IFAC
Publications

A MODULAR SOFTWARE ARCHITECTURE FOR EMBEDDED SYSTEMS

Prof. Dr.-Ing. Dagmar Meyer

Institute of Electrical Systems Engineering and Automation
University of Applied Sciences
Germany, D-38302 Wolfenbüttel
e-mail: dagmar.meyer@fh-wolfenbuettel.de

This paper presents a modular software architecture for embedded systems, which is based on the open source operating system Linux. It uses Linux device drivers to control proprietary hardware components and provides a universal communication interface that allows to control the system by arbitrary control programs running under any operating system. The architecture has been realised exemplarily for the firmware of a data logger used for remote logging of magnetotelluric data. *Copyright © 2006 IFAC*

Keywords: Software engineering, embedded systems, programming approaches, object modelling techniques, object oriented programming, operating systems.

1. INDTRODUCTION

In the industrial environment, software development gains in importance. As the costs of the development of the firmware exceed the costs of the hardware components and their development by now, the aspect of reusing software components is focussed. In times of drastically shortened development cycles, the software architecture developed for a product family must guarantee maintainability, expandability and reusability.

In this context, the choice of the operating system also plays an important role. Some years ago, embedded software was often not based on any operating system at all. The necessary mechanisms as for example a file system and concurrency were developed "by hand". But abandoning the option to use an off-the-shelf operating system means longer development times, less portability to other hardware platforms because of strong hardware dependencies and sometimes less expandability. On the other hand, relying on an operating system means depending of a supplier in terms of bugfixing and availability of special releases. Last but not least, the costs of an embedded operating systems are not negligible.

For all these reasons an open source operating system was chosen as platform for the software architecture that will be described in the following sections.

2. THE DATA LOGGER AS CONCRETE EXAMPLE

Before deriving the software architecture, a brief overview of the hardware configuration of the sample system is given to facilitate understanding of the proposed architecture.

2.1 The Hardware Configuration

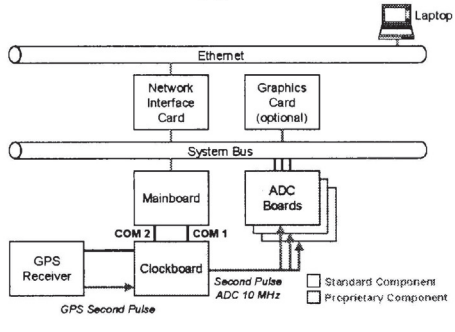

Fig. 1. Hardware configuration of the data logger

Figure 1 outlines the main components forming the hardware of the measurement instrument. Most of them are standard PC hardware components. Only two proprietary boards are included in the system (shown in grey in figure 1).

Standard PC components. The core component is a standard PC board (actually, PC/104 with 386 processor) with integrated Ethernet network adapter. A graphics card can be installed as an option. It is not necessary in normal operation mode, but may be used for installation and debugging purposes. These PC components may be replaced by different models without need of any software changes except of appropriate device drivers provided and used by the Linux operating system.

ADC boards. Two up to six ADC boards may be connected to the mainboard by the standard system bus. These boards are high precision ADC boards (resolution 24 bits) with a certain amount of memory on board. When this buffer is filled during a measurement, the board generates an interrupt. The system software has to serve this interrupt by reading the values stored in the data buffer as fast as possible in order to free it for further measured values.

Clock board. The clock board is linked to the mainboard by a serial RS232 interface (COM1). The clock board allows to synchronise the master clock frequency of an oven quartz with a highly stable 1 sec. pulse provided by a GPS module which is hooked to the clock board.

GPS module. A standard GPS module also linked to the mainboard by a serial RS232 interface (COM2) delivers this 1 sec. pulse. Also, it supplies the coordinates where the data logger is located and the actual UTC time. The UTC time is used to synchronise several data loggers in order to assure that the measurements carried out may be started simultaneously.

2.2 The Hardware Interfaces

Clock board and GPS receiver. The system software communicates with the clock board using commands coded as ASCII strings. These commands allow to read and write internal parameters and to start or stop the synchronisation process with the 1 sec. pulse delivered by the GPS receiver. The GPS module is addressed using a special interface protocol based on binary commands.

ADC boards. The ADC boards are addressed by writing to I/O addresses which provide access to the internal registers of the boards. Furthermore, appropriate handling of the interrupts generated by the boards is necessary.

