping was the first issue. At the very beginning there was two diametrically opposed concepts : either the relations between the XML *element nodes* and the registers would be hardcoded in a library, or a way to perform this mapping in a dynamic way had to be found. The latter option was of course much better in terms of flexibility, but the implementation was not foreseen to be trivial and even discouraged by experts. However, this was the chosen implementation, since the leading principle was to be as generic as possible, i.e. to avoid hardcoding, which would be harder to achieve in the first place, but would be much more likely to be extended or modified. The choice was made to use a second XML file, which would contain the same *element nodes*, but the *data text nodes* would contain information about register instead of the settings (i.e. values), this file will be referred to as *mapping file*, a schematic illustration is shown on Figure 1. This allowed to read both information in the same way, because the *element node* names were the same as in the database file, which allowed to know how to pair the information obtained from the two files.

4.2 Generalities

The developed framework component is designed to work together with the general TELL1 framework component (called fwTell1[7]). The basic idea is to read the XML database used by the emulator and to produce directly a recipe from it. These recipes are binary files, in which a kind of an associative container links register names with the value to which they should be set. Some examples are given in Table 2.

Register	**Value**
PP0.ORx.ORxCtrlReg	0x3F000009
PP1.CMNCTRL.PPCtrlReg0	0xA0700342
SL.CMNCTRL.BankClassReg	0x191B140B

Table 2: Example of recipe content for a Inner Tracker (IT) TELL1 board.

The fwTell1XML component uses another component that allows to manipulate XML files in PVSS : fwCondXML, which is a library providing wrappers around the PVSS native XML DOM parser. Using this library allows to do many checks, on the syntax of the read XML file, the number of data fields in an *data text node*, the type, etc. It also allows to get the parameters directly with the correct type, since a parser only reads character strings, and thus a conversion to integer may be needed. The library uses a complex data container, which allows to access all the needed information from the XML file with indices.

```
<paramVector name="error_bank_disable" type="string"
             comment="Error bank disable 1=Disable error bank">
  i SL.CMNCTRL.SLCtrlReg0 0x00000400
</paramVector>
```

Listing 3: Example of a tag from the mapping file, the expected type is integer (i), the information will be written in bit number 10 (0x400) of SL.CMNCTRL.SLCtrlReg0.

A single register may contain several pieces of information, therefore care is needed to ensure that writing a setting does not overwrite any other one. The mask corresponding to the value is thus also given. The mask is an integer value, given in hexadecimal representation, that tells where the value has to be written in a register or a RAM block. For instance if the value 5 has to be written in a 32 bits register with the mask 0xE0, the register will contain the value 0x000000A0. In order to make it more robust against typos, misconfiguration or using incompatible versions, the expected type of the data from the database file is also given, as shown on Listing 3. This allows to check that the data type in the database file is the correct one.

4.3 The mapping file

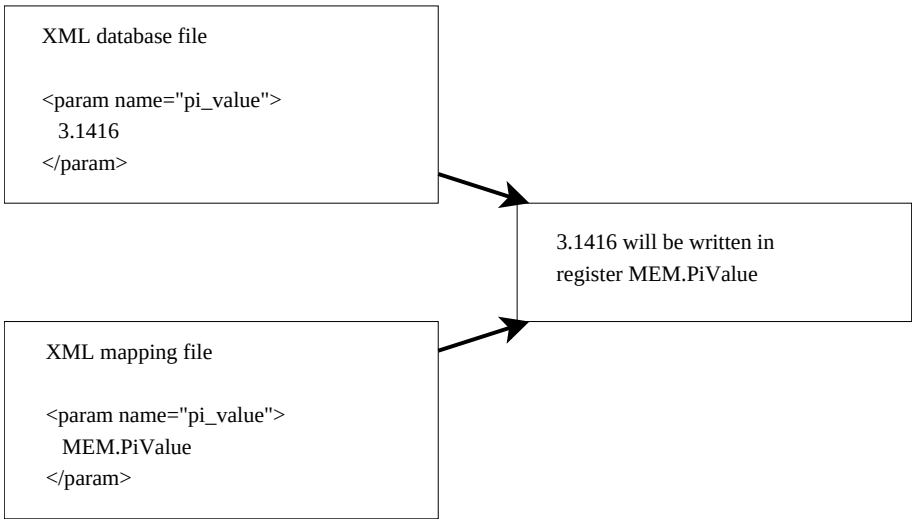

Figure 1: Illustration of the information flow in fwTell1XML : the XML database and the mapping file are read. Then the component knows that for the parameter named 'pi_value' the value 3.1416 has to be written in the register named MEM.PiValue.

The example given in Listing 3 shows a simple case : the integer value from the database has to be written in one register and the information is only 1 bit. The other and more complicated cases are :

- one piece of information has to be written in several registers,
- several pieces of information have to be 'packed' and written in one or more registers,

- information has to be written in a RAM block.

When one piece of information has to be written in several registers, the mapping file will contain something like :

```
data_type register_name_1 mask_1 register_name_2 mask_2 [...]
```

Actually there is no additional complication coming from the fact that several values have to be written in the same location. The n input values are equally and automatically distributed among the m registers, in the sense that the first $\frac{n}{m}$ values goes into the first registers, the next $\frac{n}{m}$ into the second one, etc. Only 4 use cases are known :

- $m = 1$, meaning that the n values will be put together in the same register.
- $\frac{n}{m} = 1$, meaning that each of the n value will go in its corresponding register.
- $n = 1$ and $m > 1$, for which the only value will be written m times in the m different registers.
- $\frac{n}{m} = k > 1$, meaning that the data is grouped in m subsamples, each of them written in its corresponding register.

But in any of these presented cases, the syntax of the mapping file node is the same, only the processing is different.

Concerning RAM blocks, everything gets more complicated. Usual registers of a TELL1 board are 1 word wide (1 word is 32 bits, i.e. 4 bytes), but RAM block are 1 word wide and x word deep. It can be seen as an array of usual registers, the only difference is that it is written in one go, meaning there is only one address. The way these data are handled in fwTell1XML is to build an array containing data_1, data_2, etc. But then arises the problem that multiple information can be written in the same RAM block, and they might not appear in the desired order (i.e. the order used in the XML file and the one in the recipe may be different). That is the reason why a new syntax is introduced : an index is added, and this index represents the position of the data in the array, which is related to its position in the RAM block. For consistency the index is given in hexadecimal representation, as the data from the mapping file *text nodes* are expected to be character strings. The corresponding mapping content is thus :

```
data_type register_name mask index
```

Two categories of tags can thus be found in the mapping file : the ones containing $2 \cdot k + 1$ elements, and the ones containing 4 elements, corresponding to RAM blocks.

