Python Tool for EEPROM Info Generation: fmc_eeprom_gen.py

CERN : BE-CO-HT
Ross Millar
02/12/2011

Introduction:

As explained in "EEPROM Guidelines for FMC"[1], there are certain requirements for the EEPROM device contained on FPGA Mezzanine Cards (FMCs). In order to aid the generation of this data and to ensure it is properly formatted, a python tool has been developed. This tool has been designed to take user inputs from a template file, and generate a list of addresses and associated binary data, which can be dumped to the EEPROM on the FMC. How this binary file is written to the EEPROM is dependent on the carrier board which hosts the FMC, and the EEPROM device on the mezzanine.

This document aims to describe the format of the input file, and to detail how this file can be modified by the user in order to produce an output file which is line with the standards.

Furthermore, a brief description shall be given explaining how to operate the **fmc_eeprom_gen**[2] tool and the inbuilt help functions. Details shall be given of how to add new functions to the python tool in order to easily add possibilities for new encoding types.

Finally, it shall be explained how to incorporate the tool into the Production Test Suite environment. This will make use of a python class which is available on the open hardware repository[3]. This takes an input file in the format produced by the tool, and dumps the contents to the EEPROM. This class is specific to the 24AA64, I2C EEPROM, and works in conjunction with other python classes which are used to drive the wishbone master internal to the FPGA on the Simple PCI Express Carrier (SPEC) board.

---

[1] http://www.ohwr.org/projects/pts/repository/revisions/master/show/test/fmceeprom/doc

[2] http://www.ohwr.org/projects/pts/repository/revisions/master/changes/test/fmceeprom/python/fmc_eeprom_gen.py

[3] http://www.ohwr.org/projects/pts/repository/revisions/master/entry/test/fmcadc200k16b11cha/python/eeprom_24aa64.py

Contents:

Description of the XML input file:

The input file is in the XML format as it easily allows input data elements to have associated attributes. An example has been provided with this document, which provides the Areas and Fields required by the FMC standards. This template should require minimal alteration by the user however when alterations are required there are certain formats which have to be adhered to. The attributes which may require alteration are discussed in this document.

The python file which processes this input file makes use of the libxml2 library in order to parse the XML. This particular library was chosen due to the fact that it is a standard library on Ubuntu and several other Linux OS's.

"FMC_EEPROM data"

As can be seen in the template file,"FMC_EEPROMDATA" is the highest element. It contains one attribute, which is simply the size of the EEPROM memory (in bytes) which the user intends to write to. This is used to ensure that the formatted data input to the tool does not exceed this value. If this does occur, an error will be produced, and the encoded binary file will not be written to. The user will be informed of the size of the eeprom and the size of the converted data, in order to see how much data needs to be cut.

"Area"

As explained in the "EEPROM Guidelines for FMC", FMC standards[4] dictate that certain areas must be present in the EEPROM. There are five potential areas which can be used, however not all require use. Each area type has an associated number, and it is this number which must be included in the "number" attribute, contained in the Area element. This number should not relate to the number of Areas present in the template file! The inclusion of this area number allows the common header to be generated correctly, as an omitted area should produce an offset of 0.

Area Name	Area Number
Internal Use	1
Chassis Info	2
Board	3
Product Info	4
Multi Record	5

[4] http://www.ohwr.org/projects/fmc-projects/wiki/FMC_standard

As can be seen in “EEPROM Guidelines for FMC”, certain areas contain a field indicating the length of that area, in multiples of 8 bytes. If such a field is required, the “**pos_of_len_field**” attribute should be filled in with the desired position of the length field. The “**pos_of_len_field**” does not take in to account the 0th offset. For example, a **pos_of_len_field** of 1 will generate a length field which is the 1st Field in that area. If a length field is not required for an area, the attribute should be set to ‘0’.

Within standard area elements, the child elements in the XML file are the fields themselves. The names and order of these fields should be left as given in the template file, in order to correspond correctly to the FMC standard. There are only three attributes of a field which should require to be altered by the user.

Has_typelen:

This attribute relates to the Type length byte which was described in the “EEPROM Guidelines for FMC”. Certain fields require this type length byte to be generated, therefore in order to comply with the FMC standard; this field should remain unchanged for the areas given in the template. However, if an area is added which includes custom fields, it is required that these fields have an associated type/length byte. In these instances, the **“has_typelen”** attribute should be written as “yes”. For the addition of further areas, the user should refer to “Platform Management FRU Information Storage Definition v1.0”[5] in order to find the fields required by that area, and if type length bytes are required for each field.