Encapsulation. Due to the fact that all hardware interfaces are unique, it is absolutely necessary to encapsulate access to them within the software, so that changes do not affect the whole system.

3. THE COMPONENTS OF THE SOFTWARE ARCHITECTURE

3.1 Design Goals

The software architecture for the system must inter alia achieve the following goals:

- Strong encapsulation of the hardware dependant parts of the software. Exchange of one or more boards should only require local software changes.
- The hardware dependant parts of the software are time-critical. The interrupts generated by the ADC boards must be served immediately. Time synchronisation with the GPS receiver requires non-interruptible code. Thus, operation within the Linux kernel is inevitable.
- The external communication interface of the software must enable the system to be remotely controlled.

3.2 The Modular Software Architecture

The UML deployment diagram in figure 2 emphasises the basic design approaches. The software is split into five parts:

- One device driver per board type for the ADC boards, the GPS receiver and the clock board.
- A driver interface for providing access to the device drivers and controlling the operating sequence of a single measurement.
- A control program which is used to manage the measurement jobs that have been configured by the user.

Device drivers. The device drivers encapsulate the hardware dependant parts of the software and operate in the Linux kernel execution environment. A separate driver is provided for each hardware component. The device drivers are loaded dynamically to the Linux kernel at run time. The device drivers are addressed by simple function calls.

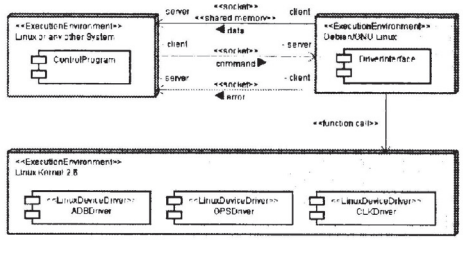

Fig. 2. UML deployment diagram

A good introduction to the concept of Linux device drivers within the Linux kernel version 2.6 is given in (Quade and Kunst, 2004) or (Salzmann, 2005).

Driver interface. This component operates in the "user space". The Debian/GNU Linux distribution was chosen as execution environment because it offers a fine granular configuration, which is an important feature to make it suitable for the application to embedded systems.

The platform independent communication interface of the driver interface component provides is based on three independent TCP/IP sockets, which form three "communication channels":

- The *command channel* is used to send commands to the driver interface, for example configuration data for a measurement, shutdown or a status request. Related to this communication path, the driver interface acts as the server.
- Via the *data channel*, the measured data is transferred to the control program. Communication across this channel is always initiated by the driver interface. Thus, it acts as the client. If the control program and the device interface are located on the same processor, this socket may be replaced by a shared memory area to accelerate data transfer.
- The *error channel* is needed to communicate error messages to the superordinate system. As errors may occur at any time and as they must be signalled as fast as possible, the driver interface again acts as the client.

Requirements for the superordinate system. The system communicating with the driver interface must provide the analogue communication channels.

4. XML CODED CONFIGURATION DATA AND ERROR MESSAGE DEFINITIONS

XML is used as data exchange format as far as possible. This format allows easy adding of extended commands, error messages etc.

4.1 Commands

The driver interface accepts simple commands like "startup" or "shutdown", but also complex configuration data for measurements. To minimise the complexity of parsing two different files have been defined, one for the simple commands and another for the measurement configuration data.

Figure 3 shows a sample XML file used to transfer simple commands to the device interface. It is sufficient to transmit such an XML file by any means to the device interface to stop a measurement, to configure or shutdown the system or to request a status information file. This concept supports simple and efficient testing of the system without the need to develop a special user interface.

```
<?xml-version = '1.0' encoding = 'utf-8?>
<commands>
  <Hardware_Version>
    <init>0</init>
    <config>
      <active>0</active>
      <irq>5</irq>
      <base_address>220</base_address>
    </config>
    <shutdown>0</shutdown>
    <stop_meas>0</stop_meas>
    <get_statusinfo>1</get_statusinfo>
    </Hardware_Version>
</commands>
```

Fig. 3. XML file used to transfer simple commands to the driver interface program.

The example in figure 3 shows how a status information request must be coded. Each command node contains an element named <active>. If its value is set to 1, the appropriate command will be carried out by the driver interface program.