4.4 Implementation

Creating a TELL1 recipe within PVSS is made calling the `fwHw_saveRecipe`[5] function from fwHw component. The prototype and documentation is given in Listing 4. From it can be deduced the needed information to be provided : `hw`, `recipeType` and `recipe` arguments are trivial. The list of register `regs` and their type `types` can be obtained using the recipe type and `fwTell1_getRecipeTypeInfo` function. The only thing that has to be built is the `data` structure. In order to handle 32 bits words, the `dyn_anytype` type has to be used in control script : a word is then an array of four character. The `data` object is thus the control script representation of an array of words. The core of fwTell1XML component is dedicated to the building of the `data` structure, putting everything in its right place.

```
/** This function creates and/or saves a Recipe (of a certain Recipe
 * Type) for a given hardware device.
 * If the optional parameters are not used, the values last written to
 * the hardware will be used as recipe data.  Otherwise the pairs
 * regs[i], data[i] will overwrite the last written data values (or
 * the last values in the recipe depending on the combineWithHw
 * parameter).
 *
 *  \param [in] hw name of the hardware device.
 *  \param [in] recipeType name of the recipe type.
 *  \param [in] recipe name of this recipe.
 *  \param [in] regs (optional) a new list of registers to be added or
 *  overwritten in the recipe.
 *  \param [in] types (optional) the list of registers types
 *  corresponding to the above registers.
 *  \param [in] data (optional) a new list of data items to replace
 *  the current last written values.
 *  \param [in] combineWithHw (optional) if 1 above data should
 *  replace the last written values (default) or if 0 the last recipe
 *  values.
 *  \return \e  1 if all went ok.
 *  \return \e -1 if error occurs in fwConfigurationDB functions.
 *              Note: see LogViewer for exception info
 */
int fwHw_saveRecipe( string hw, string recipeType, string recipe,
                     dyn_string regs = makeDynString(),
                     dyn_int types = makeDynInt(),
                     dyn_dyn_anytype data = makeDynAnytype(), int combineWithHw = 1 )
```

Listing 4: Prototype of fwHw_saveRecipe function, the doxygen documentation is also given.

As mentioned in Section 4.2, the fwCondXML component provides a data structure that stores the data from the read XML file. The component allows to be sure that the syntax of both the database and the mapping files are correct, but this requires knowing *a priori* what the file contains, i.e. the names of the *element nodes* have to be known. That is why the fwTell1XML component contains two libraries that describe the database and the mapping files. These two libraries are used to prepare the structure in which the information of the database and the mapping file will be stored. The list of the needed *element node* names has also to be provided in the main library, so that the information can be retrieved from the database and mapping files. The aforementioned list is built

[5] `Typewriter font` is used for function names, data types or objects, mainly in control script.

out of five arrays (corresponding to the five 'sections' of the database file), so that future insertion or removal of a piece of information is easy.

Once the data from the mapping file and the XML database file are loaded, the component has to combine these information into one recipe per TELL1. This is done by looping over the **param** and **paramVector** *nodes* of each **condition** *node*. The choice was made to split this loop according the the 'sections' in the database file. The main reason for this is that the first section contains information that are not written in the recipe, and the second one contains information that all go in the same RAM block (the GBE settings).

First the list of registers needed by the recipe type has to be collected, and then compared with the list of registers contained in the mapping file. If the two lists are not exactly matching, the processing is stopped. For any data written in the recipe, a set of functions are called, which first check that the data have the correct type (integer or string for hexadecimal numbers), apply the corresponding mask, and also check that the data is not too wide according to the mask[6]. Almost every processing step is checked and is able to interrupt the sequence in case of an error. All these functions are to be found in a third library[7]. Some conversion functions are also implemented here. This library also allows to read the TELL1 board configuration and write a XML file, but the process is very slow, due to a lot of unavoidable string manipulations.

The last object of the component is a small panel that allows the user to select the TELL1 boards for which a recipe has to be built, choose a name for the recipe and eventually create the wanted recipes. This panel appears in any ST TELL1 control unit (CU) panel, in the 'Configure via XML' tab.

The fwTell1XML component consists thus in the following four files :

- MappingCondXMLParserConfig.ctl, the library that configures what has to be read from the mapping file and defines the data structure that holds the mapping information,
- STCondXMLParserConfig.ctl, the library that configures what has to be read from the XML database file and defines the data structure that holds the settings value (for *every* TELL1 boards),
- fwTell1XML.ctl, the library that contains the list of used functions,
- Tell1_configFromXML.pnl, the panel used to create the recipes.

4.5 Initialisation of the XML data structures

In Section 4.4 are mentioned the two libraries that tells fwCondXML component what has to be read from the mapping and the XML database files. The syntax is the same in both

[6]For instance, the number 30 is too wide to be written on 4 bits, since 30 corresponds to 0x1E or 0b11110.

[7]The first two libraries describe the content of the database and mapping files.

cases, and will be described here. The structure type is quite complicated : it requires two string indices to access the information of the wanted *child* of a given **condition**, but the 'visible' type is `mapping`[8]. The libraries are responsible for creating the structures that will contain the information extracted from the XML files. The creation process has to be completely described in the library, in the following way : for every TELL1 board an entry (i.e. a **condition**) has to be added to the `mapping`, and every **param** or **paramVector** have to be added to each **condition**. This is done using two functions from fwCondXML, `sicondxml_addCondition` and `sicondxml_addParam`, a minimalist example is given in Listing 5. The structure that contains the mapping file information has only one condition, named "TEMPLATE", whilst the one containing XML database file information has up to 48 conditions for the Tracker Turicensis (TT). Each **condition** has 276 *children nodes*, which also have to be added by the libraries. The XML parser discards everything that is not expected, meaning a missing call to `sicondxml_addParam` implies a missing parameter, even if it is present in the XML database and mapping files.

4.6 Using the panel

The panel allows the user to perform all the wanted actions. As long as no XML file is read, the actions requiring one are disabled. When the panel is started, the available buttons are 'Get data from XML', 'Set input XML file' and 'Set output XML file' (see Figure 2). The latter one is used to set the name of the file that is written by the PVSS component from the TELL1 boards current configuration. 'Set input XML file' allows the user to choose which XML file should be used to build the recipes. The 'Get data from XML' button triggers the initialisation of the data structures and the reading of the mapping and XML files. As soon as this initialisation step is done without any errors, more actions can be performed.

Once the files are read, the user can select the TELL1 boards for which an action as to

[8] In control a `mapping` is an associative container that stores elements formed by the combination of a *key* value and a *mapped* value. It is equivalent to the `map` container in C++ or the Perl `hash`.

```
mapping myDataStructure;

string condName = "TELL1Board250";

sicondxml_addCondition( myDataStructure, condName );
// Now myDataStructure contains an empty condition named "TELL1Board250".

sicondxml_addParam( myDataStructure[ condName ], "lcms_enable", 's', 9 );
/*
 * Now myDataStructure[ "TELL1Board250" ] contains a "lcms_enable" field,
 * that contains 9 items of type string.
 */
```

Listing 5: Example of definition of a data structure that stores the information from a XML file. Here only one **condition** will be read (named 'TELL1Board250'), which contains only 1 field, called 'lcms_enable' of type string. The 'lcms_enable' field contains nine values.

be performed. The selection is done by using the green arrow buttons to copy a TELL1 board from the left list into the right one. Both lists can use TELL1 numbers or TELL1 source IDs. The former only allows to select the TELL1 boards controlled by the current CU, whilst the latter allows to select any TELL1 board. Switching from 'number mode' to 'source ID mode' is done by pressing the 'TELL1 nbr / src ID' button. The two versions of the right list are of course real-time synchronised.

