Computer Standards & Interfaces 37 (2015) 41–52

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Modular design of an open-source, networked embedded system

Ivan Cibrario Bertolotti [a,*], Tingting Hu [a,b]

[a] CNR—National Research Council of Italy, IEIIT, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy
[b] Politecnico di Torino, Dipartimento di Automatica e Informatica, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy

ARTICLE INFO

Article history:
Received 7 August 2013
Received in revised form 11 March 2014
Accepted 26 May 2014
Available online 5 June 2014

Keywords:
Open-source software
Embedded systems
Distributed systems

ABSTRACT

The ever increasing hardware capabilities typical of modern microcontrollers make it easier to add more and more functions to embedded systems, even for relatively low-end ones. In turn, this raises new requirements on their firmware, focusing on aspects like adherence to international and industrial standards, modularity, portability, fast time to market, and integration of diverse software components.
This paper shows, by means of a case study, how to design a full-fledged networked embedded system using only open-source components, including a small-scale real-time operating system. In addition, it highlights how different components addressed key design issues, like inter-task synchronization and communication.

1. Motivation

The complexity of modern embedded systems is constantly increasing, especially for what concerns their firmware, as they must perform more sophisticated functions than in the past. For instance, Internet connectivity and data logging on commodity devices—for example, a USB flash drive—are becoming widespread requirements, even on relatively low-end equipment.

Although, on the one hand, this process is made smoother by the substantial hardware capabilities nowadays provided by most microcontrollers, it raises new challenges for software development, too. As a consequence, topics like software modularity, portability, as well as the ability to conveniently reuse software components in multiple projects, are no longer relevant only for niche, high-end products. On the contrary, they will be of more and more widespread importance in the near future.

This paper shows, by means of a case study, how a fully functional networked embedded system, including its associated software development tools, can be designed and implemented only out of open-source components. Modularity and portability are emphasized by the use of a real-time operating system as foundation, while the adoption of open-source components maximizes re-usability and keeps software development cost and time low.

At the same time, contrasting the ways in which distinct components address the same design issues (like, for instance, the operating system interface for what concerns synchronization and communication among concurrent code) is helpful to better appreciate the trade-offs between them and their relative merits.

The paper is structured as follows: Section 2 outlines the general structure of the system and its foundation, that is, the open-source software development toolchain and a small-scale, real-time operating system. Sections 3 through 5 discuss in more details the most important system components, namely, the TCP/IP protocol stack, the USB-based mass storage system, and fieldbus connectivity, respectively. In Section 6 more information is given on the memory requirements and performance of the components discussed in the previous sections, which are often the most severe constraints in a small-scale embedded system. Section 7 concludes the paper.

2. Embedded system architecture

This section outlines the main hardware and software components of the system being considered in the case study. In addition, it provides some information about the software development toolchain used to build the executable form of the application software and the underlying system components. Neither the software development toolchain nor the real-time operating system will be discussed in detail in this paper, because they are readily available for many popular platforms, and hence, they can nowadays be considered a commodity item.

2.1. Hardware and software components

The hardware platform considered in the case study is built around a LPC2468 microcontroller [1,2]. It embeds an ARM7TDMI [3] processor core, running at a maximum speed of 72 MHz, and a variety of other peripherals, including an Ethernet controller and several asynchronous serial ports. The only external components required to gain Ethernet and

* Corresponding author.
E-mail addresses: ivan.cibrario@ieiit.cnr.it (I. Cibrario Bertolotti), tingting.hu@ieiit.cnr.it (T. Hu).

http://dx.doi.org/10.1016/j.csi.2014.05.004
0920-5489/© 2014 Elsevier B.V. All rights reserved.

TIA/EIA-485 (formerly RS485) [4] connectivity are an Ethernet physical layer interface (PHY) and a TIA/EIA-485 transceiver, respectively. The TIA/EIA-485 interface has been considered because it is the physical-level medium used by several widespread fieldbuses, for instance PROFIBUS [5] and Modbus [6,7].

A limited amount of flash memory and static RAM is available on chip, and more can be added by means of an external memory interface. Due to its characteristics and price tag, this microcontroller can be seen as a typical component for low-cost, embedded systems.

From the software point of view, the aim was to design a full-fledged embedded system comprising Internet and fieldbus connectivity, as well as access to a mass storage system based upon inexpensive USB memory sticks. The design emphasizes modularity as a means to achieve better code portability and make it easier to reuse it in different projects. The resulting system architecture is shown in Fig. 1. In the figure, reusable system code modules are represented by gray boxes and amount to a significant part of the total software load in typical applications.

2.2. Software development toolchain and operating system

The most natural choice for an open-source software development toolchain revolves around the GNU Compiler Collection [8] and related components, namely:

1. The `binutils` [9] package provides an ample set of tools to build, examine, and manipulate object and executable files. Most importantly, it also contains the assembler and the link editor.
2. The `gcc` [8] component includes compilers for many popular programming languages. In this case, it has been configured to build only the compiler for the C programming language [10], which is the programming language used by all the open-source components considered in the project.
3. The `newlib` [11] package contains a runtime library for the C programming language, including mathematical functions, specially tailored for embedded systems.
4. The `gdb` [12] component provides a command-line based debugger.

