GETTING STARTED WITH THE EMP FRAMEWORK – PART 6

ANDY ROSE, IMPERIAL COLLEGE LONDON

CREATING AN ADDRESS TABLE

- Copy the example into the local repo
  - Could start from scratch but that would be tedious

```
mkdir ~/my-firmware/src/my-algo-repo/an-algo/addr_table

cp ~/my-firmware/src/emp-fwk/components/payload/addr_table/emp_payload.xml
   ~/my-firmware/src/my-algo-repo/an-algo/addr_table/my_payload.xml
```

- Modify the dep file to point to the new address table:

```
addrtab my_payload.xml
```

CREATING AN ADDRESS TABLE

- Modify the address tables to describe the registers you have just created
- Change

```
<node id="dummy_payload" address="0x0" fwinfo="endpoint;width=31"/>
```

- To

```
<node id="My 1st register" address="0x00"/>
<node id="My 2nd register" address="0x1F"/>
                    ⋮
```

CREATING AN ADDRESS TABLE

- You can also add a permissions attribute
  - “read” or “r”
  - “write” or “w”
  - “readwrite” or “rw” (default)
- You can also add description attributes to document each entry

RAMS AND FIFOS

- RAMs and FIFOs have two additional attributes
- Attribute “size” specifies the maximum number of contiguous reads/writes
- Attribute “mode” specifies whether or not to increment the IPbus address on each operation
  - “non-incremental” or “non-inc”
  - “incremental” or “inc”

CREATING AN ADDRESS TABLE

- XML is naturally hierarchical
  - Just like modules within modules within modules in firmware
- IPbus address tables support hierarchy and referencing of other files
- Note:

```
<node id="My 1st register" address="0x00"/>
<node id="My 2nd register" address="0x1F"/>
```

Addresses are relative to their parent node, so we only list the part we actually set

CREATING AN ADDRESS TABLE

- Even though IPbus is word-oriented, it is convenient to be able to label individual bits
- This is done using the “mask” attribute

```
<node id="register" address="0x0">
  <node id="LSB" mask="0x0000001"/>
  <node id="MSB" mask="0x8000000"/>
  <node id="The_Rest" mask="0x7FFFFFE"/>
</node>
```

- All masking and bit-shifting operations will be done “under-the-hood”
  - i.e. Reading “MSB” above will return 0x0 or 0x1, not 0x8000000

WARNING

- EMPbutler is an “easy access” tool
  - It is to get systems up-and-running quickly
  - It is for beginners

WARNING

- EMPbutler is an “easy access” tool
  - It is to get systems up-and-running quickly
  - It is for beginners
- EMPbutler deliberately does not implement a general-purpose write-access method
  - Since that would be like handing a monkey a loaded gun

WARNING

- The safety mechanism on the gun is that the user must write code to perform write-access
- Via
  - C++
  - Python

UHAL

- uHAL is the software partner to IPbus
- Written in C++
- Compiled & distributed as “library + headers” via RPM for a number of platforms
  - Available through your favourite package-manager
- Can also be compiled from source

USER-SIDE C++

- For long-term experimental running, you will most-likely use the C++ API
- Just a matter of building against the libraries
  - For GCC

```
-I/opt/cactus/include -L/opt/cactus/lib
-lpthread \
-lboost_filesystem -lboost_regex -lboost_system -lboost_thread \
-lcactus_extern_pugixml -lcactus_uhal_log -lcactus_uhal_grammars \
-lcactus_uhal_uhal
```

USER-SIDE PYTHON

- For simplicity of getting started, Python-bindings of the C++ library are also provided
  - We will use these
- The Python API deliberately matches the C++ API very closely
  - So porting from Python to C++ should be relatively painless
  - Simply replace Python lists with C++ std::vector

LET'S WRITE SOME PYTHON TO WRITE SOMETHING

- In the my-software directory, create a new python script
- Obviously, the first thing to do is to import the API
  - Add `import uhal`