When the list of TELL1 boards is chosen, recipes can be created by pushing the 'Create recipe(s)' button, or the current TELL1 boards configuration can be exported to XML[9]. For these two actions, some debug statements can be printed in the log, by checking the corresponding field on the bottom left of the panel.

The creation process consists of three steps :

1. initialise the data structures and read both XML files,
2. select the TELL1 board(s) for which a recipe has to be created,
3. fill the `dyn_dyn_anytype` and save the recipe(s).

The TELL1 board(s) selection can be done either by TELL1 board number (but only the TELL1 boards controlled by the current CU can be selected) or by TELL1 board source ID (which allows to select all the TELL1 boards of the subdetector). The action being currently performed by the panel are shown just below the 'Recipe Name' text field. The 'Debug statements' checkboxes allows to enable to print information in the log, in order to check that the various information are correctly handled. Each clickable or editable object on the panel also has some help information. The recipe name is set by the text field. It is mandatory that the recipe name ends with '/Configure', therefore this suffix is automatically added if not present. This means that on the example shown on Figure 2 the created recipe will be called 'TEST/Configure.

[9]This functionality is deprecated, the user is advised to use the TELL1 library function `dump_tell1_para_xml`.

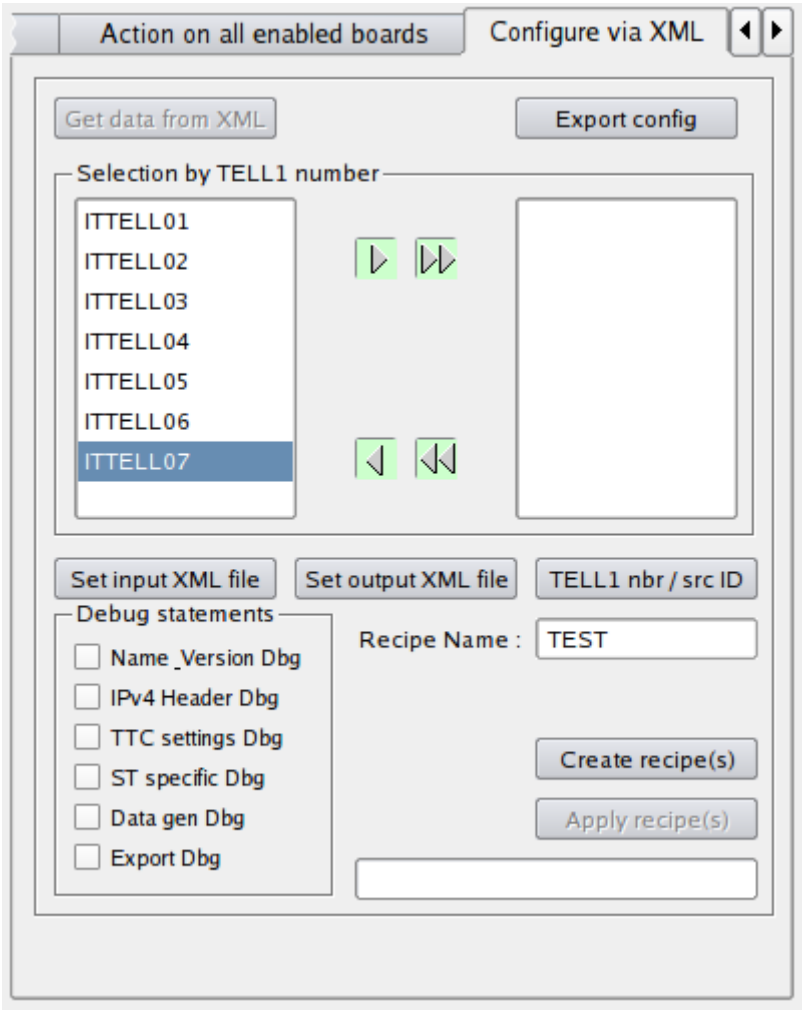

Figure 2: Overview of the panel. This picture shows the state of the panel after initialisation (i.e. once the XML files have been read).

5 Conclusion

After months of development and debugging, the fwTell1XML component is now installed in both IT and TT PVSS projects. After intensive tests, it was proven to be functional, and has been used as the default way to build recipes. This also allows to build recipes while the detector is running, which is a significant improvement compared to the configuration with cfg files. The chose implementation is now proven to be fully justified, as several registers had to be added in the database and mapping files during the development phase, procedure that went smoothly.

Again it has to be said that there are several disconnected way to read or write the XML database. A change in one place *must* be propagated to the other ones, so that everything keeps working. Apart from correcting possible remaining bugs, keeping the component up to date should be limited to adding / removing registers in the recipe.

6 Acknowledgement

The author wants to thank Rainer Schwemmer, who came with the idea about a mapping file in the first place, and spent several hours helping to debug the libraries. Guido Häfeli also receives the author's gratitude, for the help he provided understanding the TELL1 board and debugging the component. The final checks *in situ* could not have been performed without Frédéric Dupertuis, who spent several days checking every bit of the configuration, thanks to him. Finally Daniel Esperante is thanked for his advice and helping to understand the 'twisted logic' of PVSS.

References

[1] Legger, F and Bay, A and Haefeli, G and Locatelli, L, "TELL1 : development of a common readout board for LHCb," Tech. Rep. LHCb-2004-100. CERN-LHCb-2004-100. 1-2, CERN, Geneva, Nov 2004.

[2] Keune, A, "Emulation of the ST TELL1 Algorithms," Tech. Rep. , CERN, Geneva, 2011.

[3] Szumlak, T and Parkes, C, "Description of the Vetra Project and its Application for the VELO Detector," Tech. Rep. LHCb-2008-022. CERN-LHCb-2008-022, CERN, Geneva, May 2008.

[4] Haefeli, G., "tell1lib documentation." http://lphe.epfl.ch/tell1/tell1lib_c-code.htm.

[5] Haefeli, G. and Bay, A. and Gong, A. and Gong, H. and Muecke, M. and Neufeld, N. and Schneider, O., "The LHCb DAQ interface board TELL1," *Nuclear Instruments and Methods in Physics Research A*, vol. 560, pp. 494–502, May 2006.

[6] Haefeli, G, *Contribution to the development of the acquisition electronics for the LHCb experiment*. PhD thesis, EPFL Lausanne, Geneva, 2004.

[7] Potterat, Cédric and Bay, A, *Direct Search for Standard Model-Like Higgs Boson and Software Integration of Data Acquisition Cards*. PhD thesis, Lausanne, EPFL, Lausanne, 2010. Presented on 06 May 2010.