Content: This attribute is where the user enters the data which is to be formatted, and written to the EEPROM device.

Type :

This attribute indicates the type of encoding to be used when encoding the content value. As shall be explained below the type of encoding requested has an impact on how the “content” attribute should be written.

Currently supported by the tool are the following encoding types:

ASCII:

This shall take the users input and generate the ascii code for each character. **check if this can include all characters**

[5] http://www.ohwr.org/projects/fmc-projects/wiki/FMC_standard

Signed short:

All signed shorts are required to be given in 0.01 increments. If this encoding type is selected the content input must be in the form +0.00 or -0.00. Any deviation from this format shall produce an error, and the tool shall return a statement to the user indicating the position of the error in the template file, and what input format was expected.

Binary:

These field types require a binary input. At minimum this input should be of byte size. For instance “00000000” opposed to “0”. If more than one byte is required the input format is the first byte, followed by white space, followed by the next. For example “00001111 00001111”. If the input is not given in this format an error shall be produced and the tool shall return a statement to the user indicating the position of the error in the template file, and what input format was expected.

BCD:

This type requires that content is given an integer number, e.g. “123”. If the content contains any white space between characters an error shall be produced. Similarly if the content contains any characters other than numbers an error shall be produce. The fault which caused the error shall be indicated and printed to the terminal.

MultiRecord Areas:

The 5th Record Type Contains the MultiRecord Area. This area type is explained in “EEPROM Guidelines for FMC”.

As can be seen in the template file, there are several of these in the Area 4 (CHECK). The “name” attribute associated with each MultiRecord relates to a number of possible MultiRecord types, outlined by the standard and detailed in the *other document*.

The MultiRecord name must be one of the following:

- DC_LOAD
- DC_OUTPUT
- OEM_OUTPUT
- management_access_record
- base_compatibility_record
- extended_compatibility_record

The tool uses the MultiRecord ‘name’ input to generate a MultiRecord type identifier. This identifier is included in the multirecord header corresponding to that record. All such headers are produced automatically.

MultiRecord types other than the first three listed above have not been given in the template XML file, as they are not required by the FMC standard. If the format for these MultiRecord types is required, the format can be found in “Platform Management FRU Information Storage Definition v1.0”.

If an error is found in this attribute the tool shall return an error to the user, indicating that the record name was not recognized. The name which caused the error will be given to the user, as will a list of the correct choices.

In certain circumstances a field is split in to several sub-byte length binary values. In these cases, ByteFields have been used. ByteFields contain several fields which should always add up to one byte. If this is not the case, the tool shall return an error indicating that the Byte Field is not of byte size, giving the position of the fault in terms of the byte field name.

Field Descriptions:

The description attribute has been included in the XML file purely to give the user an indication of what is expected at that field. If the description is not sufficient, please refer to “EEPROM Guidelines for FMC” and “Platform Management FRU Information Storage Definition v1.0”_

Last_Multirecord

In order to correctly identify that a multi record is the last in that area, it is required that user sets the “last_multirecord” attribute to “yes”.

Using the fmc_eeprom_gen.py:

The python file “fmc_eeprom_gen.py” and the XML input file “eeprom_input.xml” can be found on the open hardware repository[6].

Once downloaded, open the XML file and edit in the way explained by this document.

Several help functions have been created in the fmc_eeprom_gen.py file in order to provide the user information regarding how to edit the XML file.

In order to view all the help options on a Linux machine, use the terminal to move to the directory containing the “fmc_eeprom_gen.py” and the XML file. Enter the following:

“python fmc_eeprom_gen.py --help”

This type of input (-- help) is the long option. As can be seen from the screenshot below, the user has the option of the long option , “ - - help”, or the short option “ – h”.

[6] http://www.ohwr.org/projects/pts/repository/revisions/master/show/test/fmceeprom/python

```
File  Edit  View  Search  Terminal  Help
user@baraka:~/mcattin/fmc_adc_100Ms_test/ross/eeprom tool$ python fmc_eeprom_gen.py -h
Usage:

usage:  python fmc_eeprom_gen.py [options] calibration_data_filename.txt

 Help Options for fmc_eeprom.py

Options:
  -h, --help            show this help message and exit
  -r, --run_script      Run the script and generate formatted output file
  -i FILE_IN, --file_in=FILE_IN
                        Filename for output file
                        [default:eeprom_formatted_data.txt]
  -o FILE_OUT, --file_out=FILE_OUT
                        Filename for output file [default:eeprom_data.xml]
  -t, --types           Prints the valid types for fields in eeprom_data.xml
  -m, --multirecords    Prints the valid options for MultiRecord types
  -a, --areas           Prints the available Areas and their associated
                        numbers
  -c, --calib           Prints the required format for calibration data
  -d, --debug           Add this option with -r to produce labels on output
                        data
user@baraka:~/mcattin/fmc_adc_100Ms_test/ross/eeprom tool$
```

Help Options:

As can be seen in the above screen shot, the user can request certain information regarding the options for types, multirecords and areas.

