RUDI VAN DRUNEN

small embedded systems

Rudi van Drunen is a senior UNIX systems consultant with Competa IT B.V. in The Netherlands. He also has his own consulting company, Xlexit Technology, doing low-level hardware-oriented jobs.

rudi-usenix@xlexit.com

EMBEDDED SYSTEMS CAN BE FOUND virtually everywhere. In this article I will describe some of the features of these small computing engines and will take you along the road to building and programming one yourself to ease simple control tasks in your daily life. In the embedded world, understanding issues of small systems is much easier when you have encountered them yourself.

Nearly all consumer electronics nowadays have some kind of microcontroller built in. Often these devices have small 8- or even 16-bit processors with on-chip peripherals, such as analog to digital (A/D) converters, input-output (I/O) ports, RAM, or ROM (flash), called microcontrollers, on board. These processor systems serve basic tasks that used to be taken care of by either mechanical devices or analog electronics. An embedded system is a small, reliable system specially designed and built to do one specific task, such as control your appliance, and cannot be user-instructed to perform other tasks easily.

A typical example of such an embedded system is in your microwave oven. The microwave often has a display, keyboard, and a number of sensors and actuators. Sensors can be a temperature sensor or the sensor that detects whether the door is closed. Actuators can be the electronic switch that controls a microwave tube or a system that controls the rotational speed of a fan. Other examples of embedded systems are the motor management systems found in all modern vehicles or controllers of all sorts, the power and temperature controller of your laptop or server, WiFi routers, and, to a lesser extent, game consoles, which feature a more complex software environment, including an operating system. You can see that the NTP server I described in [1] is also in this category.

Layout

Most, if not all, of the low-end embedded controller systems do not run an operating system that hides all devices for the user or developer. Instead, the user program is directly in full control of all hardware and there are no separate processes or tasks. Also, all interrupts on the system are to be handled directly by the user software. Often the interrupts are triggered by the hardware when asserting the interrupt pin on the chip to a specific logic level. This can be done by an external device or switch.

More complex embedded systems run specialized microkernels capable of job scheduling and doing real-time interrupt processing. In some cases this can be some flavor of UNIX, most commonly Linux or BSD. In this article we will focus on the systems that have no operating system, as these are the simplest to work with.

Hardware

There are a number of different families of embedded processors, all with their own architecture. Often the chip vendor manufactures a large number of different chips with the same core processor architecture, but with an enormous diversity of hardware interfaces, such as serial and parallel ports, analog inputs and outputs or even USB, or complete Ethernet interfaces with a hardware TCP/IP stack on the chip. This makes the development of hardware devices with these processors fairly easy. You don't need many chips to build a complete processor system. The downside is that the different families of processors all have their own architecture and instruction sets and need different development tools, so you may select a chip that has far too many I/O devices for your specific task.

Software

As I said before, the described embedded systems do not have an operating system, so development of software for embedded systems almost always takes place on a host system that has cross-compilers/assemblers to generate code for the target device. The target device is a board containing (at least) the processor and peripheral chips and sensors/actuators that will be used in the final application.

Lots of different vendors supply their own software or even hardware development environment for the processor family they make. These toolsets (and boards) often consist of an editor, a compiler/assembler, and a (down) loader. Unfortunately, almost all of these software tools run only under the Microsoft Windows environment. WINE can be of use here, but your mileage may vary, as vendor support may be very limited; especially when using complex debugging or download hardware that communicates to the IDE, you might encounter some problems.

Nowadays, it is quite common to do development for embedded systems in a higher language such as C or C++, in contrast to earlier development, which was completely done in assembly. By selecting C or C++ as a development language, it is fairly easy to use the well-known open source GNU toolchain (gcc) to generate code for a different architecture than is used to compile on. This process is called cross-compiling/assembling. After cross-compiling, assembling, and linking your program, libraries, and start-up code, the result is either a full binary file or an ASCII hex-file that contains the application and supplemental code. This file can be downloaded into the on-board or on-chip flash in the target system. A hex file is an ASCII file with the hex representation of the addresses and bytes that need to be there, with an additional checksum [2].