Listings

A XML database full example file

Here below is presented a 'full' example of a XML file generated by the TELL1 board. Parameters that occurs multiple times are not all present, and the long vectors (the Hit threshold and its 3072 values for instance) are truncated as well. Such a file is obtained by running the following command when connected onto a TELL1 board :

```
dump_tell1_param_xml
```

The TELL1 board ID, versions and GBE parameters are shown on Listing 6, then the TTC and processing information on Listing 7 and Listing 8, then the ST specific parameters (such as thresholds, pedestal values and mask, header correction values, ...) on Listing 9 and Listing 10, finally the data generator is shown on Listing 11.

```
<!--XML configuration for [lphe1tell1], Source_ID [7710], Created at:
[2000/07/03 20:19:08] -->
<condition classID="9105" name="TELL1Board7710">
  <!-- Misc information -->
  <param name="Tell_name" type="string" comment="Name of the TELL1">
    lphe1tell1
  </param>
  <param name="Tell_SN" type="string" comment="Serial Number of TELL1">
    0XFF4FFFFFFFFFFFFFFFFFFFFFFFF
  </param>
  <param name="Tell_UserPP_Ver" type="string" comment="The version of user logic for PP-FPGA">
    v4.2
  </param>
  <param name="Tell_CommPP_Ver" type="string" comment="The version of common logic for PP-FPGA">
    v4.2
  </param>
  <param name="Tell_CommSL_Ver" type="string" comment="The version of common logic for SL-FPGA">
    v4.2
  </param>
  <param name="Tell_PPcode_Gen_Date" type="string"
         comment="The generation date of PP FPGA code">
    16/06/2010--14:00
  </param>
  <param name="Tell_SLcode_Gen_Date" type="string"
         comment="The generation date of SL FPGA code">
    16/06/2010--13:42
  </param>
  <!-- Gigabit IP and Ethernet protocol parameters -->
  <paramVector name="ip_dest_addr" type="int" comment="Destination IP">
    128 180 89 133
  </paramVector>
  <paramVector name="ip_source_addr" type="int" comment="Source IP">
    128 178 89 4
  </paramVector>
  <paramVector name="mac_dest_addr" type="string" comment="Destination MAC address (hex)">
    0x00 0x07 0xE9 0x17 0x83 0x00
  </paramVector>
  <paramVector name="mac_source_addr" type="string"
        comment="Source MAC address last two are replaced (hex)">
    0x00 0x00 0x00 0x00 0x00 0x00
  </paramVector>
  <param name="ethernet_type" type="string" comment="Ethernet type (16b, hex)">
    0x800
  </param>
  <param name="ip_version" type="string" comment="IP version number (4b, hex)">
    0x04
  </param>
  <param name="IHL" type="string" comment="Internet header length in 32b words (4b)">
    0x05
  </param>
  <param name="service_type" type="string" comment="Type of desired service for IP packets (8b)">
    0x00
  </param>
  <param name="TTL" type="string" comment="Time in seconds for IP packet to stay in the Ethernet (8b)">
    0x00
  </param>
  <param name="next_level_protocol" type="string" comment="Next level protocol (8b)">
    0xF1
  </param>
  <param name="gbe_forced_stop_cycle" type="string" comment="GBE flow control">
    0x00
  </param>
```

Listing 6: Miscellaneous information and GBE parameters.

```
<!-- General parameters defining TTC and processing operation -->
<param name="DETECTOR_ID" type="int" comment="Detector ID (1-VeLo, 2-ST, ...)">
  2
</param>
<param name="pedestal_bank_schedule" type="int"
       comment="Pedestal bank schedule 1=ped bank always accompagnies NZS bank">
  8
</param>
<param name="TTC_info_enable" type="int"
       comment="1=TTC trigger used, 0=triggers and all info sent by ECS">
  0
</param>
<param name="External_trigger_input_enable" type="int"
       comment="1=Ext trigger used, 0=disable Ext trigger">
  0
</param>
<param name="TTC_trigger_type_available" type="int" comment="1=TTC, 0=ECS">
  0
</param>
<param name="TTC_dest_ip_available" type="int" comment="1=TTC, 0=ECS">
  0
</param>
<param name="detector_data_generator_enabled" type="int"
       comment="Detector data generator enable">
  0
</param>
<param name="SEP_generator_enabled" type="int" comment="SEP generator enable">
  0
</param>
<param name="pp_derandmizer_event_threshold" type="int"
       comment="PP derandomizer event usage threshold">
  32
</param>
<param name="mep_pid" type="string" comment="MEP PID (hex)">
  0xEDED1D1D
</param>
<param name="source_id" type="string" comment="Source ID (16b, hex)">
  0x1E1E
</param>
<param name="version" type="string" comment="Version (8b, hex)">
  0x1E
</param>
<param name="zs_class" type="string" comment="Class of ZS bank (8b, hex)">
  0x01
</param>
<param name="non_zs_class" type="string" comment="Class of NZS bank (8b, hex)">
  0x02
</param>
<param name="pedestal_class" type="string" comment="Class of pedestal bank (8b, hex)">
  0x03
</param>
<param name="error_class" type="string" comment="Class of error bank (8b, hex)">
  0x04
</param>
<param name="FE_data_check_enable" type="string"
       comment="FE data check (1b, hex) 1=Check, empty event sent if no FE data">
  0x00
</param>
<param name="serious_err_throttle_enable" type="string"
       comment="Enable/disable serious error throttle (1b, hex), 1=Enable">
  0x00
</param>
<param name="nzs_bank_period_factor" type="string"
       comment="NZS bank period (32b, hex) 0=Disable periodic NZS bank">
  0x00
</param>
<param name="nzs_bank_offset" type="string" comment="NZS bank offset (9b, hex)">
  0x00
</param>
```

Listing 7: TTC settings and processing operation parameters (1).

```
<param name="trigger_n" type="string" comment="Trigger number in each DAQ loop (24b, hex)">
  0x000100
</param>
<param name="consecutive_n" type="int" comment="Consecutive trigger length, max=16">
  1
</param>
<param name="wait_cycle" type="int" comment="Wait cycles min=36 (16b)">
  48
</param>
<param name="TRIGGER_TYPE" type="int" comment="ECS trigger type 5=NZS, other=physics">
  0
</param>
<param name="error_bank_disable" type="int" comment="Error bank disable 1=Disable error bank">
  0
</param>
<param name="force_info_disable" type="int"
       comment="Info bank disable 1=Disable error bank at PP">
  0
</param>
<param name="force_info_enable" type="int" comment="Info bank enable 1=Force error bank at PP">
  0
</param>
<param name="nzs_for_error_enable" type="int" comment="NZS in case of error 1=enable">
  0
</param>
<param name="detector_specific_bank_header_enable" type="int"
       comment="Enable specific bank header which is used for ST, VeLo, OT, MUON">
  0
</param>
<param name="MEP_FACTOR" type="int" comment="ECS mep factor, max=32">
  8
</param>
<param name="MTU_SIZE" type="int" comment="MTU size for the Ethernet Default=1500, Max=8192+20">
  1500
</param>
<param name="HLT_PORT" type="string" comment="GBE port select 4LSB for 4 ports (hex)">
  0x02
</param>
<param name="HLT_DP_EN" type="int" comment="HLT data flow enable">
  1
</param>
<param name="MEP_USE_THR" type="string" comment="Threshold for MEP buffer">
  0x70000
</param>
<param name="Thottle_PPx_en" type="string" comment="Threshold Enable for each PP">
  0xF
</param>
<param name="Thottle_MEP_buffer_en" type="int" comment="Enable for MEP buffer throttle source">
  1
</param>
<param name="trigger_info_throttle_en" type="int"
       comment="Enable for trigger info throttle source">
  1
</param>
<param name="throttle_en" type="int" comment="Overall throttle enable">
  1
</param>
<param name="Firmware" type="string" comment="Expected firmware version x.y => XY (hex)">
  0x00
</param>
<param name="TTCRx" type="string" comment="TTCRx.Control magic value (hex)">
  0xBB
</param>
```

