Last update: 8th Sept. 2006

Documentation for low-level Fed9USoftware (Fed9UVmeDevice and Fed9UVmeBase)

Matthew Pearson (Matthew.Pearson@cern.ch)
Gareth Rogers (Gareth.Rogers@cern.ch)

Contents

The Fed9U Namespace

The Front End Driver (FED) is a 9U 400mm VME64x card designed for reading out the CMS silicon tracker signals transmitted by the APV25 analogue pipeline ASICs. The signals are transmitted to each FED via 96 optical fibres at a total input rate corresponding to 3 Gbytes/s. The FED digitizes the signals and processes the data digitally by applying algorithms for pedestal and common mode noise subtraction. The FED home page can be found here [1].

The Fed9U software controls the FED registers and the data readout from the FED buffers. It is used to configure the run parameters, readout the events and provides format checking of events. It is also capable of monitoring the temperature of the hottest regions of the FED PCB and the voltages supplied to the FED components.

All Fed9U software is contained within the Fed9U namespace. It is structured into two layers, Fed9UUtils and Fed9UDevice, each with its own header and dynamically linked library. Fed9UUtils is a collection of utility classes that perform tasks that are used throughout the Fed9U namespace. Fed9UDevice contains the libraries and header file for accessing the FED hardware and is dependent upon Fed9UUtils. There is also a group of standalone executables that can be used to perform a variety of testing and debugging of a FED. They allow the FED to be installed as a stand alone piece of hardware, independent from any other component of the tracker readout chain.

Fed9UVmeDevice Class

Introduction

The aim of Fed9UVmeDevice is to provide a simple interface to the FED firmware. The firmware provides a set of methods that can be used to configure and control the FED. Fed9UVmeDevice accesses the FED through the class Fed9UVmeBase. This provides access to the FED registers that is similar in layout to the FED firmware. Fed9UVmeDevice provides an abstraction from this layout to an interface, which is not constrained by the FED firmware structure.

The Abstract Base Class (ABC) Fed9UABC, provided by the Fed9UUtils library, is the base class from which Fed9UVmeDevice is derived. It provides the interface, which is used by Fed9UVmeDevice for accessing the FED. The inherited interface gives the user access to the individual FED register at their lowest level of granularity. Fed9UVmeDevice extends on this interface to provide methods for the configuration and validation of the whole FED in a minimum amount of method calls.

Using Fed9UVmeDevice

A comprehensive list and description of all the methods provided by Fed9UVmeDevice class (as well as all the classes in the Fed9U namespace) are available on the FED savannah project site [2].

Instantiating a Fed9UVmeDevice Object

There is only one public Fed9UVmeDevice constructor provided; both the copy and assignment constructors are declared private and are not implemented. The constructor takes a single argument, a constant reference to a Fed9UDescription object, of which it makes a local copy. This Fed9UDescription class is also derived from Fed9UABC, hence is has the same interface and is used to store the FED settings in software. Fed9UVmeDevice uses the settings in the Fed9UDescription object for initialisation of the FED. In the event that a User updates a single register then Fed9UVmeDevice will updates it copy of the Fed9UDescription accordingly. A User must therefore ensure that their copy of the Fed9UDescription is kept up to date.

A Fed9UDescription object that contains only the default settings can be used to construct a valid Fed9UVmeDevice. There are two settings which must be correctly configured, the slot and crate number of the FED. This allows the Fed9UVmeDevice object to locate the appropriate FED. In a single crate environment the default crate number of zero is sufficient.

Upon construction of a Fed9UVmeDevice, it will interrogate a FED to ensure that the firmware versions on the FED are consistent with the global firmware version and the firmware versions are the same across the all FE and delay FPGAs. Optionally, by setting the hardware ID to zero or 998 in the Fed9UDescription object used to construct a Fed9UVmeDevice, the FED will be interrogated to determine both its EPROM and hardware version number; and also the firmware versions on each FPGA, again with consistency checked for on the FE and delay FPGAs. This process updates the Fed9UVmeDevice's local copy of Fed9UDescription, which must then be retrieved to retain these settings if the FED is deleted. If the hardware ID is set to zero then the FED construction is entirely passive, merely checking the settings and throwing an exception if an inconsistency is found that would affect the FED operation. If 998 is set to the hardware ID then the FED will verify the firmware versions against the global versions and if found to be different it will upload the appropriate firmware to the compact flash card on the FED, reloading the appropriate firmware and verifying the uploads success. This is recommended to ensure that all FEDs in the crate are running the global FED firmware. The existing firmware will be lost, however a back up can easily be made using existing FED methods.

