Developing a low-cost general-purpose device for the Internet of Things

Adriana Wilde *IEEE(S)*, Richard Oliver and Ed Zaluska *SMIEEE*
Electronics and Computer Science
University of Southampton, United Kingdom
{agw106, rjo2g10, ejz}@ecs.soton.ac.uk

Abstract—**The *Internet of Things* is the concept of Internet-enabling physical objects. This paper describes a device to allow an object to be Internet-enabled using wired-Ethernet, in a power-efficient and low-cost fashion. This paper discusses a port of the *Contiki* operating system for use with the LM3S6965 processor. We demonstrate that it is possible to run the *Contiki* operating system on an Ethernet-enabled microcontroller and that such a low-cost device is suitable for both further research and inclusion in consumer products.**

Keywords–Internet of Things; Contiki; Internet-enabled physical objects; Embedded devices; Web of Things.

I. Introduction

The Internet of Things (IoT), or "embedded Internet" is widely seen as the next logical evolution of the Internet in a proximate fully-interconnected world [1]. The IoT extends the familiar structure of the Internet into a concept of networking generic physical objects [2], possibly including RFID (Radio Frequency Identification) tags, sensors and actuators [3]. These objects may then be combined into various different 'applications', much in the same way that we build smartphone or web applications at the moment [4]. Whilst much of the current work in this field is focused on Wireless Sensor Networks (WSN) and the use of the IPv6 over low-power Wireless Personal Area Networks (*6loWPAN*) scheme, there has been significantly less research into the wired `Internet of Things'. This paper discusses some of the challenges that must be addressed in this new era of Internet connectivity. In particular, we discuss the development of a low-power, low-cost, Ethernet-enabled, general-purpose multitasking device supporting modern open protocols that is suitable for further IoT development. Such a device has the potential to support the development of IoT applications, as smart-thermostats and other fixed-location devices.

II. Background

A. The Internet of Things (IoT)

The IoT is the concept of networking real-world objects and is regarded as the next logical generation of the Internet [1]. With an estimated hundred billion devices predicted to be connected by the end of the decade [5] many challenges must be faced to make this prediction a reality. Such challenges include having sufficient address space for hundreds of billions of devices as well as the methods of communication between such smart objects [1]. These objects may interact with other networked devices under one of the paradigms "person-to-device, device-to-device and device-to-grid" [5]. Examples of device-to device and device-to-grid communications could be the interaction between a smart alarm clock and coffee machine; and the smart monitoring of buildings respectively [5]. An example of a person-to-device communication could be something as simple as a doorbell [6] or the remote operation of a household appliance via the Internet or from a smartphone.

B. Communication

Whilst there has previously been much debate over which protocols to use for the higher levels of the model, the majority of current systems use either 802.15.4, 802.11 or Ethernet for the physical and data link layers [7]. However, Dunkels and Vasseur suggest that the Internet Protocol (IP) is a more suitable communication scheme for the IoT due to it being stable, scalable and lightweight [8]. Initially it was deemed unsuitable for devices with severe constraints in memory and power [9], but now it has been proven that it can be used under these scenarios [7] [10]. With the introduction of IPv6, there is now sufficient address space for billions of IoT devices to have their own IP address on the wider Internet without the use of non-global routable subnets. This would make IoT devices individually addressable on the global Internet. The use of pre-existing technologies and protocols should allow for easier integration and adaptability for use with current technology.

The REST architectural style (introduced by Fielding [11]) is often advocated as an application layer architectural style for communication between IoT objects [6] as it describes stateless client-server interaction with a uniform interface between components. REST relies on the principle that any form of information can be considered a resource and can be referenced uniquely whether it is static, dynamic, or even if it has not yet been created [11], allowing for manipulation via generic interfaces. Hypertext Transfer Protocol (HTTP) and Constrained Application Protocol (CoAP) are examples of RESTful protocols for the IoT. Though CoAP was designed with embedded devices in mind, HTTP is now more ubiquitous.