For example, for details on MultiRecord options, the user would enter:

“python fmc_eeprom_gen.py –m” or
“python fmc_eeprom_gen.py -- multirecords”

On receiving this command, a list of the possible multirecord types will be printed to the screen. Similar functions are available for “types” and “areas”. This has been included to give the user information regarding the valid options available to be input to the XML file.

Once the XML file has been finalized and edited in the manner described by this document, the eeprom tool can be run by the following command:

“**python fmc_eeprom_gen.py –r” or**
“python fmc_eeprom_gen.py - -run_script”

By default this will read from the XML file named **“eeprom_input.xml”** and write to **“eeprom_formatted_data.txt”**.

In order to give different input and output files the tool can be run using the following command. If one of these commands is omitted, the default file shall be used.

“python fmc_eeprom_gen.py -o filename_out.txt -i filename_in.xml -r”

Finally, by adding “- d”, when running the tool, the output file shall contain a label for each byte of data. This label shall indicate the encoding type of the data, and shall label the common header and MultiRecord headers.

This allows an easy debugging mode if someone wishes to add a function or modify the function in any manner.

Adding a type conversion function:

The “Platform Management FRU Information Storage Definition v1.0” indicates that an encoding type of ‘unspecified’ can be used, as well as the encoding types mentioned previously.

In order to add a function to allow a further conversion type the following must be done:

The function should be written in such a way that it takes a string, and returns a list of bytes. Each element of the returned list must a binary string of 8 bits long. For example: [‘00000000’,’00000001’,’00000010’].

Even if only one byte is returned it should be the single element of a list. Examples can be seen in existing functions such as “convert_to_ascii”.

Below is an **example function** which returns a value based on a yes or no input:

```
def yes_no_function(string):
        if string.lower() =="yes":
                return ["00000001"]
        elif string.lower()=="no":
                return ["00000000"]
        else:
                print "Content not as expected for type 'yes_no'"
                print "Expected 'yes' or 'no'"
                return "error"
```

Figure 1. Example of new type function.

As can be seen above, if the received string isn't as expected, the function should print a message indicating the fault, and return the string "error". This will allow the tool to generate an error message and to print the location of the error in the XML file.

The next stage is to add the name of the new type to "type_list", which is defined near the top of the file. This will allow new types to be recognized when generating a type_len byte for this type of encoding. All custom encoding types added by the user will generate a type code of "00", which is the same code generated by binary inputs. This corresponds to the type length byte for type "unspecified", as outlined in the "Platform Management FRU Information Storage Definition v1.0" standard.

```
#####################################
# Define New Type
#####################################

type_list =["yes_no"]
type_list_description = [" Takes an input of the form 'yes' or 'no', returns an associated number"]
```

Figure2. Add new type to 'type_list' and add description.

By adding the new type to this list, the user also makes this type printable by the "-t" or "- -type" help function (figure 5). This will allow new users to see the encoding types which are available in updated versions of the tool. In order to provide the user with information regarding the input requirements for the new type, a description should be added to the type description list. This should be at the same position in the list as the new type.

Once the function has been completed, and the new type has been added to the type list, the final stage is to map the new type to the function. This should be done at the “typeConvFuntionsMap”. The new function is shown to be added on the bottom line.