Listing 8: TTC settings and processing operation parameters (2).

```
<!-- ST specific part -->
<param name="Pseudo_header_lo_thr" type="string" comment="Low threshold for Pseudo header">
  0x90
</param>
<param name="Pseudo_header_hi_thr" type="string" comment="High threshold for Pseudo header">
  0xA0
</param>
<param name="zerosuppress_enable" type="int"
       comment="ZS Enable: used in PS and Clustermaker process">
  1
</param>
<param name="header_correction_enable" type="int" comment="Header Enable: used in PS process">
  0
</param>
<param name="header_corr_threshold_lo" type="int"
       comment="Lower header threshold: used in PS process">
  0
</param>
<param name="header_corr_threshold_hi" type="int"
       comment="Upper header threshold: used in PS process">
  0
</param>
<paramVector name="header_corr_value_00" type="int"
       comment="Lower and upper correction per link: used in PS process">
  1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
<!-- Values for links 1 - 94 -->
<paramVector name="header_corr_value_95" type="int"
       comment="Lower and upper correction per link: used in PS process">
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
<param name="pedestal_enable" type="int" comment="Pedestal Enable: used in PS process">
  1
</param>
<param name="pedestal_auto_update_enable" type="int"
       comment="Update Enable: used in PS process">
  0
</param>
<param name="lcms_enable" type="int" comment="CMS Enable: used in CMS process">
  0
</param>
<param name="cluster_number_max" type="int"
       comment="Max number of clusters per PP: used in ZS process">
  128
</param>
```

Listing 9: ST specific parameters (1).

```
<paramVector name="orx_beetle_disable_0" type="int"
        comment="Optical link disabled (1) or enabled (0) for links 0 to 11">
  1 1 1 1 1 1 1 1 1 1 1 1
</paramVector>
<paramVector name="orx_beetle_disable_1" type="int"
        comment="Optical link disabled (1) or enabled (0) for links 12 to 23">
  1 1 1 1 1 1 1 1 1 1 1 1
</paramVector>
<paramVector name="orx_lck_ref_0" type="int"
        comment="All 24 optical links need this flag to be set 1 for proper operation">
  1 1 1 1 1 1 1 1 1 1 1 1
</paramVector>
<paramVector name="orx_lck_ref_1" type="int"
        comment="All 24 optical links need this flag to be set 1 for proper operation">
  1 1 1 1 1 1 1 1 1 1 1 1
</paramVector>
<param name="orx_tlk_en" type="int"
        comment="All en set to 1, tlk ch 0..5 en (bit0 and bit3 of orx_ctr_reg">
  9
</param>
<paramVector name="arx_beetle_disable_0" type="string"
        comment="Analog port disabled/enabled, 4 bits per beetle, for links 0 to 11">
  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
</paramVector>
<paramVector name="arx_beetle_disable_1" type="string"
        comment="Analog port disabled/enabled, 4 bits per beetle, for links 12 to 23">
  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
</paramVector>
<paramVector name="pcn_select" type="string"
        comment="Select which PCN is sent with the ZS bank 0...5 per PP (hex)">
  0x00 0x00 0x00 0x00
</paramVector>
<paramVector name="pedestal" type="int"
        comment="Initial pedestal values per strip: used in PS process">
  3072 integer values between 0 and 255
</paramVector>
<paramVector name="pedestal_mask" type="int" comment="Masks strips to be skipped in PS process">
  3072 integer values, either 0 or 1
</paramVector>
<paramVector name="hit_threshold" type="int"
        comment="Hit thresholds per strip: used in ZS process">
  3072 integer values
</paramVector>
<paramVector name="cms_threshold" type="int"
        comment="CMS thresholds per strip: used in CMS process">
  3072 integer values
</paramVector>
<paramVector name="spill_over_threshold" type="int"
        comment="SpillOver thresholds (48): used in ZS process">
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
<paramVector name="confirmation_threshold" type="int"
        comment="Confirmation thresholds (48): used in ZS process">
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
<paramVector name="start_strip_n_0" type="int"
        comment="The start_strip_n channel 0-23, values per Process_channel">
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
<paramVector name="start_strip_n_1" type="int"
        comment="The start_strip_n channel 24-47, values per Process_channel">
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
</paramVector>
```

Listing 10: ST specific parameters (2).

```
  <!-- Data RAM generator -->
  <!-- PP 0 -->
  <paramVector name="st_data_gen_array_0_0_0" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link0, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_0_1" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link1, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_0_2" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link2, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_0_3" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link3, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <!-- Similarly for beetle 1, 2, 3, 4 -->
  <paramVector name="st_data_gen_array_0_5_0" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link0, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_5_1" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link1, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_5_2" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link2, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <paramVector name="st_data_gen_array_0_5_3" type="int"
        comment="optical data generator RAM, PP0, Beetle 0, link3, words 0-35, values per strip, header included">
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  </paramVector>
  <!-- Same for PP 1, PP 2 and PP 3 -->
</condition>
```

Listing 11: Data RAM generator.

B fwTell1XML documentation

In this section are given the prototypes of the functions implemented in fwTell1XML library (Section B.1.1) and the GUI panel (Section B.2.1). For each of them, the documentation *à la* doxygen is provided, which often means that a tiny example is given. The list of global variables used in the library (Section B.1.2) and in the panel (Section B.2.2) is also given.

B.1 fwTell1XML.ctl library

B.1.1 Functions

```
/** This function creates the list of the tag names present in the ST condition database.
 *
 * It will set the boolean ListInitialised to "true" once done. The list is used to read
 * the xml files (the mapping one and the database have the same tags). If the xml are
 * changed, only this list has to be updated.
 *
 * The implementation changed into something more generic : 5 different arrays are built
 * in a first step, then each of them is stored in an array of arrays. The flat list is
 * also created by appending all the sub arrays into one.
 *
 * Once this function is executed, you can use :\n
 * - FWTELL1XML_SmartTagList, which is the array of arrays.\n
 * - FWTELL1XML_ListOfTags, which is the flat array.\n
 * - FWTELL1XML_NbrOfTags, which is the number of tags.
 *
 * \b Example :
 *
 * \code
 * if (!ListInitialised)
 * {
 *   fwTell1XML_initialiseListOfTags();
 * }
 *
 * \endcode
 *
 */
void fwTell1XML_initialiseListOfTags()
```

Listing 12: Prototype of `fwTell1XML_initialiseListOfTags` function, with the doxygen documentation.

```
/** This function gets the list of the existing registers in a ST-TELL1, according to
* the recipe type.
 *
 * The function just calls fwTell1_getRecipeTypeInfo to get the list of the registers,
 * based on the TELL1 type (given by the TELL1 name) and the recipe type (which is
 * hardcoded to be ST_TELL1).
 *
 * The resulting list is not sorted, so that the order of the register in the recipe
 * is correct.
 *
 * Once executed, this function allows you to use the following variable :\n
 * - FWTELL1XML_RegisterList, which is the register list, got from the recipe type.\n
 * - FWTELL1XML_NbrOfRegisters, which is the number of registers.
 *
 * @param [in] tell1type type of the TELL1
 * @return \e 1 if everything went fine
 *         \e -1 if an error occurred
 *
 */
int fwTell1XML_createList(string tell1type)
```