Setting up a FED

If the User does not know the details of FED in a crate setting hardware ID of zero or 998 can be used to interrogate a FED, as described in section 0. A setting of 998 is recommended as this will ensure the FED is running the global firmware.

Once constructed, even with default Fed9UDescription settings, a Fed9UVmeDevice object can be used to readout events once its registers have been initialised. It is recommended that a User always provides a Fed9UDescription object that has been preconfigured for the run type they wish to use, rather than configuring a FED directly after default initialisation.

Fed9UVmeDevice provides a selection of initialisation methods, which of these are called depends on the operating needs of a User. Only init(bool) and fastInit() provide overlapping functionality. The methods are as follows:

- *init(bool):* Initialises the FED for running. Passing a Boolean true means the clock source set and a FED reset is performed before the registers are initialised. All but the strips settings, temperature control and voltage monitor are initialised. A reset of the front and back end counters is also performed. stop() is called during this method and start() must be called to allow data taking to begin.

- *fastInit():* Does not setup any of the FED registers, just resets the front and back end counters.

- *initStrips():* Initialises the pedestal, cluster threshold and disabled strip settings. This is required only in processed raw data or zero suppression modes.

- *initAuxChips():* Initialises the temperature control and voltage monitors chips. These are not required for data taking, but ensure the FED runs within safe physical operating parameters.

In the event that the FED is incorrectly configured it is possible to return the FED settings to their hardware default values by performing a FED reset. It will clear most of the FED registers, the exceptions are TTCrx chip, the temperature and voltage monitors, strip settings, and the clock source. The TTCrx, temperature monitors and voltage monitor all have their own resets. There are also FE and BE FPGA soft resets, which clear the FPGA logic, but do not affect any of the register settings. The clock source is unaffected by resets.

Fed9UVmeDevice also provides two methods for enabling and disabling data taking:

- start(): enables/disables the APVs and ADCs according to the description setting and enables the BE FPGA.

- stop(): disables all APVs and ADCs as well as the BE FPGA.

After init() and start() have been called with a Fed9UDescription object provided to the Fed9UVmeDevice upon construction data taking can begin.

Data Readout

The FED can either accept data from the optical links or fake events can be sent from within the FE units. Data from the optical links passes through the full FE unit before being passed onto the BE FPGA for readout via S-link or VME. Fake events originate in the FE FPGAs and can only test the FE FPGA event processing logic.

Once the data reaches the FE FPGAs they can respond by either detecting triggers or frames, known as scope frame finding mode respectively. When in scope mode no data processing is performed and both the APV tick mark and the frame are present in the event data. The amount of data recovered is determined by a FE unit register, which specifies the scope length to be recorded. When in frame finding mode only the frame, containing the strip data, is present in the event data. There are three different frame finding modes: virgin raw data mode, where no data processing is performed; processed raw data mode, only strip reordering and pedestal subtraction are performed; and zero suppression mode, where strip reordering, pedestal and common mode median subtraction, and cluster finding are performed.

Scope mode, virgin raw, processed raw and zero suppressed data modes make up the four basic data acquisition (DAQ) modes available on the FED. There is also a DAQ super mode, which is used to specify whether it is a normal or fake run. In normal runs data is taken from the optical inputs, fake mode the data is generated in the FE units. Further super modes of zero suppressed lite normal and fake are also possible to select. Zero suppressed lite is a data mode similar to zero suppression mode just with less data in the event. It can only be used in conjuction with the normal zero suppressed mode in the DAQ. There are normal and fake versions of this as for the normal and fake super DAQ modes.

Once the FED has been properly configured in the desired DAQ mode event readout can be performed. When in scope mode the FED will respond to triggers rather than frames and no optical input is required on the FE FPGAs. In the frame finding modes the FE FPGAs must be driven either by the optical inputs or fake events.

Events can be readout either over the VME link, which all FED commands are sent or via S-link a fast readout link to be used primarily.