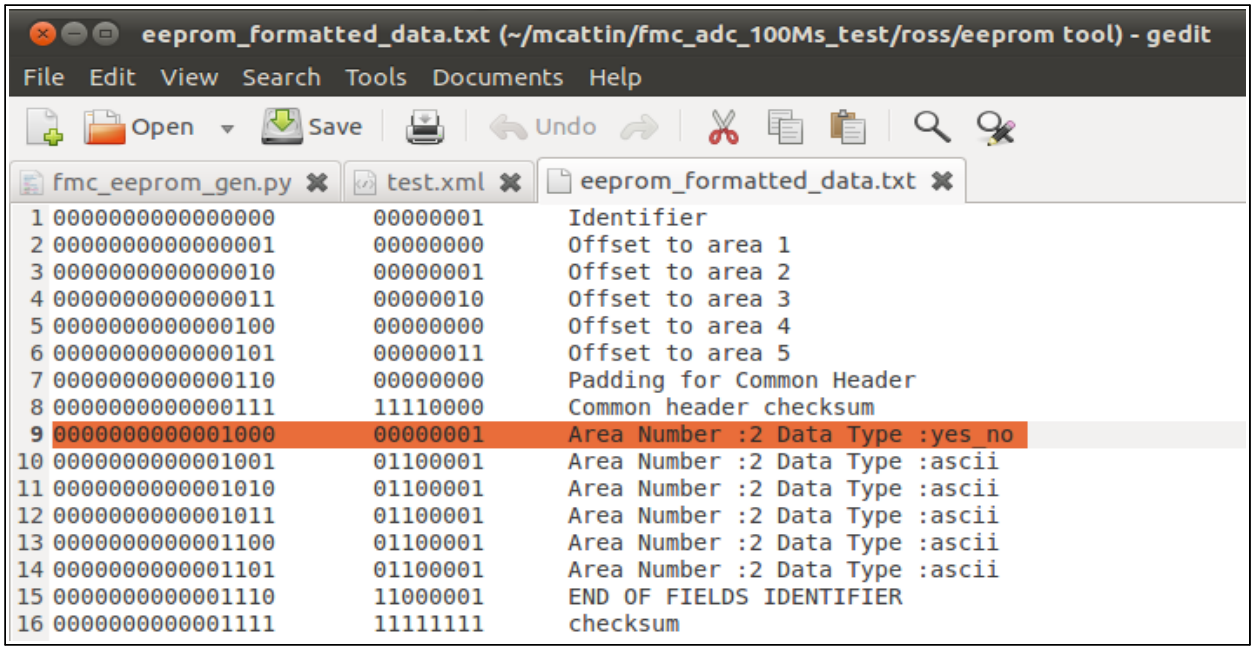
```
########################################
# Funtions map
########################################
typeConvFunctionsMap = {
    'ascii'             : convert_to_ascii,
    'bcd'               : convert_to_bcd,
    'signed_short'      : convert_to_signed_short,
    'binary'            : check_byte_size,
    "yes_no"            : yes_no_function
}
```

Figure3. Add to function map.

Once modified, it is suggested that the user checks the conversion of the new function in the following manner. A field of choice in the XML file should have its type element changed to the new type. The content should be changed appropriately.

The tool should then be run with debug mode and the output file should be analyzed to see if the conversion and the data is as expected. A fully example of adding a new function is shown below.

Figure 4. Run in debug mode.



As can be seen in figure 5, the type and the description are now added to the –t (or - - type) help function, which can be run as explained previously.

