Support for device drivers in TASTE

Version 1.0 written by Maxime Perrotin (ESA) 15/09/2010

Introduction

TASTE tools allow generated software to communicate with the external world using device drivers. To be more precise, we distinguish three families of drivers:

1) drivers that are used for the transparent communication between TASTE sub-systems in a distributed environment : these drivers are handled by the middleware (Polyorb-HI) and are never in direct visibility to the user code. They appear only in the deployment view, and are associated to a communication bus (e.g. Spacewire, 1553, Serial...). The deployment view allows to select configure them.
2) Drivers that are used for the transparent communication between a piece of software generated by TASTE and a hardware components implemented in VHDL or SystemC (FPGA)
3) Drivers that are used to control devices which are not implemented using TASTE tools. This covers for example sensors and actuators which come with their own, legacy software and hardware interfaces.

This memo only covers the description of the third family of drivers: the control of sensors and actuators. From now on, we will call these components "blackbox devices", since we know nothing of their intrernal structure. We can only characterize them by their interfaces.

To make a quick summary, this note shows that blackbox devices are treated in the same manner as software components, using ASN.1 description of interface parameters, and that the toolchain automates the integration of the drivers in a seamless way, taking care of binary encodings, endianess issues, etc.

A case study illustrates our approach.

Case study

We want to control a robotic arm using an exoskeleton. The exoskeleton is a set of sensors that are put around the arm of an operator and that are connected to a computer. The computer receives the sensor data at a given rate, and transforms it into a set of commands sent to a distant mechanical arm. The objective is to allow remote control of robots with a "convenient" user interface.

Here is a picture of the exoskeleton we used for this study – for more information, check here: http://www.esa.int/TEC/Robotics/SEMA9EVHESE_0.html

Illustration 1: ESA Exoskeleton

The exoskeleton is connected to an acquisition board (PCI 6071E: http://sine.ni.com/nips/cds/view/p/lang/en/nid/1042). This board is placed in an industrial PC on a PCI bus. In the future, a cPCI equivalent board will run together with a Leon processor (keep this in mind, it is very important to understand the benefit of the TASTE approach).

The input data is a set of 16 channels of voltage in the range 0-6V.

In order to validate our tools (without breaking everything!), we have started by connecting the output generated by TASTE to a 3D model of the real robotic arm we want to control. The interface is handled via a UDP port and commands have to be sent at a given frequency to the 3D application software.

The PC is running Linux.

We had at our disposal the code for a Linux driver that communicates with the acquisition board (interface to poll and get raw data) – the COMEDI driver (http://www.comedi.org/) and the UDP/IP API from Linux for the output control.

The challenge

What is important to understand about communication with external devices (sensors/actuators) is that there are two dimensions to consider: the low-level hardware interface, which consists in writing an API to send and receive raw data by accessing shared memory, registers, interrupt lines, etc., and the transformation of the raw, hardware-dependent data into data that can be understood by application code.

The first dimension is not easy and may require some knowledge that go beyond pure software and deal with specificies of the hardware interface. However, it usually can be written once for all for a given family of hardware, since it is limited to providing a communication channel. It does not

directly address the capabilities of the hardware component in terms of functional interfaces. For example, a PCI low level driver will contain functions that will detect a device on a PCI bus (*any* device), and means to read/write from/to their addressing space. The challenge here is not to show how we can deal with this low-level driver (refer to this document for information for more information: http://polaris.dit.upm.es/~ork/download/21392_08.UPM.TN.01.I1R5_090923_drivers_manual.pdf)

In our case, we do have the COMEDI driver, and we can use it with no modification.

Now lets look at the second dimension, and related issues when software has to "talk" to hardware:

The most known issue is endianness : it is sometimes necessary to swap bytes in a stream of data if this stream was created by a computer that has a different memory representation as the one which receives it.

But endianness is not the only issue to deal with, of course. Legacy devices impose their data stream format, and software has to adapt. In some cases this job is very simple, in other cases it can be a very complex, laborious work to perform. And usually since interfaces are rarely frozen at the begining of a development, many adjustments and fixes have to be done frequently to make sure that everything is properly decoded – or encoded. If not impossible to manage, this is definately error-prone, and obviously not very exciting to program. In space applications, star trackers are a good example of such complexity since they come with a high number of interfaces, containing much more information that what is strictly used by the operating software.

To recall the philosophy behind TASTE interfaces and illustrate more concretely the challenge induced by this second "dimension" of device drivers, imagine that we want to command an equipment on or off. To do this with TASTE, you would simply define a command using ASN.1 :

Command-type ::= ENUMERATED { on, off }

And specify one interface "SwitchPower" with a parameter of type Command-type.