When reading via VME, unread events will be stored in the QDR memory until it has been told the VME event buffer is empty. At this point the next event is sent from the QDR to the VME event buffer. An event can be checked for using the method hasEvent(), which has three different return values: no event present, event present, and last fragment of an event. A user will typically only be concerned whether or not an event is present, in which case the return can be treated as boolean, where a false represents no event and a true represents an event. If there is an event present the method getCompleteEvent should be called. This must be provided with a suitably large buffer to store the event. It gives the user the option of using a block transfer if their hardware supports it. Once read the data should be passed to the Fed9UEvent class, which can check the event format for errors. After getCompleteEvent has completed the FED will prepare the next event to be readout. The number of events that have been received and are left to be readout can be monitored using the method getBeEventCounterStatus.

System ACE, Compact Flash (CF) card and firmware updates

Firmware for the FE, BE and delay FPGAs are stored on a 60MB[1] compact flash (CF) card which can be accessed via a system ACE controller, giving the User some control over the firmware loaded onto a FED. The VME FPGA firmware is not stored on this card and must be loaded manually by a FED engineer.

The system ACE controller can handle up to eight different firmware revisions on the CF card. Each revision can contain a different set of FPGA firmware versions and can be loaded manually by the User if necessary, requiring just the revision number from the CF card they wish to load. There is no test the Fed9UVmeDevice can perform to ensure the load was successful, if there is at least one valid firmware revision on the CF card. The User must just ensure that the firmware versions they wished to load are present by checking the appropriate FPGA registers. If there are no valid revisions on the CF card then the reload will fail, however if there is already firmware loaded into the FPGAs then this will remain valid.

[1] 1MB = 2^{20} (1024x1024) bytes and not $1x10^{6}$(1000x1000) bytes as defined by the CF card makers. Hence there are only 60MB and not 64MB as stated on the CF card label.

The Fed9U software maintains a set a global firmware versions for each FPGA and there is a corresponding .ace file that contains the image of the CF card containing that firmware. The file names are constructed from the last three hex digits of the FE, delay, BE and VME FGPAs, in that order. Each revision number is stored in the filename in ascending order. Empty revisions are removed for trailing revisions, however zero padding is used for empty revisions between valid ones. For example:

- *33922337832b.ace:* FE version 0x339, delay 0x223, BE 0x378 and VME 0x32b in revision 0.
- *41a22337332b00000000000033922337832b.ace:* FE version 0x41a, delay 0x223, BE 373 and VME 32b in revision 0, FE version 0x339, delay 0x223, BE 0x378 and VME 0x32b in revision 2, with no firmware stored in revision 1.

Note that in both cases the VME FPGA firmware is the same, as this cannot be updated using the CF card. If it is different from the global version then it will not update the VME FPGA firmware and this must be done manually.

Fed9UVmeDevice provides the functionality to both upload and download the full 60MB image of the CF card and read/write it from/to a file following the above naming structure or User named file. The files themselves have a human readable header, which contains all of the firmware versions in each revision, the rest of the file is a bitwise copy of the CF card, plus an end of file marker. A member function is provided to read the file headers, hence allowing a User to determine the revision numbers that contain valid firmware versions. Firmware can be reloaded from any valid revision using the appropriate Fed9UVmeDevice member function. Member functions are also provided to read the status registers of the system ACE.

When uploading or downloading new firmware to/from the CF card it may be that the process is aborted during a transfer cycle. This can cause the system ACE to enter a state where its buffers are continually waiting to transfer the remaining data that is requested. In this case methods are provided to dump the contents of the transfer buffer and release the system ACE deadlock. In the case of the write cycle being interrupted this can only be released by uploading more data to the CF. As a result the member function will upload zeros and corrupt the CF card. It is recommended not to interrupt the set/get member functions once they have begun a transfer to/from the system ACE. There are safe guards to prevent this situation arising. The final state that Fed9UVmeDevice guards against is setting/getting outside of valid CF memory addresses. This will cause the system ACE to enter into an unrecoverable error state and the FED will have to be power cycled. Only for unrecoverable errors is such a drastic measure required.

Monitoring the FED

The FED contains two hardware-monitoring devices. These are the temperature monitors present on each FPGA and a voltage monitor. Each temperature sensor can monitor the FPGA temperature and sensor temperature. Each temperature can have an independent maximum temperature limit, which when exceeded will be flagged in a status register on the device. There is also a critical temperature limit, which, if exceeded, causes power to be cut to the FED to prevent damage to the components.

This is a last resort as the FED will have to be switched back on at the physical FED location, and cannot be performed via software.