System Robustness

While writing software for embedded devices it is important to remember that these systems often will be running 24x7 for many years without being rebooted. Also, often these systems will be controlling devices that need safety precautions. Recently, embedded systems in cars have been in the news because of lack of software robustness [3]. If the software or the hardware of an embedded system fails, often dangerous situations can occur. Keeping this in mind, software robustness in embedded systems is very important. Also, the robustness of the hardware is a key factor. In software it is important not to use uninitialized variables, so be careful with dynamic allocation of memory, for example. On the hardware side, it is important to use a quality platform, good PCBs, and good manufacturing processes, and allow for timing and temperature margins in the hardware design.

Watchdog

To enhance system robustness it is wise to build in a watchdog system. A watchdog is a combination of a hardware timer, which resets the processor when it counts down, and some software statements throughout the main loop of the program. When running, the software statements reset the hardware timer before it runs out and resets the processor. If for some reason the software crashes, the hardware timer will not be reset and the system processor will be reset and the software restarted.

Downloading

Downloading the code onto the target processor or target board will get you to a bootstrap problem, as the processor generally has no code at that very moment and cannot help you with downloading the image into its flash memory. There is no code present to take a byte from an (e.g., serial) interface and write it to flash. So JTAG is often used to bring up a board from scratch. JTAG [4] is a bit-bang protocol that is supported by a number of different hardware components to do in-circuit production testing, but downloading bytes into memory devices or on-chip memory without using the processor is also supported. After downloading your binary image into the memory of the processor or on the board, the reset line of the processor can be released and the processor starts executing the just downloaded code.

Some embedded evaluation boards come complete with a processor that has a piece of code in its ROM that is either factory programmed or programmed into the memory using a chip-programmer. This piece of code is called the bootloader. It runs after reset and instructs the processor to accept databytes, often in a HEX file, and put them in memory. Then it instructs the processor to start executing the just downloaded code. This is by far the easiest way to program/download code into a processor or board. Often, the toolchain used contains special download software to do just this.

Debugging

As the target system often has no operating system, it can be quite difficult to debug an embedded program running on a piece of target hardware. A number of options are available here:

1. Use of a simulator/emulator running on a host machine, simulating the target processor and all peripherals. This system provides a complete simulation of the target in software and thus all software debugging features.
2. Use of an in-circuit emulator. This piece of hardware replaces the hardware processor and gives full control of all innards of the processor, including breakpoints.
3. Use of an in-circuit debugger, mostly connected to the system using JTAG (see above) if your target processor has features for debugging using these kind of tools.

4. Use of I/O, adding statements to your software to show the state on the available I/O: for example, displaying the state on an LCD display if available, or flashing LEDs on the board if your software reaches a certain state. The problem here is that you are actively interfering with the program code you are debugging.

Processor Families

In the embedded world there are a number of popular processor families. Next to the standard processors, some large vendors design their own processor using semi-custom chips and (VHDL) processor cores that they integrate. The most important and popular processor families are:

- **Intel 8051** [5] The 8051 is one of the earliest embedded processors controlling appliances. It is an 8-bit processor built on the Princeton architecture.
- **ARM** [6] The ARM family features a 32-bit core and is licensed to a number of different manufacturers that integrate the core with different peripherals on one chip. The ARM chip is a RISC architecture and capable of running at a fairly high clock speed.
- **PIC** [7] The PIC (Programmable Interrupt Controller) is a controller family built by the Microchip Corporation. They are available in different subfamilies featuring 8-, 12-, or 14-bit memory addresses, but all feature a small set of 8-bit instructions and have a Harvard architecture. PICs are generally popular with hobbyists and industrial developers. Microchip, as well as other vendors, supplies IDEs and compilers/assemblers for different languages. Some of them are open source, such as JAL [8].
- **AVR** [9] The AVR microcontrollers are 8/16-bit RISC devices built by Atmel. There are three subfamilies of the AVR chips: TinyAVR, MegaAVR, and xMegaAVR, each featuring more memory and complex peripherals. The latest addition to the AVR family is a 32-bit DSP-like architecture.