Now, if your equipment happens to be a star tracker, the **actual** command that has to be sent to the physical line would be a 48 bits stream, containing a complex CCSDS packet header with a device identifier, a version number (on 3 bits), etc, etc... And maybe only one bit would be used for the actual command (on/off).

Building this packet in an automatic way and sending it properly to the destination device is what TASTE is now capable of doing.

Let's come back to our case study.

The challenge applied to the case study

The acquisition board is build upon a little-endian architecture, and it sends sequences of 16 real numbers of 64 bits each (these 16 values comprise data for all the exoskeleton sensors) at a periodic rate.

The transformation algorithm (implemented in Simulink) takes 16 real numbers which value vary from 0,0 to 6,0 (these raw values represent a voltage). Its output is a series of 3 triplets (x,y,z,p) followed by a sequence of 16 real numbers representing angles in radian.

The UDP port to which this result has to be forwarded has to send the values in **a different order**

from the one generated by Simulink.

We want to run the algorithmic application on a PC (which is little-endian, too) with Linux, but in the future we want to port it to Leon with RTEMS. Leon is a big-endian processor, and we do not want to rewrite the code to swap all the bytes manually – this is too much error prone!

Modeling with TASTE

In the TASTE approach we take the point of view of the application developer, who is concerned by the semantics of the piece of data he receives. What counts is the type of data and its range.

We modelled the input and output data using ASN.1 this way:

```
Analog-Inputs ::= SEQUENCE (SIZE(16)) OF REAL (0.0 .. 6.0)

-- Output types

VR-Model-Output ::= SEQUENCE {
    x1 REAL (-1000 .. 1000),
    y1 REAL(-1000 .. 1000),
    z1 REAL(-1000 .. 1000),
    p1 REAL(-1000 .. 1000),
    x2 REAL(-1000 .. 1000),
    y2 REAL(-1000 .. 1000),
    z2 REAL(-1000 .. 1000),
    p2 REAL(-1000 .. 1000),
    x3 REAL(-1000 .. 1000),
    y3 REAL(-1000 .. 1000),
    z3 REAL(-1000 .. 1000),
    p3 REAL(-1000 .. 1000),
    j-rad SEQUENCE (SIZE(16)) OF REAL (-1000 .. 1000)
}
"DataView.asn" 37L, 638C
```

This model, destinated to the application developer, does not contain any information on how the driver will actually build his packet.

This essential information is placed in a **separate** ASN.1 model describing using a dedicated binary control notation (ACN) how the packet expecteed by the driver has to be formed.

```
DataModel DEFINITIONS ::= BEGIN

        /*Input types*/

        Analog-Inputs[size 16] {
                dummy [encoding IEEE754-1985-64, endianness little]
        }

        /*Output types*/
        VR-Model-Output []{
                j-rad [size 16] {
                        dummy [encoding IEEE754-1985-64, endianness little]
                }
                x1 [encoding IEEE754-1985-64, endianness little] ,
                x2 [encoding IEEE754-1985-64, endianness little] ,
                x3 [encoding IEEE754-1985-64, endianness little] ,
                y1 [encoding IEEE754-1985-64, endianness little] ,
                y2 [encoding IEEE754-1985-64, endianness little] ,
                y3 [encoding IEEE754-1985-64, endianness little] ,
                z1 [encoding IEEE754-1985-64, endianness little] ,
                z2 [encoding IEEE754-1985-64, endianness little] ,
                z3 [encoding IEEE754-1985-64, endianness little] ,
                p1 [encoding IEEE754-1985-64, endianness little] ,
                p2 [encoding IEEE754-1985-64, endianness little] ,
                p3 [encoding IEEE754-1985-64, endianness little] ,
        }

END
```

If you look carefully at these definitions, you will see that:

