The CreditCard PC python module ’ccpc’

Public Note

Issue:	1
Revision:	0
Reference:	LHCb-XXX-2005
Created:	September 5, 2005
Last modified:	December 15, 2005

Prepared by: Stefan Rosegger TU Graz and Niko Neufeld CERN-PH

Abstract

This document describes the Pythonmodule ’ccpc’. The member-functions are derivatives of the functions included in the standard libraries ’lblib’, ’i2culib’ and ’JTAGlib’ which are mainly used for the LHCb CCPC glue-card. This document gives detailed instructions how they were build and how to use them.

Document Status Sheet

1. Document Title: The CreditCard PC python module ’ccpc’			
2. Document Reference Number: LHCb-XXX-2005			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Release	0	December 15, 2005	Public release
Draft	0	September 5, 2005	First version

Contents

1 Introduction

Python is an interpreted, interactive, object-oriented programming language. It is used intense in the ONLINE subsystem of the LHCb Dedector in CERN. It combines remarkable power with very clear syntax. Python has modules, classes and exceptions. New built-in modules are easily written in C or C++.

To make the CreditCard PC libraries available in form of a pythonmodule improves the handling of the CCPC libraries[1]. Rapid prototyping and direct control of possible errors via Pythonexceptons are just a few advantages of using an interpreted language like python.

A huge amount of programs and libraries can be found to build a module as for example BOOST[2] and LCG-Dict. Those programs can be used as tools to include large C and C++ libraries in python. They are providing the conversion of objects of the chosen language to equivalent python objects which can be used and manipulated directly in python interpreter.

These programs are very powerful and strongly supported but of course they can not guess what to do with for example a void* object. The conversion of those has to be done by hand. Because of the fact that the CCPC libraries are using these kind of objects very often the benefits of Boost and related programs shrink in this case.

Making the link between python and the CCPC libraries, a so called pythonwrapper, by hand would increase the legibleness and the easy change of the sourcecode of the pythonmodule and decrease the filesize.

The following sections describe this module. The latest version is per default installed on all CCPCs runing Linux SLC4 or higher.

2 Building a python module

To build an extension module in python is described in detail in "Extending and Embedding " in [1]. Only a small introduction is given in the following sections of this document.

Examples from the ccpc module building files are shown to give a short overview how they are created and how they can be extended to increase the capabilities of the ccpc module.

2.1 InitProzess and method table

The initialization process for a python module has to be done by a function similar to the following. It has to be included in the c-code of the module and must be named initname(), where name is the name of the module:

```
void initccpc(void) {
  (void) Py_InitModule(''ccpc'', ccpc_methods);
}
```

The initializationprocess needs the definition of the Memberfunctions in form of a "method table" where the memberfunctions are defined as for example:

```
static PyMethodDef ccpc_methods[] = {
      ...
      { ''lbread'',  (PyCFunction)ccpc_lbread, METH_VARARGS|METH_KEYWORDS,
        ''Description shown in help''},
      ...
      {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

[1]More informations about those libraries can be found in [2]
[2]More informations can be found in [3]

2.2 Argumentparsing

Because python can only handle pythonobjects, the memberfunctions have to understand what arguments they get and they have to return pythonobjects. This can easily be done by keywordparameters and building pythonobjects. Detailed information about building pythonobjects can be found in "Python/C API Reference Manual" in [1].

```
static PyObject *
ccpc_lbread(PyObject *self, PyObject *args, PyObject *keywds) {
  int width, addr, n;
  char *block = ''s'';
  static char *kwlist[] = {''width'', ''addr'', ''n'', ''block'', NULL};
  if (!PyArg_ParseTupleAndKeywords(args, keywds, ''iii|s'', kwlist,
                                   &width, &addr, &n, &block))
        return NULL;
  ...
  return Py_BuildValue(''i'', numOut);
}
```

2.3 Exceptions

If an error occures then a significant string, which describes the error, can be set to an standard built-in exception like

```
if (glue_default_init()) {
  PyErr_SetString(PyExc_IOError, ''glue_default_init() failed\n'');
  goto error_out;
}
```

2.4 Setup file

To build an extension module distuils can be used. The distuils package contains a driver script, `setup.py`. This is a plain Python file which can look in a simple case like this:

```
from distutils.core import setup, Extension
module1 = Extension('ccpc',
                    sources = ['ccpcmodule.c'])
setup (name = 'ccpc',
      version = '1.0',
      description = 'Module for CCPC's',
      ext_modules = [module1])
```

With this `setup.py`, and a file `ccpcmodule.c`, running

```
 python setup.py build
```

will compile the file, and produce an extension module named "ccpc" in the build directory. Depending on the system, the module file will end up in a subdirectory build/lib.system, and may have a name like `demo.so` or `demo.pyd`.

More information about building, building and rpm-package and installing modules can be found in "Distributing Python Modules " in [1].

3 Usage of the module 'ccpc'

An introduction about how to use Python is given in "Tuotorial" in [1]. This section should give more details about the specific usage of the pythonmodule 'ccpc', the included functions and the possible features.

3.1 Import

The ccpc module can easily be imported with standardprocedures of the python interpreter like ...