Of course there are many more microcontroller families with their own specific features and instruction sets, often targeted to a specific application area. Nowadays most applications requiring an embedded microcontroller are often built in an Application Specific Integrated Circuit (ASIC), where the processor often is integrated as a VHDL module together within the custom chip.

Practicalities

To get started building a small system and developing software for an embedded system, you need to consider a number of different things. First of all, you need to select the processor family for the job. This can be difficult, since many parameters are in play. One of the major issues is the number of resulting devices you need to produce. If you are building a one-off device for a project, you may want to focus on ease of learning to program the device. Next, the hardware interface possibilities (interface ports, analog and digital on the selected chip) and software development environment are important.

AN EXAMPLE

As a starting point, I suggest a readily available small controller board. This lowers the threshold to get started, especially for non-electronics engineers. The Arduino [10] board as shown in Figure 1 is a reasonably cheap AVR board with a powerful processor with many interface possibilities. This popular board features an AVR ATmega328 processor and an open source development environment. Following the Arduino concept and the same microprocessor, there are also a number of other boards available in the public domain, such as the JeeNode [11] or StickDuino [12].

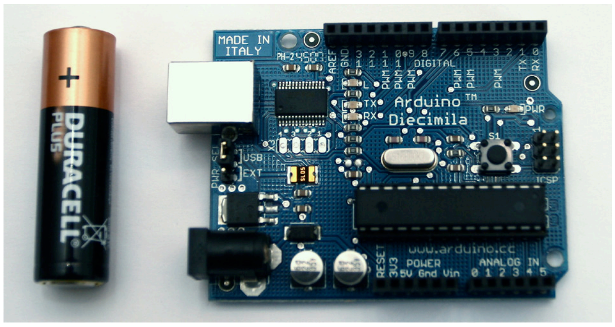

FIGURE 1: AN ARDUINO DIECIMILA BOARD WITH AN AA CELL AS SIZE REFERENCE

First of all, design your hardware and build a prototype target system. This can be as easy as using an on-board LED to experiment with. The Arduino Diecimila has an on-board LED connected to I/O pin number 13. With other boards you might need to wire up an LED as in Figure 2. The I/O ports of the processor can drive an LED directly. I recommend starting with something as simple as an LED to get used to the development cycle and the programming language.

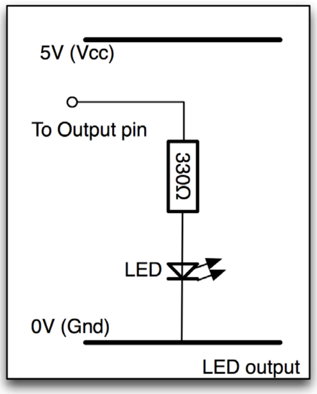

FIGURE 2: CONNECTING AN LED

The next phase is installing the development software on your host. The integrated development environment for Arduino is open source and uses the Processing [13] programming language. Processing looks a lot like C and was developed in a visual arts environment, but has matured to a production-ready development environment. In addition, a gcc toolchain is available to generate code for the Arduino [14]. The easiest way to start is to get the Processing integrated processing development environment for your platform (Windows, Linux, Mac OS X) and work with that. The Arduino IDE supports editing Processing with syntax highlighting and compiling/linking and uploading the code, as well as a simple terminal feature. The terminal can be used to connect to the target board, when you choose to output data using the serial interface. It also features a system to keep track of your libraries. The modern Arduino boards feature a USB interface to do the download and serve as I/O for the serial port, so you may also need to install a driver for the FTDI USB to serial interface.

The processor chip that is mounted on the Arduino board already has a boot loader program in part of the on-chip flash, so there is no need to add software on the target to communicate with the host machine. If you use an Arduino Diecimila board, the power can be delivered through the USB interface if you set the jumper accordingly, so you are ready to roll.