C. Operating Systems for Embedded Devices

A number of operating systems (OS) have been implemented to run embedded devices. We used Contiki OS, focusing on the code size and power consumption, often key issues, especially in the case of WSNs. The purpose of an OS can be summarized as providing hardware abstraction so that users (such as services and applications) may operate without requiring an intimate underlying knowledge of the specific hardware intended to be used, typically managing memory and the loading and execution of programs/services. Many modern OSs support time-sharing (or multitasking), allowing multiple services/applications to run concurrently. Typical examples of the hardware abstractions that OSs provide are I/O, access to permanent storage and access to the communications stack.

When selecting an OS for embedded devices, hardware abstraction and support for multi-tasking are desirable, as well as a closely integrated IP stack for communication. As the IoT encapsulates Internet-enabling everyday objects, the use of IPv6 is preferable due to the larger address space available. Another consideration lies with the constraints of the hardware available. With the introduction of the Contiki OS, the typical target devices were 8-bit microcontrollers with approximately 100kB of ROM and less than 20kB of RAM [12]. Features such as threading and IPv6 stacks, which are commonplace in larger OSs, are difficult to implement in an OS that is intended to run on a very limited hardware. Typically, when threading is implemented in a regular OS, each thread has its own stack [13]. This implementation of threading is problematic for the kind of hardware we describe as it allows various thread stacks to use up a large proportion of available RAM [13]. Similarly, implementing a standard IPv6 stack can also cause problems as the RFC4861 specification requires a 1280-byte packet buffer for every neighbour on the link layer [9]. This is also problematic for WSN where there are several neighbours at the link layer, again, due to limitations in RAM. The Contiki OS offers a solution to implementing both threading and IPv6 stacks in these resource-constrained systems.

D. Contiki

The Contiki OS created for WSN is a portable, lightweight operating system that supports dynamic loading, preemptive multi-threading, the protothreads programming abstraction for event-driven programs, and the uIPv6 stack for embedded devices [7] [12][13]. We note Contiki's most notable features.

The uIP TCP/IP stack, first introduced by Dunkels [10], was designed to have a minimal set of features allowing implementing a full TCP/IP stack on a resource-constrained system. The uIP stack now also supports UDP as well as TCP. The uIPv6 stack later introduced exposes the same interface as the standard uIP stack and fulfils all of the Phase-1 requirements of the IPv6 Ready program [7]. Contiki supports dynamic loading and replacement of programs and services at run-time, so all loaded programs share the same address space [12], in contrast to many other OS, which make use of virtual address space when loading programs dynamically. When Contiki was first introduced, *preemptive multithreading* was included as an application library that could be linked against applications that required it whilst still maintaining an event-driven kernel [12]. Preemptive multithreading is the ability of the kernel to interrupt and resume a process without that process having to voluntarily yield control. As explained above, the way threading is typically implemented is not viable for embedded systems. As Contiki implements this preemptive multi-threading as an application library, the memory requirements are greatly reduced as only the processes that require this feature are linked, reducing the number of individual stacks.

Contiki also provides the programming abstraction of Protothreads for event-driven programs development in a thread-like style [13]. This method does not require an individual stack for each thread, reducing memory requirements. Protothreads are architecture-independent and can be implemented entirely by the C Preprocessor [13]. Protothreads implement the blocking-wait mechanism of threading which allows a thread to yield control until certain conditions are met. The use of protothreads, as opposed to pre-emptive threading mentioned earlier, only incurs a penalty of one or two bytes per protothread (instead of a separate stack) and only a few extra processor cycles when compared to the state machine method of event-driven programming [13].

E. Compiling and Executing Code on the Cortex-M3

Before executing a program on an embedded system several issues must be addressed. For example, when executing code on a Cortex-M3, the initial stack pointer must be set up and the NVIC table must be in the correct place in memory. The correct setup of environment can be achieved by using linker scripts and start-up code to ensure that the relevant variables are correctly located in SRAM before execution.