```
>>> import ccpc
```

```
>>> from ccpc import *
```

In both procedures the whole module will be avaiable. The secound statement loads the memberfunctions directly into the namespace. The first one loads them into the scope of the module. The following examples shows the usage of the memberfunctions in both cases ...

```
>>> ccpc.lbread()
```

```
>>> lbread()
```

The built-in functions can be listed with `dir(ccpc)` (just when the first import was used). How to use these functions is shown in this document and also by using the syntax `help(ccpc)` directly on the python commandline (or if the module was imported with the secound importstatement with `help(functionname)`).

The usage is very similar to the usage of the functions via C-code. More details about the C-functions can be found in [2].

3.2 Embedded lblib

The local bus library is the basic library to program the CCPC and glue-card. It interfaces directly to the device driver plx. It offers functions to program the PLX registers to read and write local bus addresses and to remap local memory regions into user space.

Two of those functions are included in the ccpc module, `lbread` and `lbwrite`. Those functions make it possible to read and write from and to the local bus. They can be used as follow ...

```
values = lbread(bitwidth, address, n, readmode)
      ... bitwidth must be either 8, 16 or 32
      ... address to read from (integer)
      ... n is number of blocks to read (integer)
      ... readmode ... optional: 'b' or 's' (string)
```

The values which were read are written into an integertuple. They can be handled in python in the usual way.

Parsed Integers can be written in any form for example in decimal `65551` or in the equvalent hexadecimal `0x1000f`. The readmode can be either blockreading `'b'` or singlereading `'s'`. This is an optional argument (default is singlereading). It must be parsed as a string indicated by `'...'`.

The same conventions are used in `lbwrite`.

```
lbwrite(bitwidth, address, (value, ...), writemode)
      ... bitwidth must be either 8, 16 or 32
      ... address to write on (integer)
      ... values to write (integertuple)
      ... writemode ... optional: 'b' or 's' (string)
```

The values to write have to be parsed in form of an integertuple. In python a tuple is characterized through round brackets so even if there is just one value to write, it has to be parsed as a tuple for example `(0xff, )`. The maximum size of each value is determined by the parsed bitwidth.

3.3 Embedded i2culib

The I2C controller on the glue-card can drive 4 different I2C channels. The I2C library makes it possible to write, read and scan. It also allows combined write and reads (separated by a restart condition on the bus). The following functions are included in the ccpc module.

```
values = i2cwread(bus, addr, subaddr, n)
      ... bus (integer)
      ... address (integer)
      ... subaddress (integer)
      ... n is number of values to read
```

This is the function to read from the i2c bus. The returned values are parsed as an integertuple.

```
i2cwrite(bus, addr, (subaddr, [values] ...) )
      ... bus (integer)
      ... address (integer)
      ... subaddress and values to write (integertuple)
```

This is the function to write to the i2c bus. The subaddress and the values to write have to be combined in an integertuple. Maybe this will be changed in a later version, so that the subaddress can be parsed in form of an integer detached from the valuetuple.

```
value = i2cscan(bus, addr, [scanmode])
      ... bus (integer)
      ... address (integer)
      ... scanmode ... optional: 'w' or 'r' (string)
```

This function is used to scan an address of the i2c bus in read or write mode. It does not actually read or write any data on the bus. It just checks whether it is acknowledged upon a read or write request for a specific addressFunction. Return value is `1` (True) on success and `0` (False) otherwise.

The scanmode can be either writemode `'w'` or readmode `'r'`. This is, like in the lblib library an optional argument (default is writemode). It must be parsed in form of a string.

3.4 Embedded JTAGlib

The JTAGlib is a C library which make possible to send data to the JTAG Controller LVT8980. This Library exposes all the controller function, and other functions necessary to control the controller registers (for a detailed description see [2] and linked).

The following functions of the JTAG library are included in the ccpc module.

```
bitsIn = drscan(chain, nbits, bitsOut)
      ... chain is the JTAG chain (integer: 1, 2 or 3)
      ... nbits is number of bits to shift out (integer)
      ... bitsOut is bits to shift out (integer or string)
      ... bitIn is received bits which were read (integer  or string)

bitsIn = irscan(chain, nbits, bitsOut)
      ... chain is the JTAG chain (integer: 1, 2 or 3)
      ... nbits is number of bits to shift out (integer)
      ... bitsOut is bits to shift out (integer or string)
      ... bitIn is received bits which were read (integer  or string)
```

These functions are performing an DRSCAN or IRSCAN. The variables `bitsIn` and `bitsOut` have to be integers of the maximum size of 32bit.

If larger bitchains have to be shifted strings can be used to parse them. In such a case the first two characters characterize the integerformat of the string. For example `'0x0f'` is an 8 bit hexadecimal number and `'0i00001111'` is the equivalent binary number. If *nbits* is greater than the values of the parsed string, the string will be filled up to get to this size.

If a string was parsed, the returned variable `bitsIn` will also have a string format (to allow greater bitchains). This string will contain a hexidecimal number.

Those conventions are used in both functions `drscan` and `irscan`.