Table 1 lists the exact version of the components used in the case study. It should be noted that, although it is quite possible to build all toolchain components starting from their source code—like it has been

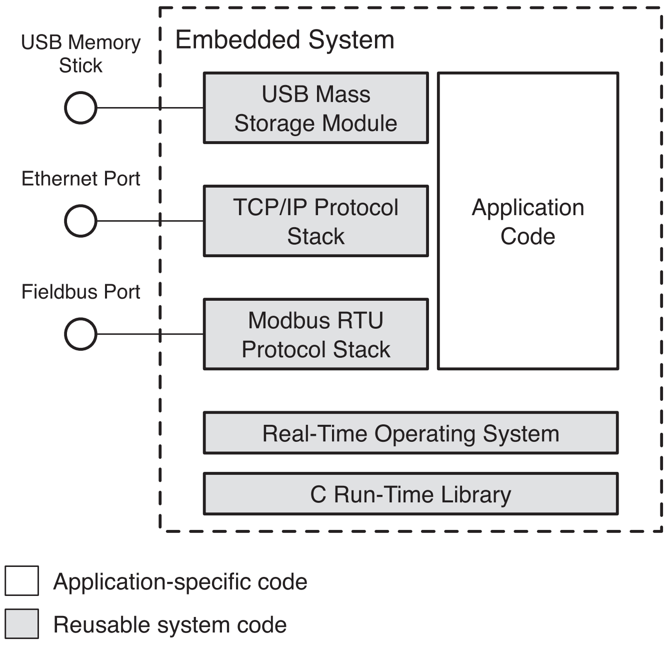

Fig. 1. High-level architecture of the embedded system considered in the case study.

Table 1
Summary of the open-source system components used in the case study.

Name	Version	Purpose
`binutils` [9]	2.20	Utilities
`gcc` [8]	4.3.4	C Compiler
`newlib` [11]	1.17	C Library
`gdb` [12]	6.6	Debugger
`FreeRTOS` [13]	6.0.1	Operating System

done in the case study—it is often faster and easier to acquire them directly in binary form. As an example, at the time of this writing, Mentor Graphics offers a free, lite edition of their Sourcery CodeBench toolchain [14], which is able to generate code for a variety of contemporary processor architectures.

A wide choice of real-time operating systems for embedded applications is nowadays available, for instance [15,13,16]. Basically, they represent different trade-offs between the extent of their application programming interface (API) and their memory and processor requirements. For this case study, the choice fell on the `FreeRTOS` real-time operating system [13,17]. This was done in order to minimize memory occupation, because memory is often a scarce resource in small embedded systems. At the same time, as it will be better detailed in the rest of the paper, this operating system still provides all the functions needed to effectively support all the other modules.

The adoption of a small real-time operating system like FreeRTOS on the embedded system platform of choice is usually not an issue. This is because these operating systems are designed to be extremely portable—also thanks to their limited size/complexity—and their source code package is likely to already support the selected architecture with no modifications required. In this case, a working C compiler is all what is needed to build and use them.

3. TCP/IP protocol stack

The TCP/IP protocol stack used for the case study is lwIP [18]. With respect to other competing open-source projects, lwIP was chosen because it is a good trade-off between tiny protocol stacks, like uIP [19], and feature-rich, Berkeley BSD-derived protocol stacks [20]. In fact, at one end of the spectrum, uIP aims at the absolute minimum memory footprint, even at the cost of sacrificing some useful features—most notably full reentrancy—to that purpose. On the other hand, the BSD protocol stack was originally designed for workstation-class machines and its footprint is often unsuitable for small embedded systems, as it is also pointed out in [19].

In addition, lwIP has already been used successfully for distributed computing with embedded systems [21] and its UDP protocol performance has recently been thoroughly evaluated in an embedded computing environment, with satisfactory results [22]. Last, but not least, another advantage of choosing a simple, streamlined protocol stack is that its internal structure is relatively easy to understand and well documented [19,23]. Hence, its adaptation to new processor architectures and network devices is faster, easier, and produces more reliable code.

As shown in Fig. 2, the lwIP code can be informally divided into four hierarchical levels. When lwIP is configured to support a single Ethernet interface, as in the case study being described here, the code is executed concurrently by (at least) three distinct tasks, listed in bottom-up order:

1. A low-level receive task associated with the Ethernet interface pulls the incoming frames from the network interface itself and pushes them into the main lwIP processing path.
2. The main lwIP task (called “tcpip” task in the lwIP documentation although, strictly speaking, it is quite unrelated to the TCP/IP protocol itself) accepts incoming frames from the receive task described above, as well as timer expiration events and application-layer requests, by means of a message passing interface. The reception of