Now, as the Arduino Diecimila board has an on-board user-programmable LED, a good first program, which is called a *sketch* in Processing, should blink this LED. Therefore we initialize the I/O port, turn on the LED, wait and turn off the LED again, as shown in Listing 1. This code is available as a demo in the Arduino IDE under File -> examples -> Digital -> Blink.

```
/* Blink a LED on the Arduino Using Processing */

int ledPin=13;

void setup() {
  PinMode (ledPin, OUTPUT);
}

void loop(){
  digitalWrite (ledPin, HIGH);
  delay (1000);
  digitalWrite (ledPin, LOW);
  delay (1000);
}
```

LISTING 1: YOUR FIRST ARDUINO PROGRAM: BLINK THE LED CONNECTED TO I/O PIN NUMBER 13

The setup() method runs once, at start of the sketch, and the loop() method runs as long as the board has power or until you download another sketch.

After compiling and downloading the sketch, the board should start the sketch and the LED should blink.

You also can use standard UNIX utilities and a gcc toolchain that has been configured to cross-compile (avd-gcc) and generate code for the ATmega processor, if you do not like the Processing language or the IDE. Downloading code and data is done using an utility called AVRDUDE, the AVR Downloader/UploaDEr [15]. This program reads a hex-file, connects to the virtual serial port behind the USB interface, and speaks the Arduino bootloader protocol, which is standardized by Atmel as STK500 [16]. In addition, this program can use JTAG to program the bootloader in an empty chip.

Another relatively unknown language that supports the Arduino platform is called concurrency.cc [17]. It allows running Occam-like parallel programs on tiny devices.

Interfacing

There are libraries for other input and output devices, as well as a vast number of boards, called *shields* for the Arduino or *plugs* for a JeeNode, that connect to the processor board and have special functions, including Ethernet connectivity, zigbee interfaces, and interesting sensors such as temperature, GPS position, compass heading, barometric pressure, or force to connect your system to the real world.

CONNECTING INPUT

The ATmega processor that is used on the Arduino board has pins that can be configured as digital inputs. The status of these pins shows as the bits in a word that you can read in software. Standard practice is to pull the port high with a resistor and connect it to ground using the switch (see Figure 3). Also, use a small series resistor in case you make a programming mistake and program the port as output and set it high, so that you do not completely short-circuit it to ground and destroy your Arduino I/O pin.

Small switches can be read this way. An important thing to note here is that if your software polls the switch line, you will often detect multiple opens and closures when just pressing the button once. This phenomenon is called *bounce* and is caused by the mechanical nature of the switch. Building in a small delay when polling a switch will help de-bounce the input reading.

There are also a number of pins that take analog input voltages; the value of the voltage on the pin is shown as the value of a word that can be read in software. These pins use the on-chip analog-to-digital converter.

Loads of sensors nowadays have the analog-to-digital conversion built into the chip. The sensor measures an inherent analog value, such as temperature or barometric pressure, but has output pins that serially output the value using a protocol such as I2C, SPI, or 1-wire [18]. I2C, SPI, and 1-wire are all bus-like systems where you can serially transmit data from and to multiple devices using a small number of lines (wires). All of these protocols have their own software library that can be used in your user programs to communicate according to the protocol.

If you use the sensors with integrated AD conversion you do not need to take a lot of precautions to condition the (often very low) voltage analog signals using sensitive amplifiers and converters. Figure 4 shows a setup of how to connect a I2C device to an Arduino board. Here you see a compass sensor. The I2C library uses two analog input lines as digital inputs.