Listing 13: Prototype of `fwTell1XML_createList` function, with its doxygen documentation.

```
/** This function checks if the list of registers needed by the recipe and the one got
 * from the xml are the same.
 *
 * The checks are performed in several steps, each of them can break the sequence
 * if not OK.\n
 *   - Create the register list from the recipe type, if not done before\n
 *   - Create the register list from the xml mapping file.\n
 *   - Compare the sizes of the two arrays.\n
 *   - Compare the register names one by one (once they are sorted).\n
 *
 * @param [in] Map is the information stored in the database, ie the mapping between
 *             the tags and the register names.
 * @return \e 1 if everything went fine
 *         \e -1 if an error occured
 *
 */
int fwTell1XML_checkList(mapping & Map, string dp)
```

Listing 14: Prototype of `fwTell1XML_checkList` function, with its doxygen documentation.

```
/** This function prepare the structure that is needed to handle network settings.
 *
 * This is done so because all the data related to the network go into the same memory,
 * which is 32-bits wide and xx words depth. In order to simplify the processing of
 * these data, they are ordered in an array, and then concatenated and sliced into
 * 32-bits words.
 *
 * Apart from the reserved bits, all the values defined here will be overwritten.
 *
 * @param [out] data Contains 'default' data, is all is set to "0"
 * @param [out] mask Contains the masks, "0" for read-only.
 * @return \e the length of the arrays
 *
 */
int fwTell1XML_prepareHeader(dyn_string & data, dyn_string & mask)
```

Listing 15: Prototype of `fwTell1XML_prepareHeader` function, with its doxygen documentation.

```
/** This function returns the position of a given register in the list of all registers.
 *
 * This is done in order to ensure the data and mask are related to the right register.
 * The registers order is given by the register list, and the data / mask byte array
 * is ordered according to that.
 *
 * @param [in] tagName the name of the TELL1 register.
 * @return \e -1 if the name is not found
 *         \e the index
 *
 * \b Example :
 * \code
 * string reg = "PP0.CMNCTRL.PPCtrlReg0";
 *
 * int index = fwTell1XML_findIndex(reg);
 *
 * if (index < 0)
 * {
 *   DebugTN("Something very bad happened");
 * }
 * \endcode
 *
 */
int fwTell1XML_findIndex(string tagName)
```

Listing 16: Prototype of `fwTell1XML_findIndex` function, with its doxygen documentation.

```
/** This function takes the data and the mask, and fills the corresponding fields of
 * the byte array.
 *
 * The function checks first that data and mask have the same size. Then the hexadecimal
 * strings are converted into byte array, and put at the right position in the global
 * byte array, which contains all the information to configure a TELL1 board.
 *
 * @param [in] data The data in hex string
 * @param [in] mask The mask in hex string
 * @param [in] Merged The byte array, containing data and masks DDDDMMMM, DDDDMMMM, ....
 * @param [in] index Corresponds to the register name, converted into the position in
 *             the register list.
 * @param [in] debug Toggles On/Off some print out
 * @return \e -1 if an error occured
 *         \e 1 if everything went fine
 *
 * \b Example :
 *
 * \code
 * dyn_string confData, confMask;
 * (...)
 *
 * if (fwTell1XML_prepareStructure(confData, confMask, position) < 0)
 * {
 *   DebugTN("Something nasty happened : cannot configure register nbr : " + position);
 *   return -1;
 * }
 *
 * \endcode
 *
 */
int fwTell1XML_prepareStructure(dyn_string & data, dyn_string & mask,
                                dyn_dyn_anytype & Merged, int index, bool debug = false)
```

Listing 17: Prototype of `fwTell1XML_prepareStructure` function, with its doxygen documentation. This function is not used anymore, since the function creating recipes for TELL1 boards doesn't expect any information about the mask.

```
/** This function takes the data and the mask, and fills the corresponding fields of
 * the byte array.
 *
 * It doesn't act the same as fwTell1XML_prepareStructure, in the sense that here
 * the mask is not appended to the data in the byte array (ie the structure is not
 * DDDDMMMM).
 *
 * The function checks first that data and mask have the same size. Then the
 * hexadecimal strings are converted into byte array, and put at the right position
 * in the global byte array, which contains all the information to configure a TELL1
 * board.
 *
 * @param [in] data The data in hex string
 * @param [in] mask The mask in hex string
 * @param [in] MergedData The byte array containing only data
 * @param [in] MergedMask The byte array containing only masks
 * @param [in] index Corresponds to the register name, converted into the position
 *             in the register list.
 * @param [in] debug Toggles On/Off some print out
 * @return \e -1 if an error occured
 *         \e 1 if everything went fine
 *
 * \b Example :
 *
 * \code
 * dyn_string confData, confMask;
 * dyn_dyn_anytype recipeData, recipeMask;
 * (...)
 *
 * if ( fwTell1XML_prepareStructureNoMask( confData, confMask, recipeData,
 *      recipeMask, position ) < 0 )
 * {
 *   DebugTN("Something nasty happened : cannot configure register nbr : " + position);
 *   return -1;
 * }
 *
 * \endcode
 *
 */
int fwTell1XML_prepareStructureNoMask(dyn_string & data, dyn_string & mask,
                                      dyn_dyn_anytype & MergedData,
                                      dyn_dyn_anytype & MergedMask,
                                      int index, bool debug = false)
```

Listing 18: Prototype of `fwTell1XML_prepareStructureNoMask` function, with its doxygen documentation. Similar to `fwTell1XML_prepareStructure`, but the mask information isn't appended to the data in the output structure. Instead, the mask and the data are merged into different arrays.

```
/** This function computes how wide is the mask in terms of bits.
 *
 * @param [in] hex The hexadecimal mask string
 * @return \e the number of bits
 *
 * \b Example :
 *
 * \code
 * string mask = "00003FC00";
 *
 * DebugTN(fwTell1XML_getNumberOfBits(mask)); // Will print 8
 *
 * \endcode
 *
 */
int fwTell1XML_getNumberOfBits(string hex)
```

Listing 19: Prototype of `fwTell1XML_getNumberOfBits` function, with its doxygen documentation.

```
/** This function rewrites the hex data string taking into account the mask.
 *
 * The function extracts the least significant non-zero bit and compute
 * an 'offset factor'. The data are the multiplied by this 'offset factor'.
 *
 * @param [in/out] data The original data string and the modified one.
 * @param [in] mask The mask applied to the data (with the '0x' prefix).
 * @return \e -1 if an error occurred
 *         \e 1 if everything went fine
 *
 * \b Example :
 *
 * \code
 *
 * string data = "1", mask = "0x80000000";
 *
 * fwTell1XML_writeCorrectByte(data, mask);
 *
 * DebugTN(data, mask); // prints out [80000000][80000000]
 *
 * \endcode
 *
 * \n
 * \n
 */
int fwTell1XML_writeCorrectByte(string & data, string mask)
```