The ARM Cortex-M3 is capable of achieving very low-latency exception handling. The NVIC table is used to define Interrupt Service Routines (ISR). This table, although initially located at address 0x0, can in fact be moved to almost any other address location [14]. This is of great benefit as interrupts can be registered and unregistered at run-time.

Also, on the Cortex-M3, when an interrupt is serviced, the code that was previously running is preempted. The state of the processor is automatically placed on the stack whilst the interrupt is handled before being restored on completion [14]. This operation incurs no instruction overhead on the Cortex-M3 [14]. This is useful as it means interrupts can be quickly serviced whilst having no noticeable effect on running code.

III. DESIGN

A. Hardware Design

The Stellaris 6000 series of microcontrollers was chosen as it is based on the ARM Cortex-M3 Micro-Controller Unit (MCU). The Cortex-M3 was chosen as it is a 32-bit processor with high performance and low power consumption. The Cortex-M3 also supports JTAG [1] programming/debugging which makes for easier implementation and testing. Also, the Stellaris 6000 series of microcontrollers is the only series currently available with a "fully-integrated Ethernet MAC and PHY layer" [2], which is desirable in order to keep a low component count (and cost). The reduction of component count can also aid in the PCB routing of future hardware designs.

The LM3S6965 MCU was chosen as a microcontroller due to its good balance of peripherals and flash SRAM memories[3].

[1] Joint Test Action Group (JTAG), used to debug embedded systems.

[2] http://www.ti.com/mcu/docs/mculuminaryfamilynode.tsp?sectionId=95i&tabId=2599&familyId=1758

[3] http://www.ti.com/mcu/docs/mcuorphan.tsp?contentId=135688&DCMP=Stellaris-Dustdevil&HQS=Other+OT+dustdevil

Although higher performance processors with a greater number of peripherals are available, features such as USB host and a higher maximum clock speed were set aside in order to achieve a lower cost. The LM3S6965 was also chosen as its development board, the ek-lm3s6965, is supported by the Open On-Chip Debugger (OpenOCD) software package. The use of open-source software was considered desirable. Also, using OpenOCD for programming/debugging was practical as the software allows control of the chip through either a telnet interface or by attaching a GNU Project Debugger (GDB) session. GDB allows code to be easily flashed to the microcontroller and facilitated the debugging with setting of hardware breakpoints. This interactive debugging of code on hardware meant that testing by simulation was unnecessary.

Originally, the intention was to create custom hardware using the previously mentioned LM3S6965 processor. However, we soon realised that the only gain from custom hardware when compared to the already available ek-lm3s6965 evaluation board would be a reduction in size and component count, which would make it “low-cost”. As the outcome of this project is targeted towards engineers wishing to internet-enable smart devices, it was the realisation that the intended stakeholders would in fact be creating custom hardware anyway making such work surplus to requirements and outside the scope of this project. It was, therefore, decided that the software development for the project would be carried out on the already available ek-lm3s6965 development board. The use of this development board also aided in much of the implementation and testing phase as both JTAG and the Universal Asynchronous Receiver/Transmitter (UART) were made available over USB.

B. *Software Design*

As mentioned in the previous section, the Texas Instruments Stellaris LM3S6965 MCU was chosen for use with the project. The choice of this MCU was of great influence in some of the toolchains and libraries used in this project. Of note, was the use of the Texas Instrument Stellaris Peripheral Driver Library, hereafter referred to as *Driverlib*. The use of *Driverlib* greatly reduces development time and the learning curve when attempting to control the peripherals of an unfamiliar architecture. Another advantage of using *Driverlib* is that a common API is made available for the entire family of Stellaris MCU. This will allow for the easy porting of any software written to other microcontrollers in the family, even what would otherwise be low-level, hardware-specific code.

Contiki was chosen as it is specifically designed for the IoT and it supports IPv4 and IPv6 as well as a wide variety of hardware, therefore portable to different architectures. Contiki supports many other useful protocols such as ARP for Ethernet, DHCP for auto-configuration, DNS to resolve domain names in addition to the essential TCP, UDP, and ICMP protocols. Also the implementation of protothreads is an appropriate choice for multitasking in such a resource-constrained MCU. It should be noted that Contiki also has many server ‘apps’ that can be used depending on the thoroughness of the OS port, such as HTTP, COAP, and several others that can be used as a basis for testing the system.