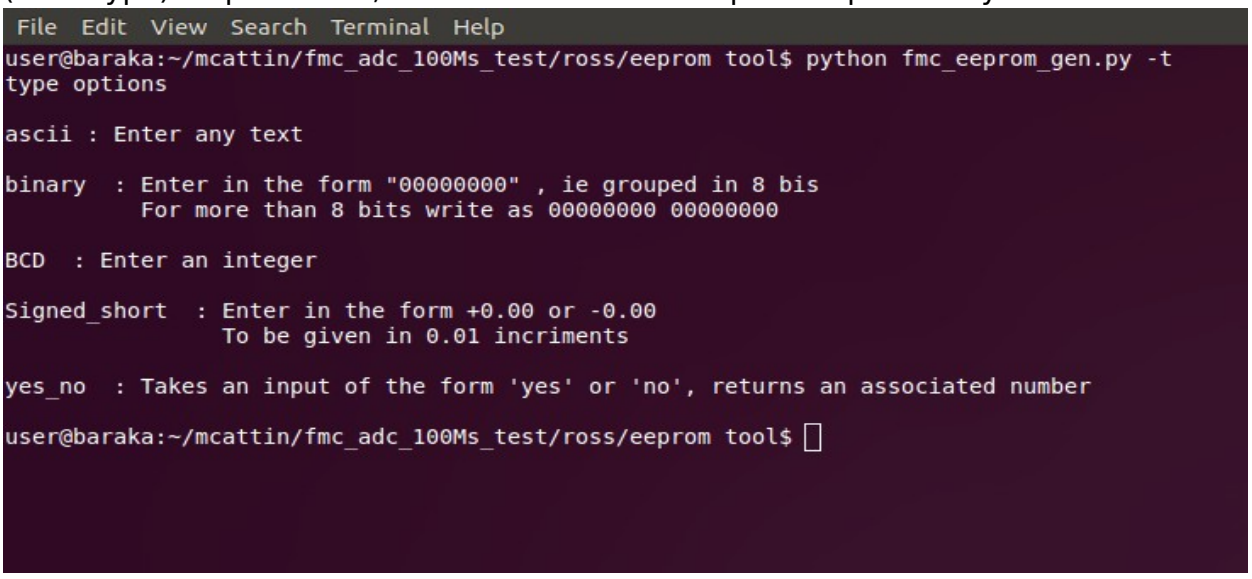


Figure 5. Help option displaying available content types

Using with calibration data:

As described in “EEPROM Guidelines for FMC” there is the possibility of using an area to store custom manufacturer data. In certain circumstances this area shall be useful for storing calibration data for an FMC. The fmc_eeprom_gen.py tool has an

option to support the conversion and formatting of such data, in order that calibration can be added in an automated manner.

The format for this data has been decided within the Hardware and Timing section for a couple of specific FMC cards, however this option has been written to be easily configurable to support other formats.

Following the board manufacturing process automated tests shall be run to obtain certain calibration parameters. As it would be impractical to include this data in the XML format, the tool reads this information from another source. Calibration data should therefore be written in to a text file in the following format:

Hex_Number1, Type_len_byte_required(yes or no)
Hex_Number2, Type_len_byte_required(yes or no)
...
Hex_NumberN, Type_len_byte(yes or no)

Example:
0x4323,yes
0x1,no
0x0000,yes

The given hex number can be of any length. The function to process such an input generates however many bytes are required to represent each field. Whether or not the field has an associated type/length byte can be selected by entering yes or no after a comma.

Once this calibration file has been generated, the fmc_eeprom_tool should be run. In order to process this file, the file name should be given as an argument when running the tool.

For example, a script can contain:

python fmc_gen_tool.py -r calibration_data.txt

This will process the default XML file associated with the FMC as well as the calibration data, in order to produce the formatted output.

Calibration data given in this form shall be included in Area 1, which is the Internal Use area. It is therefore important that **Area 1 is not given as an input in the XML file** if it is intended for further data to be provided as calibration data.

Example of using tool in Production Test Suite environment:

In order to make these tests fully automated, the fmc_eeprom_gen.py tool can be incorporated into the Production Test Suite (PTS) for FMC cards. In order to do this,

the following files shall have to be contained in the same directory as the python files used in the PTS:

- fmc_eeprom_gen.py
- eeprom_input.xml
- calibration_data.txt

It is assumed that the calibration_data.txt is generated during a test which is run before the test which runs fmc_eeprom_gen.py. If any other filename is used as the XML input file, the name shall have to be added as an option when running the tool (in line 38 ***figure 6***). This was described a previous section. By this stage the XML file should be fully edited by the user to include all the information for the FMC. It should have been run in order to check no errors are found.

The tool should also be tested with a calibration_data.txt file containing the same amount as data which is expected to be produced from the actual calibration process. This way the user can ensure that no errors shall be produced due to lack of space in the memory during the automated process.

The output file generated by fmc_eeprom_gen.py is produced, and then automatically dumped to eeprom. For FMC designs in the Hardware and Timing section which use the SPEC board, PTS tests shall use various python classes to control the wishbone slaves in the FPGA. The python class for the 24AA64, 64k I2C EEPROM contains a function to dump a file to the EEPROM. If using this device, the python class can be found at www.ohwr.org.

```
24 def main (default_directory='.'):
25
26     path_fpga_loader = '../firmwares/fpga_loader';
27     path_firmware = '../firmwares/TestSuite.bin';
28     firmware_loader = os.path.join(default_directory, path_fpga_loader)
29     bitstream = os.path.join(default_directory, path_firmware)
30     os.system( firmware_loader + ' ' + bitstream)
31     time.sleep(2);
32
33     # Objects declaration
34     spec = rr.Gennum()
35     fmc = fmc_adc_test_suite.CFmcAdc100ks(spec)
36
37     # Runs fmc_eeprom_gen.py - giving file containg calibration data as an argument
38     os.system("python fmc_eeprom_gen.py -r calibration_data.txt")
39
40     # Uses eeprom class to dump the output file from fmc_eeprom_gen, into eeprom
41     fmc.fmc_eeprom_gen_wr("eeprom_formatted_data.txt")
42
```

Figure 6. Example of using the fmc_eeprom_gen tool in a PTS test