The configuration data file is coded in a similar way, but it is of higher complexity and will thus not be presented here.

4.2 Error code definitions

All definitions of error messages including error codes, texts, severity information etc. are summarised in an XML file. This file is located in the file system. By this means, the user interface is automatically kept up to date concerning the error codes belonging to the actual system software. For adaptation to foreign languages, only an appropriate XML file must be supplied.

```
<Hardware_Version_Errors>
<!--Errors caused by the GPS receiver-->
  <GPSErrors>
    <ERR>
      <Type>ERROR</Type>
      <Code>000010000</Code>
      <Text>Open error device file</Text>
      </ERR>
      <ERR>
        <Type>ERROR</Type>
        <Code>000010001</Code>
        <Text>Close error device file</Text>
      </ERR>
      <ERR>
        <Type>ERROR</Type>
        <Code>000010002</Code>
        <Text>Write-command failed</Text>
      </ERR>
      ...
  </GPSErrors>
</ Hardware_Version_Errors>
```

Fig. 4. Excerpt of the XML file used for error code definitions.

Figure 4 shows an excerpt of a valid XML file used to store the error definitions. The errors are grouped according to the hardware or software component which caused it.

5. THE INTERNAL SOFTWARE ARCHITECTURE

The internal software architecture of the driver interface program is shown in figure 5. The classes shown in light grey form the external interface of this component. The ones shown in dark grey are the Linux device drivers, which are actually no classes but dynamically loadable drivers that have to be implemented in C according to the rules applying to Linux device drivers.

5.1 Wrapper classes

The program was implemented in C++. As many system functions like the socket interface and the thread library (Pthread library) are only available in C programming language, some wrapper classes were designed. These classes are applicable for many purposes and may also be used within the control program software, if it is also implemented in C/C++.

The XML library. Parsing XML data is done by using the LibXML2 library. It has been used because it provides a lean implementation of a simple SAX parser. This concept has – in contrast with the DOM model – much less memory requirements. The LibXML2 was again programmed in C and not in C++. A wrapper class called *BaseXMLHandler* was designed to cope with this incommodity.

5.2 The communication channels

The classes realising the communication channels are derived from the appropriate socket classes *TCPClientSocket* and *TCPServerSocket* and from the class *Thread*. This implementation is necessary because the sockets have to listen on their port and wait for incoming data.

This must be realised without blocking the other parts of the system. Furthermore, the communication channels are implemented according to the singleton design pattern (confer to Gamma et al., 2001). This pattern prevents a class from being instantiated more than once.

5.3 Layered architecture

The class diagram emphasises the layered architecture of the device interface program. The bottom layer is formed by the device drivers. Theses are addressed by the *DeviceInterface* class which implements a hardware independent interface named *UniversalDeviceInterface*.

The *DeviceInterface* class is addressed by the *CommandXMLHandler* class which parses the XML files containing commands and measurement configuration data. During a measurement, the instance of the *DeviceInterface* class uses a *DataReader* object to send the measured data to the *DataChannel* object.

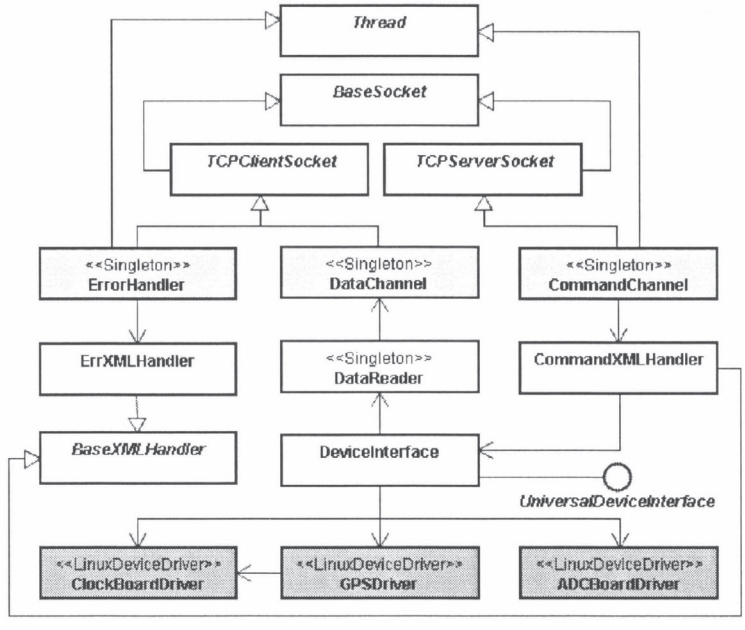

Fig. 5. UML class diagram of the device interface program.