The voltages monitored on the FED are 2.5V, 3.3V, 5V, 12V, its core voltage and the supply voltage. Each of these can have lower and upper limits set. If the voltage is outside a limit then it will be flagged in a status register on the device. It also monitors its temperature, with upper and lower limits that are flagged if it goes out of bounds as with the voltages. It is possible for the voltage monitor to measure an external temperature as well as the internal chip temperature, however in the current FED design this ability is not used.

Using the Eprom

The EPROM is a 2 kilobyte area of non-volatile memory that is split into four equal sized sections. It is possible to write protect one quarter, half or all of the EPROM memory. In the final system it is expected that the write protection level will be locked in the hardware giving the User access to only certain quadrants of memory. Currently, however, a User must set the write only level after each power up, otherwise the EPROM memory cannot be accessed. Due to the current nature of the write only level and the lack of any definition of what the EPROM memory should be used for the User should be careful that they do not overwrite any important data that someone else may have written; and also care should be taken not to write over the FED serial number, for which a specific set/get method exists in Fed9UVmeDevice. At present the only bounds checking that takes place is that the user is writing to a valid area of memory (i.e. inside the 2KB region) -- there is no software warning if a user attempts to write to a write protected area. If a write-protected area is written the write will appear to have succeeded, and there is no warning from either the hardware or software. The only mechanism that can be used to check is by reading back from that area of memory to ensure the written data is now present.

Fed9UVmeDeviceException

ICUtils::ICException provides a set of pre-processor macros that are used to create the framework for an exception class derived from ICUtils::ICException, which can then be customised to the meet the exception class requirements. It is not provided as part of the Fed9U namespace, but is provided as a general utility in the ICUtils namespace.

Fed9UVmeDeviceException is an exception class used to describe the various error conditions that can occur during the use of Fed9UVmeDevice. It is derived from ICUtils::ICException, which is derived from std::exception. It contains an exception code list, which can be used to identify the type of error thrown. It also contains an error string that identifies where the error was thrown from and gives some information on the error conditions that generated the exception.

Most member functions in Fed9UVmeDevice guarantee that Fed9UVmeDeviceException is the only exception that can be thrown. In this case all exceptions thrown during this member function call are caught and rethrown as a Fed9UVmeDevice exception, appending an error string detailing where it was rethrown from and any further information that may be of help indiagnosing the problems.

Typically Fed9UVmeDevice member functions throw errors for out of bounds conditions on input variables or consistency checks across FED registers that are required to all have the same value. There are a few exceptions to this, but for the most part these are the errors a User must be prepared to handle. Out of bounds errors are fixed by simply recalling the appropriate member function with a value that is within the desired range. Inconsistencies in settings can either by calling the appropriate member function to reset the register values or in the case of firmware version inconsistencies either reload the appropriate firmware revision or uploading a new firmware file to the compact flash card.

For errors that are caught and rethrown during a member function call the User must consult the relevant class documentation that should be included in the exception message. Often Fed9UVmeDevice does not know which method has thrown the error so merely appends information about the member function that caught the error. Where possible Fed9UVmeDevice will try to deal with known errors thrown during a member function call, but in most cases this is not possible and the error is simply rethrown as a Fed9UVmeDeviceException. One error that a User may encounter contains within the error string:

- "cannot write to address X with 4 bytes of data"

This typically means that the Fed9UVmeDevice object has been constructed for a slot that does not contain a FED. In this case the Fed9UVmeDevice should be remade with the appropriate slot number.

The Fed9UVmeBase Package

Introduction

The Fed9UVmeBase class contains methods which enable FED configuration and data readout. It sits below Fed9UVmeDevice and acts as an interface between the user friendly methods in Fed9UVmeDevice and the FED firmware commands. It contains methods to mirror the firmware in the FED FPGA chips. Therefore, for every firmware command there is a corresponding method in the class. This class is a low level part of the Fed9USoftware and should not be used directly by a user.

For a detailed documentation on the FED commands see the FED FPGA documents (http://www.te.rl.ac.uk/esdg/cms-fed/qa_web/documentation/). These documents contain a description of each command and details such as the command format, timing issues, etc.

Low level utility classes

The Fed9UVmeBase class uses two additional utility classes to help construct the command and send it to the FED using the Hardware Access Library (HAL). They are called **Fed9UConstructCommand** and **Fed9UHalInterface**. Fed9UConstructCommand is used by every method in Fed9UVmeBase to construct the bit pattern that is sent to the FED. It uses command specific information and user

defined information (for example, the data to write to the FED). Fed9UHalInterface is used by every method in Fed9UVmeBase to provide an interface to HAL to allow writes/reads to/from the FED VME space. It requires a HAL VME address table as input to its constructor, in order to initialize the VME interface in HAL.