Listing 20: Prototype of `fwTell1XML_writeCorrectByte` function, with its doxygen documentation.

```
/** This functions gets the xml values of a given tag and dumps them into one
 * string array.
 *
 * The function converts automatically the int parameters into hex strings. If the
 * value is negative, the value is first transformed into 256 - value, in order to
 * avoid '-1' to be written as 'FFFFFFFE'. The string parameters are just read,
 * and the prefix '0x' is removed.
 *
 * @param [in] Map Contains the xml information.
 * @param [in] condName The name of the TELL1 (not the physical name...).
 * @param [in] tagName The tag of the wanted value(s).
 * @param [in] type The type of the wanted data, can be either 'i' for int
 *             or 's' for string.
 * @param [out] values The conditions we want.
 * @return the size of the array which contains the values.
 *
 */
int fwTell1XML_getRawData(mapping & Map, string condName, string tagName, string type,
                          dyn_string & values)
```

Listing 21: Prototype of `fwTell1XML_getRawData` function, with its doxygen documentation.

```
/** This function is called to create a mapping between the TELL1 source ID and
 * the ccpc name.
 *
 * @param [in] cond the conditions
 * @param [out] names the source ID - ccpc name mapping
 * @return the size of the mapping
 *
 * \b Example :
 *
 * \code
 *
 * initializeCondXMLParser(conditions);
 *
 * mapping NamesAndAliases;
 *
 * fwTell1XML_getNamesMapping(conditions, NamesAndAliases);
 *
 * for (unsigned i = 1; i <= mappinglen(NamesAndAliases); i++)
 * {
 *   DebugTN(mappingGetKey(NamesAndAliases, i) + " -> "+
 *                         mappingGetValue(NamesAndAliases, i));
 * }
 *
 * \endcode
 *
 */
int fwTell1XML_getNamesMapping(mapping & cond, mapping & names)
```

Listing 22: Prototype of `fwTell1XML_getNamesMapping` function, with its doxygen documentation.

```
/** This function is called to create a xml file of the present configuration of
  * the TELL1.
  *
  * Here the TELL1 registers are read in the order given by the mapping file. The names
  * of the TELL1s are read from the Mapping of the names. The output is a regular XML
  * file, which content is similar to the TELLCond.xml file.
  *
  * @param [in] Map is the structure corresponding to the mapping file Mapping.xml
  * @param [in] nameMapping contains the TELL1 source ID vs TELL1 name relation (which is
  *             is read from TELLCond.xml).
  * @param [in] condName is TELL1Boardxx, where xx is the source ID.
  * @param [out] readbackInfo is the data structure we use to store the xml information.
  * @param [in] debug enables the verbose output, ie each xml line will be printed
  * @return 1 is everything went fine, -1 in case of error.
  *
  */
int fwTell1XML_writePresentConfiguration(mapping & Map, const mapping & nameMapping,
                                         string condName, mapping & readbackInfo,
                                         bool debug)
```

Listing 23: Prototype of `fwTell1XML_writePresentConfiguration` function, with its doxygen documentation. This function should be used any more, since the TELL1 board itself can export its present configuration into a XML file

```
/** This function allows to read properly a data field in a x-byte wide word.
  *
  * It first makes an "AND" with the mask and then removes the meaningless '0'.
  *
  * @param [in] data the array of byte array wich contains data
  * @param [in] mask the byte array of the mask
  * @param [out] strData the correctly formatted array of data, in hex
  * @param [out] strMask the correctly formatted array of mask, in hex
  *
  * \b Example :
  *
  * \code
  */
int fwTell1XML_readCorrectByte(dyn_dyn_char data, dyn_char mask, dyn_string & strData,
                               dyn_string & strMask)
```

Listing 24: Prototype of `fwTell1XML_readCorrectByte` function, with its doxygen documentation.

```
/** This function formats the information got from TELL1 into xml data value.
  *
  * The data are sliced if needed and then put in the corresponding xml field,
  * via CondXMLWriter library functions. The resulting line is also printed.
  *
  * @param [in] data contains the information read from the register, converted into
  * hex strings
  * @param [in] mask contains the mask, read from the mapping file.
  * @param [out] output the data structure that will be used to write the xml file
  * @param [in] condName name of the condition that will be written (i.e. "TELL1Board0",
  *             ...)
  * @param [in] tagName name of the written tag (ie "pedestal_auto_update_enable",
  *             "Header_low_threshold", ...)
  * @return \e -1 one if an error occured,
  *         \e 1 if everything went fine.
  *
  */
int fwTell1XML_setRawData(dyn_string data, dyn_string mask,
                          mapping & Map, mapping & output,
                          string condName, string tagName, bool debug)
```

Listing 25: Prototype of `fwTell1XML_setRawData` function, with its doxygen documentation. This function is called by `fwTell1XML_writePresentConfiguration` and shouln't be used.

```
/** Converts Hex string into binary string.
  *
  * This is just a stupid function, but it prevents to write things like
  * \code
  * fwCcpc_convertDecToInt( fwCcpc_convertHexToDec( "FA" ) );
  * \endcode
  *
  * @param [in] pHex hexadecimal string
  * @return binary string
  */
string fwTell1XML_convertHexToBin( string pHex )
```

Listing 26: Prototype of `fwTell1XML_convertHexToBin` function, with its doxygen documentation.

```
/** Converts binary string into Hex string.
  *
  * This is just a stupid function, but it prevents to write things like
  * \code
  * fwCcpc_convertDecToHex( fwCcpc_convertBinToDec( "1000101" ) );
  * \endcode
  *
  * @param [in] pBin bin string
  * @return hex string
  */
string fwTell1XML_convertBinToHex( string pBin )
```

Listing 27: Prototype of `fwTell1XML_convertBinToHex` function, with its doxygen documentation.

B.1.2 Global variables

```
// Variable that indicates whether the list of registers has been retrieved from the hw
// function (\e true) or not yet (\e false).
global bool FWTELL1XML_ListInitialised = false;
// Array that contains the names of the registers that will be written.
global dyn_string FWTELL1XML_RegisterList = makeDynString();
// Variable that holds the number of registers to be written.
global int FWTELL1XML_NbrOfRegisters = 0;
```

Listing 28: List of variables related to register management.

```
// Recipe type, needed by fwTell1_getRecipeTypeInfo.
global string FWTELL1XML_recipeType = "TELL1_ST";
// Name of the system, is initialised by the CU panel, either "IT" or "TT"
global string FWTELL1XML_systemName = "";
```

Listing 29: List of variables related to type of the detector and recipe.

```
// Variable that indicates whether the list of tags has been built from the library
// function (\e true) or not yet (\e false).
global bool FWTELL1XML_ListTagsInit = false;
// List of all tags in a flat list (the order corresponds to the order in the
// XML file).
global dyn_string FWTELL1XML_ListOfTags;
// List of all tags, organised in sublist, which are corresponding to the XML
// database "sections". There are 5 categories, each of them processed by a
// dedicated function from the panel.
global dyn_dyn_string FWTELL1XML_SmartTagList;
// Variable that stores the number of tags that need to be read.
global int FWTELL1XML_NbrOfTags = 0;
```

Listing 30: List of variables related to tag management.

```
// Common version of cfg file.
global int FWTELL1XML_CMNVERSIONNBR = 26;
// Specific version of cfg file.
global int FWTELL1XML_STVERSIONNBR = 6;
```

Listing 31: List of variables related to cfg version management.