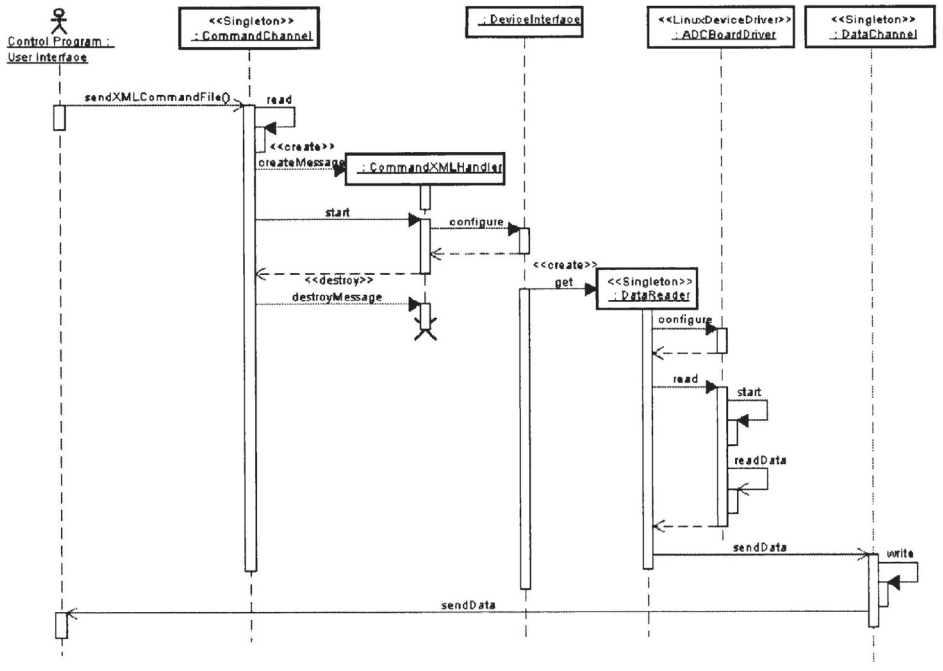

Fig. 6. UML sequence diagram demonstrating the operation sequence of a measurement.

5.4 Program operation

The sequence diagram shown in figure 6 demonstrates the internal operation of the device interface program when a measurement is started. This is done by the control program which sends a configuration data XML file to it.

The *CommandChannel* object reads the data from the socket. When a complete configuration file has been received, an instance of the *CommandXMLHandler* is created and started to parse the file. The *CommandXMLHandler* object passes the appropriate data to the *DeviceInterface* object. If no *DataReader* object exists so far, the *DeviceInterface* object creates one. This object has to configure the *ADCBoardDriver* and to start the measurement by calling its read method. When the driver has received an interrupt because the data buffer on the ADC board is full, the read command returns. The *DataReader* object then sends the data to the *DataChannel* object.

6. CONCLUSION

The software architecture which has been presented and realised exemplarily for a geophysical measurement device provides a highly flexible concept that may be applied to any embedded system. Usage of the Linux device driver concept provides strong encapsulation of all hardware dependencies and allows exchange of hardware components without any modifications to the software apart from the device driver.

The communication concept based upon sockets is platform independent and allows the system to be controlled either by a program which is running on the same processor and which may for example provide its output to a web server also hosted on the system or by an application running on a PC or laptop which realises a graphical user interface.

7. ACKNOWLEDGEMENTS

This work was supported by the "AGIP" (Arbeitsgruppe Innovative Projekte beim Ministerium für Wissenschaft und Kultur des Landes Niedersachsen) under project contract F.A.-Nr. 2003.568

REFERENCES

Gamma, E., R. Helm, R. Johnson and J. Vlissides (2001). *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, München.

Quade, J. and E.-K. Kunst (2004). *Linux-Treiber entwickeln*. dpunkt.verlag, Heidelberg.

Salzmann, P. J. (2005). *The Linux Kernel Module Programming Guide. Version 2.6.1*. http://www.tldp.org