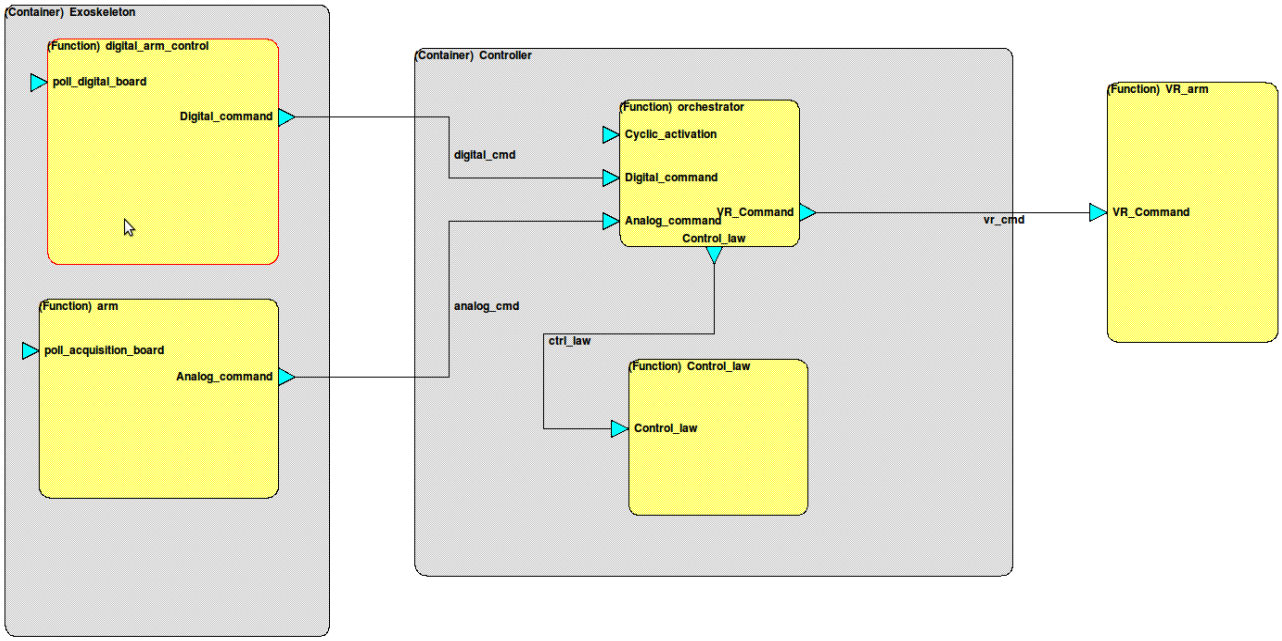
1) they enforce the little endianness attribute for all data

2) they enforce the size (64 bits) of each field
3) they enforce the re-ordering of fields to comply with the requirements

In summary, this specification provides all the details that are needed to make sure that whatever the platform the algorithm will run on, data will always be interpreted correctly – and the TASTE ASN.1 compiler invoked with this model provides the code that guarantees it.

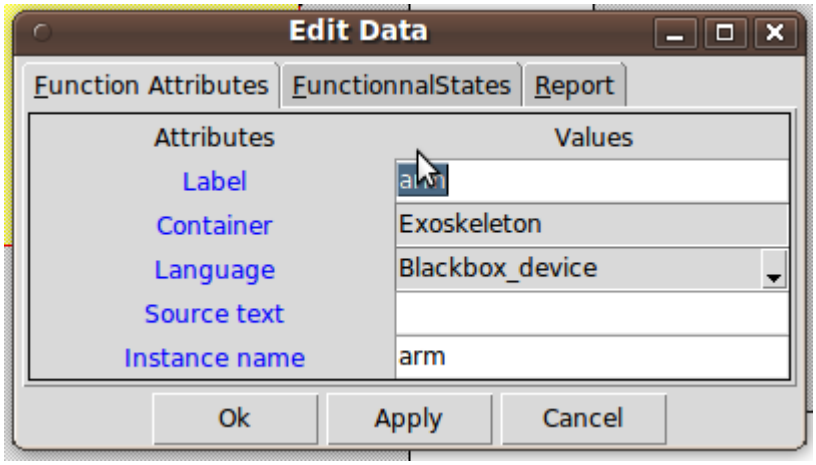
The driver itself will only have to take the encoded packets, without having to look into them, and place them on the physical line.

Now lets look how we used this information in a complete TASTE model. First the "interface view" captures all the functional blocks, i.e. the exoskeleton, the algorithmic block, the 3D output processor – and a bit more because we did a sligthly more complex demonstration in practice..



In the picture above, the "arm" on the left corresponds to the input sensors of the exoskeletons (as described earlier in this memo), the "VM_arm" function on the right corresponds to the 3D model, and the "controler" container in the middle contains the various software function that interact with the hardware blocks.

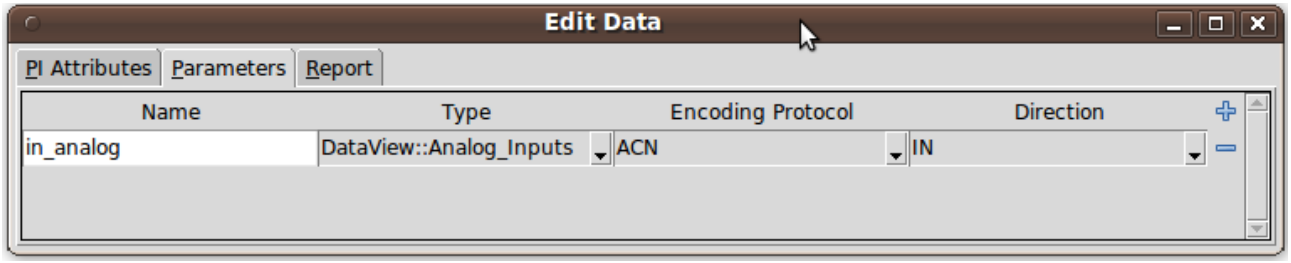
If we double-click on the "arm" function, we'll see some interesting attributes:



The language has been set the "Blackbox device". This instructs the toolchain to treat the associated code as driver code. This has a major consequence for the code, because the toolchain expects the block to be responsible for the encoding of the data that is passed as parameters or required interfaces. In other words, the data that this block will receive from the hardware will be left raw. More on this just below.