IV. TESTING BASIC FUNCTIONALITY

Throughout the Contiki port, as extra functionality was added, small applications either provided with Contiki or written for purpose were used to determine that correct operation was being achieved, such as a *printf* via a hello-world example, as well as checks on the successful port of *clock.c*. In an example process, the timer is set to a period of one second and the process yields. When the process is then given execution time again, a check is made to see if the timer has in fact expired. If so, the count variable (the number of times this behaviour has occurred) is printed to an attached serial console using the *printf* function call. The timer is also reset to one second. In this way, we can observe the process starting and then counting the seconds since the start of the execution. This value can be compared to that obtained by a stopwatch to determine if the program is exhibiting correct behaviour and thus determine the correctness of *clock.c.*

A. *Enabling Networking*

When undertaking the network port, a Driverlib example application using an old version of the uIP stack was the starting point. The 6502 processor’s Ethernet code provided by Contiki allowed adapting the Driverlib code by updating the uIP function calls and replacing any delays with Contiki’s *etimer*. Amongst other developments, DHCP was handled in a separate process to increase readability and the ease of the port. DHCP functionality was ascertained by running a DHCP server in foreground mode and observing the *DHCPDISCOVER*, *DHCPOFFER*, *DHCPREQUEST*, *DHCPACK* process.

A simple test of the device was a communication with a remote webserver, e.g. sending an HTTP POST to *http://posttestserver.com*, which allows received POST requests to be viewed in a standard web browser. Code using the DNS service was also written to test functionality. This code successfully resolved the IP address of the above site to its address. Code which allows posting twitter status updates was adapted so that the necessary HTTP POST requests could be sent to the test server (without authentication).

Correct functionality was observed with status updates made available over the UART peripheral and also by following the appropriate TCP streams using the network protocol analyser, *Wireshark*, allowing the observation of the operation of the DHCP, ARP, DNS and other protocols.

Finally, POST requests were confirmed by browsing to the correct subdirectories of the test server. The next test was to verify that the system could function correctly as a web server which was compiled and tested as working without further modifications. This webserver does not require the files it hosts to be present on a filesystem. Instead, a perl script is able to turn a specified directory structure into a series of *const char C* arrays. In this way, the files being served up by the webserver may be simply flashed to the microcontroller’s ROM at the same time the program is. Also, the default server supports the use of CGI scripts. These scripts allow for the dynamic generation of web pages by the server.

A copy of the included webserver was made to ascertain if a CGI script could be written that could modify server state or cause some kind of action to be performed. It was decided to create a web form to be hosted on the Contiki server that would allow a submitted message to be sent out of the UART peripheral. When the form is submitted, it is handled by *printf serial.shtml.* In this file, a call is made to the CGI script submit-message. This script causes the HTTP request the URL as well as any query strings to be printed on the UART peripheral. The script also causes the message to be displayed in the web browser as confirmation.

V. TOWARDS A FULLER EVALUATION OF THE DEVICE

At the start of this work, we set out to develop for the wired IoT. A shortcoming of the Contiki operating system we highlighted is the limited support for modern Ethernet-enabled embedded devices, and has been addressed by the development of an Ethernet-enabled embedded device. By porting the Contiki OS to the EKC-LM3S6965 development board, the suitability of Contiki for this hardware platform has been demonstrated. Successful operation of the system has been confirmed both by the verification of commonplace protocols on the system (ARP, DHCP, DNS, TCP, UDP, etc.) and by the construction of an example application, e.g. an approximation to a smart thermostat using an embedded webserver.

We have achieved our goals of creating an inexpensive, low-powered, general-purpose device for the wired IoT. The device created was both low-cost and low-powered, together with a low-component count, exhibiting a fully-working IPv4 stack over Ethernet with support for the TCP, UDP and ICMP protocols as well as accompanying protocols such as ARP, DNS and DHCP. IPv6 is supported, though this has still to be tested for this device.