B.2 Tell1_configFromXML.pnl panel

B.2.1 Functions

```
/**
  * Update the TELL1 list from the list given by TELL1 number.
  */
void updateListFromNbr()
```

Listing 32: Prototype of `updateListFromNbr` function, with its doxygen documentation.

```
/**
  * Update the TELL1 list from the list given by TELL1 source ID.
  */
void updateListFromSrc()
```

Listing 33: Prototype of `updateListFromSrc` function, with its doxygen documentation.

```
/**
  * Function used to enable/disable buttons at start, when only the initButton,
  * is enable, or after initialisation.
  *
  * @param [in] state state of buttons.
  */
void setActive(bool state)
```

Listing 34: Prototype of `setActive` function, with its doxygen documentation.

```
/**
  * Allows to switch from Number to Source ID view.
  */
void switchDisplay()
```

Listing 35: Prototype of `switchDisplay` function, with its doxygen documentation.

```
/**
  * Enables or disables the buttons, lists related to TELL1 number.
  *
  * Is called by switchDisplay().
  *
  * @param [in] state state of buttons.
  */
void setByNumberDisplay(bool state)
```

Listing 36: Prototype of `setByNumberDisplay` function, with its doxygen documentation.

```
/**
  * Enables or disables the buttons, lists related to TELL1 source ID.
  *
  * Is called by switchDisplay().
  *
  * @param [in] state state of buttons.
  */
void setBySourceIDDisplay(bool state)
```

Listing 37: Prototype of `setBySourceIDDisplay` function, with its doxygen documentation.

```
/**
  * Initialises the data structure that contains the information that will be
  * written in the recipe. A full clearing of the array is made.
  */
void initDataStructure()
```

Listing 38: Prototype of `initDataStructure` function, with its doxygen documentation.

```
/**
  * Extracts the TELL name from the XML database.
  *
  * If \c debug is set to \e true, the name of the TELL1 is printed in the log,
  * as well as the other information contained in the corresponding XML
  * "section".
  *
  * @param [in] CondName name of the condition, i.e. "TELLBoard0".
  * @param [out] TELL1Name name of the corresponding DU (uppercase version of
  *                        the ccpc name).
  * @param [in] debug toggles degug print out.
  *
  * @return 1 if everything went fine, -1 otherwise.
  */
int print_nameVersions(string CondName, string& TELL1Name, bool debug = false)
```

Listing 39: Prototype of `print_nameVersions` function, with its doxygen documentation.

```
/**
  * Extracts the GBE parameters from the XML database and populates the
  * corresponding fields in \c RecipeData_TELL1_ST.
  *
  * If debug is \e true, the tag name is shown, together with the value, the
  * mask and the index.
  *
  * @param [in] CondName name of the condition, i.e. "TELLBoard0".
  * @param [in] debug toggles degug print out.
  *
  * @return 1 if everything went fine, -1 otherwise.
  */
int configure_Ipv4Header(string CondName, bool debug = false)
```

Listing 40: Prototype of `configure_Ipv4Header` function, with its doxygen documentation.

```
/**
  * Extracts the TTC and process info settings from the XML database, and
  * writes the corresponding information in \c RecipeData_TELL1_ST.
  *
  * If \c debug is \e true, the tag name and the loop number is printed in the
  * log.
  *
  * @param [in] CondName name of the condition, i.e. "TELLBoard0".
  * @param [in] debug toggles degug print out.
  *
  * @return 1 if everything went fine, -1 otherwise.
  */
int configure_TTCSettings(string CondName, bool debug = false)
```

Listing 41: Prototype of `configure_TTCSettings` function, with its doxygen documentation.

```
/**
  * Extracts the ST specific info settings from the XML database, and
  * writes the corresponding information in \c RecipeData_TELL1_ST.
  *
  * If \c debug is \e true, the tag name is shown, as well as the size of the
  * read array, the values ant the mask.
  *
  * @param [in] CondName name of the condition, i.e. "TELLBoard0".
  * @param [in] debug toggles degug print out.
  *
  * @return 1 if everything went fine, -1 otherwise.
  */
int configure_STSpecific(string CondName, bool debug = false)
```

Listing 42: Prototype of `configure_STSpecific` function, with its doxygen documentation.

```
/**
  * Extracts the data RAM generator info settings from the XML database, and
  * writes the corresponding information in \c RecipeData_TELL1_ST.
  *
  * If \c debuf is \e true, the content of each RAM block is printed in the log.
  *
  * @param [in] CondName name of the condition, i.e. "TELLBoard0".
  * @param [in] debug toggles degug print out.
  *
  * @return 1 if everything went fine, -1 otherwise.
  */
int configure_DataGenRAM(string CondName, bool debug = false)
```

Listing 43: Prototype of `configure_DataGenRAM` function, with its doxygen documentation.

```
/**
  * Allows to disable all buttons when processing is performed by the panel.
  *
  * @param [in] state state of the buttons.
  */
void suspendButtons(bool state)
```

Listing 44: Prototype of `suspendButtons` function, with its doxygen documentation.

B.2.2 Global variables

```
string ControlUnit;                 // Name of the CU
dyn_string Children;                // Names of children
dyn_string ChildrenTypes;           // Types of children
dyn_string DeviceUnits;             // Names of DU
dyn_string DeviceDPs;               // Datapoint names
dyn_string DeviceUnitsEnabled;      // Enable DU names
```

Listing 45: List of variables related to interaction between CU and Tell1_configFromXML panels.

```
unsigned mappingSize;       // Total number of elements in DB (ie nbr of TELL1s)
mapping conditions;         // The xml database information are there
mapping RegisterMapping;    // The mapping between tags and registers is here
mapping NamesAndAlias;      // Must store the ccpc name for each board !
```

Listing 46: List of variables related to the XML database and mapping files.

```
string xmlInDataFilePath  = ""; // Path of the input database xml file
string xmlOutDataFilePath = ""; // Path of the output database xml file
```

Listing 47: Variables related to files management.

```
string HwType;                              // hardware type, should be "HwTypeCCPCTELL1_ST"
string hardware;                            // name of the TELL1 dumtell01 for instance
string detType;                             // name of the detector, ie IT or TT
dyn_dyn_anytype RecipeData_TELL1_ST;        // Will contain the recipe data
dyn_dyn_anytype RecipeMask_TELL1_ST;        // Will contains the masks
dyn_int Registers_Size;                     // Contains info about register size
dyn_string Types_ST;                        // Contains info about register types.
dyn_string RegisterNames;                   // Contains register names.
```

Listing 48: List of variables related to hardware type ans recipe information.

```
// Is \e true if list by TELL1 number is shown, \e false if list by source ID is used.
bool displayByNumber = true;
```

Listing 49: Variable used to manage display.