```
int = chainscan(chain)
      ... chain is the JTAG chain (integer: 1,2 or 3)
```

`chainscan` scans a chain an returns the number of attached devices as an integer.

```
tuple = getid(chain)
      ... chain is the JTAG chain (integer: 1, 2 or 3)
```

`getid` finds the ID's of all attached Devices of a JTAG chain. The returning value is an integertuple and contains the deviceID's. (It is using `chainscan` to find the number of attached devices).

```
int = chainreset(chain)
      ... chain is the JTAG chain (integer: 1, 2 or 3)
```

This function resets the JTAG chain and returns the new JTAGState as an integer.

```
int = statemove(chain, endst)
      ... chain is the JTAG chain (integer: 1, 2 or 3)
      ... endst is the state to move on (integer)
          possible Endstates are:
              0 ... ENDST_TLRST
              1 ... ENDST_RTIDLE
              2 ... ENDST_PSDR
              3 ... ENDST_PSIR
```

`statemove` changes the JTAGState by moving (not jumping) to the new state defined by `endst`. It returns the new JTAGState (note: those integers are different to the numbers above).

3.5 Exceptions

The ccpc module uses the standard built-in exceptions provided by python. In these a significant string is set to describe the type of the error and where it appeared.

It is possible to write programs that handle selected exceptions without exiting the whole program, as the following code shows.

```
>>> for addr in range(0x10000, 0x10010):
...   try:
...       lbread(8, addr, 0xff, 'w'))
...   except IOError:
...         print 'IOError on address ', addr
```

In this case, if an IOError occurs, the program will not exit. The Message shown above will be printed and the program will continue.

In the ccpc module just three of the standard built-in exceptions are used to provide elementariness. The significant string set in the module gives more details about the occured error. The three main-errors are:

`StandardError`	...	if parsed arguments are incorrect
`TypeError`	...	parsed arguments are of invalid type
`IOError`	...	libraryfunctionerror, boarderror, mallocerror, writeerror etc.

4 Simple python examples

The following examples should give a short introduction about how to use the module 'ccpc'.

The first python code uses the lblib functions `lbread` and `lbwrite`. It reads 8 values from the local-bus, shifts those values and writes them on the same address.

```
import ccpc

addr = 0x13000
width = 8

tuple_a = ccpc.lbread(width, addr, 8)
print tuple_a

tuple_b = () for pos in range(0,len(tuple_a)):
    tuple_b = tuple_b + (tuple_a[pos] >> 1,)

print tuple_b
ccpc.lbwrite(width, addr, tuple_b)
```

The second python code uses the i2clib functions. It scans addresses on the i2c busses in readmode. The readable are printed. This code is similar to the commandline utility 'i2cscan' mentioned in [2].

```
from ccpc import *

for bus in range(0,4):
    print "scan␣bus:␣", bus

    for addr in range(0, 0x100):
        if i2cscan(bus, addr, 'r'):
            print hex(addr)

print 'finished'
```

The last python code uses the JTAGlib functions. This one scans a chains, gives the number of devices and the belonging ID's. Then it writes the `'101'` command (= `0x06` in hex) via irscan. The benefit is that now drscan is able to read the device ID directly from the buffer (provided that it shifts 32 bits).

```
from ccpc import *

chain = 1
print "————————————————————"
print "chain", chain, "␣−␣devices", chainscan(chain)
a = getid(chain)
print "ID", hex(a[0])

# read the ID via drscan
print "irscan:␣bitsOut:", hex(6)
irscan(chain, 3, '0i101')

print "drscan:␣bitsIn:␣", drscan(chain, 32, '0xfeedbabe')

print "————————————————————"
```

The last code produces the following output[3].

```
——————————————————
chain 1   – devices 1
ID 0x40a093
irscan: bitsOut: 0x6
drscan: bitsIn:  0x0040a093
——————————————————
```

5 References

[1] Python 2.2.3 Documentation, `http://www.python.org/doc/2.2.3/`

[2] CreditCard PC softwareguide, Niko Neufeld EPFL & CERN, LHCb note, `http://cern.ch/lhcb-online/ecs/ccpc/software`

[3] boost C++ Libraries, `http://www.boost.org`

[3]produced on the 8.Sept 2005 on the CreditCardPC 'pclbcc06'