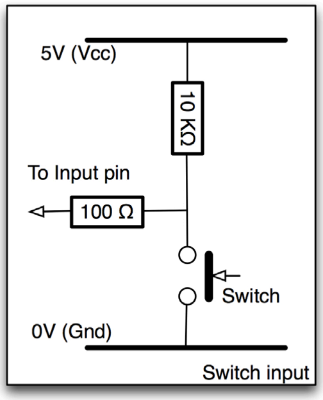

FIGURE 3: SWITCH INPUT

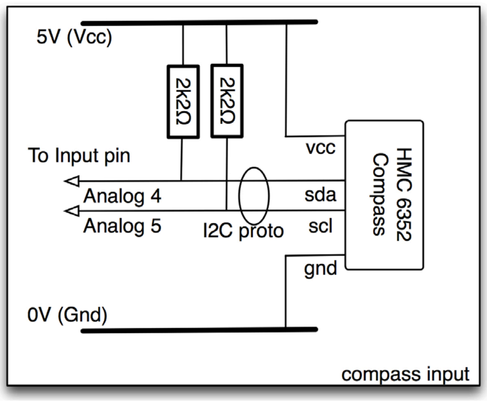

FIGURE 4: A SERIAL CONNECTION TO A COMPASS SENSOR

CONNECTING OUTPUT

The Arduino outputs a standard 0 or 5 volts level at its output pins. These output pins can drive small loads. For larger loads you need to amplify the signal or provide indirect control using a relay. There are special add-on boards for the Arduino that allow you to control small low-voltage motors or LED arrays. There are even small LCD displays with ASCII only or bit-image graphics that are supported by different libraries [19]. Often these displays are connected in parallel to the Arduino board, using four or eight data lines and some control lines connected to I/O ports.

You should be very careful if you want to connect the Arduino board to control a device that is at mains voltage. For this case a device called a solid-state relay (SSR) should be used, separating the low voltage from the mains. This building block accepts standard low-voltage output from an Arduino board and separates it optically, using an LED and a photo (light sensitive) transistor from the part that actually switches 110 or 230 volts. These devices come in a hermetically sealed and insulated housing, so apart from the terminals there are no live voltage-carrying parts outside. Figures 5 and 6 show such a device and the way it is connected. The additional advantage of using such a solid-state relay is that the device switches the load on or off at the 0 volts crossing of the AC mains voltage, reducing noise and power surges.

A note of caution is important here: *Make sure the mains connections are properly insulated and protected from accidental touch.*

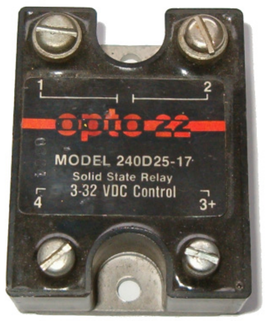

FIGURE 5: A SOLID-STATE RELAY

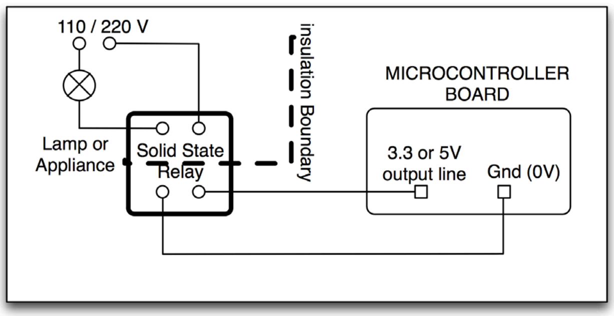

FIGURE 6: SETUP OF A SOLID-STATE RELAY TO CONTROL A HIGH-VOLTAGE DEVICE

CONNECTING TO A NETWORK

For the Arduino boards, there are different methods for connecting to an Ethernet network. All rely on separate boards that add an Ethernet interface. The most commonly used board holds an Ethernet chip (Whizznet W5100) that also implements the low levels of the TCP/IP stack [20]. It communicates over a three-line serial (SPI) protocol to the processor board and is supported by a library to read from and write to the network (address). The advantage of using just three lines is that a vast number of I/O lines of the processor stay available for your own I/O. The disadvantage is that more processor cycles are used to implement the SPI protocol.