Fed9UVmeBase

The methods in Fed9UVmeBase address three types of FPGA on the FED: the 8 front end FPGAs, the back end FPGA and the VME interface FPGA. Every front end FPGA method requires an input telling it which FPGA to address (1-8 are the front end FPGAs, and 15 is used to address all front end FPGAs). These numbers mirror the FED firmware FPGA numbering scheme. Internally other FPGA numbers are used to address the back end FPGA (10 and 16), and the software trigger (9). To address the VME FPGA the software uses the VME memory map defined in the HAL address table.

Constructor

The location of the HAL address table text file must be specified as a parameter to the Fed9UVmeBase constructor, along with the FED VME base address (in base 16) and crate number (the crate number in effect selects the PCI-VME interface, of which there are one per crate).

```
Fed9UVmeBase::Fed9UVmeBase (unsigned long baseAddress,
                            string theAddressTable,
                            unsigned short crateNumber)
```

One could use the Fed9UVmeBase class independently of the rest of the Fed9USoftware, in order to facilitate a low level FED test, by the simple constructor call above and a set of subsequent method calls (described below). However, using it via Fed9UVmeDevice is advised and is much more user friendly.

Read/Write commands

The read/write methods take as input a read/write flag and a vector holding the data to write. Every method is also passed a reference which can be used to hold the data when reading from the FED. Each method can throw an exception in the form of a Fed9UVmeBaseException object. For front end FPGA commands the FPGA number (1-8 or 15) is a required input. An example of a read/write front end FPGA method is the “load threshold” command (which loads the threshold for frame finding in the front end):

```
void Fed9UVmeBase::feCommandLoadThresh (
                                   unsigned short selectedFpga,
                                   bool read,
                                   const vector<unsigned long> &arguments,
                                   vector<unsigned long> &readArguments)
                        throw (Fed9UVmeBaseException)
```

In the Fed9UVmeBase source code this method will be accompanied by a comment explaining what the command does, and what each input parameter means. There is a common naming convention throughout Fed9UVmeBase, so that *selectedFpga*

always means the FPGA number to address (1-8 or 15), *read* is always the read/write flag, the *arguments* vector always holds the data to write (this can be a dummy zero length vector when reading) and the *readArguments* vector always holds the data read from the FED when reading (this can be a dummy zero length vector when writing).

Read only commands

The read only commands have a simpler interface. For example, the method which reads the front end FPGA firmware IDs is:

```
void Fed9UVmeBase::feCommandFirmwareIdReg(unsigned short selectedFpga,
                                          unsigned long &readArguments)
                                throw (Fed9UVmeBaseException)
```

and the method to read the back end FPGA status register is simply:

```
void Fed9UVmeBase::beCommandBeStatusRegister(unsigned long&readArguments)
                                throw (Fed9UVmeBaseException)
```

Additional commands and private methods

In addition to the basic format outlined above there are a few commands that have a slightly more complicated interface. One example is the control command for the LM82 temperature sensor on the back end FPGA. As well as the standard read/write interface presented above there are additional I^2C parameters that return any I^2C error or busy flags. There are comments in the source code as and when these situations arise.

There are also numerous private methods that factor out common code from many of the public methods. These perform tasks such as error checking, data formatting/unformatting, using the Fed9UHalInterface and Fed9UConstructCommand classes and a few FED commands factored out of the public commands.

Summary

Fed9UVmeDevice is a derived class of Fed9UABC and provides access to the FED registers. It has been designed to allow the user a simple and easy to use interface that is flexible enough to meet all users needs. It relies on a variety of support classes that range from exception handling to classes that correspond directly to devices on the FED. It has been integrated into the Fed9USoftware and serves as a base class for Fed9UDevice, which provides additional higher level methods.

Operating below Fed9UVmeDevice is Fed9UVmeBase, which provides an interface for Fed9UVmeDevice to the actual FED commands in firmware. Fed9UVmeDevice acts as a user friendly interface to Fed9UVmeBase.

References

[1]FED UK - http://www.te.rl.ac.uk/esdg/cms-fed/qa_web/

[2]Fed9U savannah project page - http://savannah.cern.ch/projects/fed9usoftware/