LET'S WRITE SOME PYTHON TO WRITE SOMETHING

- We need to tell our code what hardware we are talking to
- Remember the connection file? We need it again!
  - Add `ConnFile = uhal.ConnectionManager("file://connections.xml")`
- And use the ConnectionManager to create a device by name
  - Add `MyBoard = ConnFile.getDevice("my-board")`

LET'S WRITE SOME PYTHON TO WRITE SOMETHING

- We need to tell our code what register we are talking to
  - Add `MyReg = MyBoard.getNode("payload").getNode("My 1st register")`
- We could also do `MyReg = MyBoard.getNode("payload.My 1st register")`

LET'S WRITE SOME PYTHON TO WRITE SOMETHING

- And we want to write some data to it, so
  - Add `MyReg.write(0xC0FFEE00)`
- Add as many writes to as many registers as you like

NOTE

- TO OPTIMISE BANDWIDTH USAGE, UHAL USES A DELAYED DISPATCH MODEL
  - THE WRITE OPERATION WE JUST PERFORMED DOES NOT ACTUALLY TALK TO THE HARDWARE, IT ADDS THE WRITE OPERATION TO A QUEUE
- This doesn't normally causes the user a headache for write operations, but read operations are a different matter

LET'S WRITE SOME PYTHON TO WRITE SOMETHING

- So let's commit our operations to hardware
  - Add `MyBoard.dispatch()`

EXERCISE

- Save your script
- Run it
- Use EMPbutler to inject a counter and capture some data
- Prove to yourself that your python script has done something

LET'S WRITE SOME PYTHON TO READ SOMETHING

- Let's read our board register back
  - Add `value = MyReg.read()`

LET'S WRITE SOME PYTHON TO READ SOMETHING

- Let's read our board register back
  - Add
```
value = MyReg.read()
```
- REMEMBER, uHAL uses delayed dispatch.
  - "value" is NOT a number
- Try adding:
```
print("{0:08x}".format(value))
```
- Running script should raise an exception

LET'S WRITE SOME PYTHON TO READ SOMETHING

- Add a dispatch between the read command and the print command
- Running the script should now print the value you programmed into the FPGA

IF YOU HAVE A LOT OF REGISTERS…

- You might have noticed that if you have a lot of registers, explicitly writing out each name would get a bit tedious
- uHAL lets you use Regular-Expressions to get a list of nodes matching names

```
MyRegList = MyBoard.getNode("payload").getNodes("My .* register")
```

or

Note the "s"

```
MyRegList = MyBoard.getNodes("payload\.My .* register")
```

- Which you can then iterate over

IF YOU HAVE A LOT OF REGISTERS…

- The “Node” object has the methods
  - `getId()` to return it’s “local” name
  - `getPath()` to return it’s “full” name
- Useful when iterating over objects returned by a regex search

OTHER USEFUL INCANTATIONS

- `getPermissions()` tells you whether it is “read”, “write” or “readwrite”
- `getMode()` tells you whether it is a register, a RAM or a FIFO
- `getSize()` tells you the size of the endpoint in 32-bit words

READING AND WRITING RAMS & FIFOS

- Accessing RAMs and FIFOs is similar to registers, but the API is

```
MyRAM.writeBlock( [ ... Data ... ] )

values = MyRAM.readBlock( Size )
```

- Similarly to the read command, “values” is not actually a list and cannot be used as one until a dispatch has been issued
- You can also

```
MyRAM.writeBlockOffset( [ ... Data ... ] , Offset )

values = MyRAM.readBlockOffset( Size , Offset )
```

EXERCISE

- Add a 1024 x 32-bit block RAM to your payload
  - Address the RAM with the bottom 10-bits of the IPbus address
  - Make it readable and writeable by Ipbus
  - Think carefully about the “ack”
- Create an entry for it in your address table
- Modify your python script to write a block of random data to it and read it back
- Verify that you get back what you loaded