This block has a provided interface called "poll_acquisition_board", which will be activated to check every millisecond if some new data is available from the input board. In practice this means that after the vertical transformation has been performed on this model, the "arm" driver will consist of one periodic thread (refer to the vertical transformation rules for more information – I know, they are not documented).

Lets also look at the required interface of this block, called "Analog_command".



It contains one attribute (named "in_analog"), of the ASN.1 type named "Analog_Inputs" (described above), and it states that the encoding is "ACN": this will instruct the toolchain to use the binary encoding definition that we presented in the previous section.

So what happens is that the "arm" block will read, using the low level driver, the raw data from the aquisition board, and will pass it unmodified as a parameter of the required interface "Analog_command", when invoked. There is no risk of mis-interpretation due to hardware platform dependence - this data will be properly decoded to a data structure that will be used natively in the processing block called "orchestrator".

On the other side (control of the remote arm, in our case the 3D model), it is similar: it is defined as a blackbox device and there is an ACN-encoded parameter.

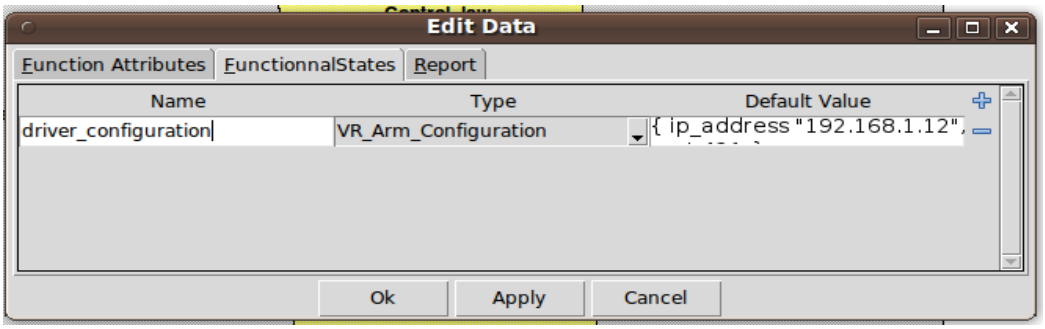
But there is more. Remember that the 3D model is running on a remote machine, connected with an Ethernet connection – using UDP to communicate.

For this, and in general for most drivers, this "Blackbox_Device" needs some configuration parameters, which are dependent on context.

We did not show it in the ASN.1 model earlier, but each blackbox device can optionally come with some additional data type definitions that allow to specify how the device shall be configured. In our case, we have the following configuration type:

```
VR-Arm-Configuration ::= SEQUENCE {
        ip-address OCTET STRING (SIZE (15)),
        port       INTEGER (0..16535)
}
```

And in the Interface view, there is a new box dedicated to these parameters – called "Functional states":



The "Default value" field allows the user to specify his context-dependent parameters, in a standarized format (ASN.1 value notation).

When the toolchain is invoked, it generates a C variable with the equivalent information, translated by the ASN.1 compiler (which will also make syntax and semantic checks on the values entered by the user). The user code that implements the initialization function of the driver can directly use this variable to properly startup the driver.

Conclusion

This way of describing and interacting with drivers is powerful in the sense that it gives to the designer a lot of possibilities. In particular, it lets him benefit from the standard provided interface attributes for all the driver interfaces. An interface can be "blocking" (if specified as protected or unprotected – for synchronous calls) or asynchronous (*send and forget* approach). Minimal inter-arrival times can be specified to make sure that an output to a hardware device is not made outside some allowed ranges of frequencies. FIFOs can be dimensioned to make sure no messages are lost ; context parameters allow driver-specific attributes to be captured at interface view level, which in most cases reduces to zero the manual interventions in the driver code once the low level hardware access functions have been implemented and tested.

ASN.1/ACN proves here also to be a key technology, by providing an homogeneous way of specifying data not only between software blocks implemented in heterogeneous languages, but also to communicate with hardware with almost no cost/effort. With one single, standardized language, all communication is automated between all parties in the system ; this is the main strenght of the TASTE toolset.

Regarding the case study, now that the approach has proven to work efficiently, it will be extended to replace the 3D model with the real arm, and later a much more complex exoskeleton will be used.