In this prototype, an Internet-enabled device was built, which can be connected to a household heating system for example. The test device (see Figure 1) contains an incremental rotary encoder with a push-switch to modify the status of RGB Light-Emitting Diode (LED) of which the status could be set to a great range of colours. The brightness for each of the component colours (red, green and blue, hence RGB) can be set either directly on the device or via the Internet, where again, users had a choice of accessing the device via a website or an Android phone. This interface allowed both consulting and setting the status of the IoT device, as shown in Figure 2.

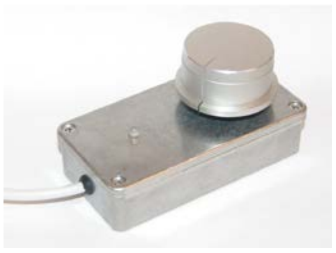

Figure 1. Completed test device

VI. CONCLUSION

A proof-of-concept device was developed, using relatively low resources, both in terms of materials and in time invested. The developed prototype is fully functional, demonstrating the feasibility of connecting an everyday object to the Internet of Things. With this prototype, an Internet-enabled device was built, with an interface allowed both consulting and modifying the status of the device over the Internet, clearly evidencing the viability for interconnecting everyday objects by producing a truly general-purpose device for the Internet of Things, one that is suitable for further research and possible incorporation into consumer products as the main application processor.

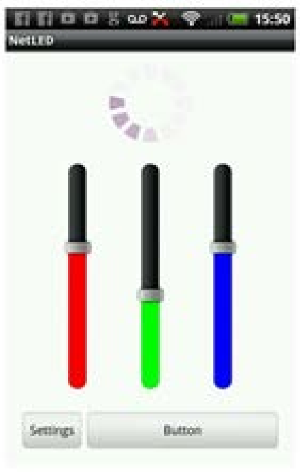

Figure 2. Android app for controlling the device over the Internet

REFERENCES

[1] Intel Corporation, "Rise of the embedded Internet," Tech. rep., 2009.

[2] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy, "Smart objects as building blocks for the Internet of things". Internet Computing, IEEE 14, 1 (jan.-feb. 2010), pp. 44 –51

[3] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," Computer Networks 54, 15 (2010), pp.2787 – 2805.

[4] D. A. Guinard, "A web of things application architecture - Integrating the real-world into the web". PhD thesis, University of Fribourg, 2011.

[5] The Hammersmith Group. "The Internet of Things: Networked objects and smart devices". Tech. rep., 2010.

[6] S. Hodges et al., "Prototyping connected devices for the Internet of Things". Computer 46, 2 (2013), pp.26–34.

[7] M. Durvy et al., "Making sensor networks IPv6 ready". 6th ACM Conf. on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session, Raleigh, North Carolina, USA, November 2008.

[8] A. Dunkels, and J. Vasseur, "IP for smart objects," White paper 1, Internet Protocol for Smart Objects (IPSO Alliance), July 2010. ver1.1.

[9] J. Abeillé, M. Durvy, J. Hui, and S. Dawson-Haggerty, "Lightweight IPv6 stacks for smart objects: the experience of three independent and interoperable implementations". White paper 2, Internet Protocol for Smart Objects (IPSO Alliance), November 2008.

[10] A. Dunkels, "Full TCP/IP for 8 bit architectures," 1st ACM/Usenix Int. Conf. on Mobile Systems, Applications and Services (MobiSys 2003), San Francisco, May 2003.

[11] R. Fielding, "Architectural styles and the design of network-based software architectures," PhD thesis, University of California, Irvine, 2000.

[12] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki: a lightweight and flexible operating system for tiny networked sensors". 1st IEEE Workshop on embedded networked sensors (Emnets-I), November 2004.

[13] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems". 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, Nov. 2006.

[14] ARM. "Cortex-M3 devices generic user guide," ARM Limited, 2010