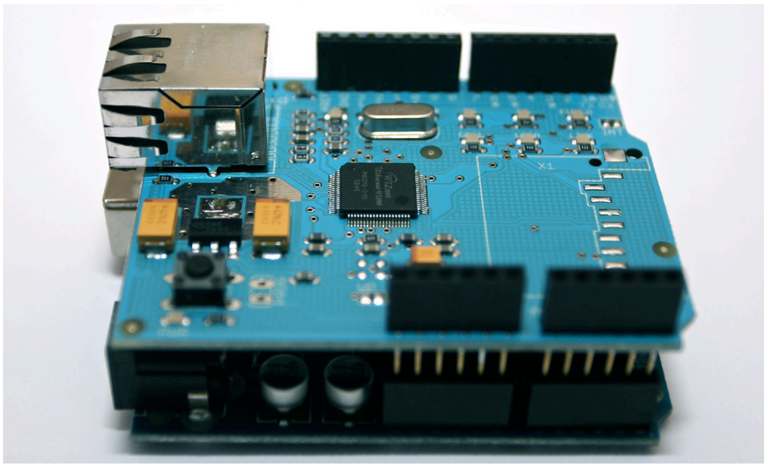

FIGURE 7: AN ARDUINO BOARD (BOTTOM) WITH AN ETHERNET SHIELD (TOP)

Applications

Applications for these small embedded systems are plentiful. They range from small systems such as an alarm clock or a controller board, to control devices such as fans in a server rack or electronic locks, to a more complex system that controls an autonomous robot or a sensor network measuring different environmental parameters at different locations and communicating wirelessly to a master device. Applications are not restricted to the technical field, as the creative crowd has also discovered the power of small controller systems to, for example, build art installations that emit sound or light and work cooperatively [21].

Conclusion

In contrast to the more complex systems we work with every day, there is a complete world of small controllers and computing devices that help us in daily life. Building a hardware-oriented small controller to help you with simple tasks is not difficult. There are a number of modern tools and ready-to-be-used hardware platforms that can be implemented easily, enabling you to build embedded systems with little effort while yielding great results.

ACKNOWLEDGMENTS

Thanks go to Rik Farrow for the comments on the initial draft of this article. A word of thanks also goes to Competa IT, my employer, for giving me the freedom to write this article

REFERENCES

[1] Rudi van Drunen, “A Home-Built NTP Appliance,” *;login:,* vol. 34, no. 4, August 2009.

[2] Intel HEX file: http://pages.interlog.com/~speff/usefulinfo/Hexfrmt.pdf.

[3] Toyota: http://developers.slashdot.org/story/10/03/13/1611248/Toyota-acceleration-and-Embedded-System-Bugs.

[4] JTAG: http://en.wikipedia.org/wiki/Joint_Test_Action_Group.

[5] 8051 series: http://www.eg3.com/8051.htm.

[6] ARM: http://www.arm.com.

[7] PIC overview: http://en.wikipedia.org/wiki/PIC_microcontroller.

[8] JAL: http://www.voti.nl/jal/index.html; JALv2: http://www.casadeyork.com/jalv2/.

[9] AVR Atmel: http://www.atmel.com/products/AVR/.

[10] Arduino: http://www.arduino.cc.

[11] JeeNode: http://news.jeelabs.org/2009/03/05/jeenode-v2-pcb/.

[12] StickDuino: http://spiffie.org/kits/stickduino/start.shtml.

[13] Processing: http://processing.org/; http://hardware.processing.org/.

[14] avr-gcc: http://www.symbolx.org/robotics/107-arduinoavr-command-line-dev-environment.

[15] AVRDUDE: http://www.ladyada.net/learn/avr/avrdude.html.

[16] STK500 Protocol: http://www.atmel.com/dyn/resources/prod_documents/doc2525.pdf.

[17] Concurrency.cc: http://concurrency.cc/.

[18] Arduino and Two-Wire Interface: http://www.nearfuturelaboratory.com/2007/01/11/arduino-and-twi/.

[19] LCD Library: http://www.arduino.cc/playground/Code/LCD4BitLibrary.

[20] Ethernet Shield: http://www.arduino.cc/en/Main/ArduinoEthernetShield.

[21] Arduino in Vancouver: http://vimeo.com/9821419.