Embedded Systems/Embedded Systems Introduction

Embedded Technology is now in its prime and the wealth of knowledge available is mindblowing. However, most embedded systems engineers have a common complaint. There are no comprehensive resources available over the internet which deal with the various design and implementation issues of this technology. Intellectual property regulations of many corporations are partly to blame for this and also the tendency to keep technical know-how within a restricted group of researchers.

Before embarking on the rest of this book, it is important first to cover exactly what embedded systems are, and how they are used. This wikibook will attempt to cover a large number of topics, some of which apply only to embedded systems, but some of which will apply to nearly all computers (embedded or otherwise). As such, there is a chance that some of the material from this book will overlap with material from other wikibooks that are focused on topics such as low-level computing, assembly language, computer architecture, etc. But we will first start with the basics, and attempt to answer some questions before the book actually begins.

What is an Embedded Computer?

The first question that needs to be asked, is "What exactly is an embedded computer?" To be fair, however, it is much easier to answer the question of what an embedded computer is not, than to try and describe all the many things that an embedded computer can be. An embedded computer is frequently a computer that is implemented for a particular purpose. In contrast, an average PC computer usually serves a number of purposes: checking email, surfing the internet, listening to music, word processing, etc... However, embedded systems usually only have a single task, or a very small number of related tasks that they are programmed to perform.

Every home has several examples of embedded computers. Any appliance that has a digital clock, for instance, has a small embedded microcontroller that performs no other task than to display the clock. Modern cars have embedded computers onboard that control such things as ignition timing and anti-lock brakes using input from a number of different sensors.

Embedded computers rarely have a generic interface, however. Even if embedded systems have a keypad and an LCD display, they are rarely capable of using many different types of input or output. An example of an embedded system with I/O capability is a security alarm with an LCD status display, and a keypad for entering a password.

In general, an Embedded System:

- *Is a system built to perform its duty, completely or partially independent of human intervention.*
- *Is specially designed to perform a few tasks in the most efficient way.*
- *Interacts with physical elements in our environment, viz. controlling and driving a motor, sensing temperature, etc.*

An embedded system can be defined as a control system or computer system designed to perform a specific task. Common examples of embedded systems include MP3 players, navigation systems on aircraft and intruder alarm systems. An embedded system can also be defined as a single purpose computer.

Most embedded systems are time critical applications meaning that the embedded system is working in an environment where timing is very important: the results of an operation are only relevant if they take place in a specific time frame. An autopilot in an aircraft is a time critical embedded system. If the autopilot detects that the plane for some reason is going into a stall then it should take steps to correct this within milliseconds or there would be catastrophic results.

What are Embedded Systems Used For?

The uses of embedded systems are virtually limitless, because every day new products are introduced to the market that utilize embedded computers in novel ways. In recent years, hardware such as microprocessors, microcontrollers, and FPGA chips have become much cheaper. So when implementing a new form of control, it's wiser to just buy the generic chip and write your own custom software for it. Producing a custom-made chip to handle a particular task or set of tasks costs far more time and money. Many embedded computers even come with extensive libraries, so that "writing your own software" becomes a very trivial task indeed.

From an implementation viewpoint, there is a major difference between a computer and an embedded system. Embedded systems are often required to provide ***Real-Time response***. A ***Real-Time system*** is defined as a system whose correctness depends on the timeliness of its response. Examples of such systems are flight control systems of an aircraft, sensor systems in nuclear reactors and power plants. For these systems, delay in response is a fatal error. A more relaxed version of ***Real-Time Systems***, is the one where timely response with small delays is acceptable. Example of such a system would be the Scheduling Display System on the railway platforms. In technical terminology, ***Real-Time Systems*** can be classified as:

- ***Hard Real-Time Systems*** - systems with severe constraints on the timeliness of the response.
- ***Soft Real-Time Systems*** - systems which tolerate small variations in response times.
- ***Hybrid Real-Time Systems*** - systems which exhibit both hard and soft constraints on its performance.

What are Some Downfalls of Embedded Computers?

Embedded computers may be economical, but they are often prone to some very specific problems. A PC computer may ship with a glitch in the software, and once discovered, a software patch can often be shipped out to fix the problem. An embedded system, however, is frequently programmed once, and the software cannot be patched. Even if it is possible to patch faulty software on an embedded system, the process is frequently far too complicated for the user.

Another problem with embedded computers is that they are often installed in systems for which unreliability is not an option. For instance, the computer controlling the brakes in your car cannot be allowed to fail under any condition. The targeting computer in a missile is not allowed to fail and accidentally target friendly units. As such, many of the programming techniques used when throwing together production software cannot be used in embedded systems. Reliability must be guaranteed before the chip leaves the factory. This means that every embedded system needs to be tested and analyzed extensively.

An embedded system will have very few resources when compared to full blown computing systems like a desktop computer, the memory capacity and processing power in an embedded system is limited. It is more challenging to develop an embedded system when compared to developing an application for a desktop system as we are developing a program for a very constricted environment. Some embedded systems run a scaled down version of operating system called an RTOS (real time operating system).

Why Study Embedded Systems?

Embedded systems are playing important roles in our lives every day, even though they might not necessarily be visible. Some of the embedded systems we use every day control the menu system on television, the timer in a microwave oven, a cellphone, an MP3 player or any other device with some amount of intelligence built-in. In fact, recent poll data shows that embedded computer systems currently outnumber humans in the USA. Embedded systems is a rapidly growing industry where growth opportunities are numerous.

~~Who is This Book For?~~

~~This book is designed to accompany a course of study in computer engineering. However, this book will also be useful to any reader who is interested in computers, because this book can form the starting point for a "bottom up" learning initiative on computers. It is fundamentally easier to study small, limited, simple computers than it is to start studying the big PC behemoths that we use on a daily basis. Many topics covered in this book will be software topics as well, so this book will be the most helpful to people with at least some background in programming (especially C and Assembly languages). Having a prior knowledge of semiconductors and electric circuits will be beneficial, but will not be required.~~

~~What Will This Book Cover?~~

~~This book will focus primarily on embedded systems, but the reader needs to understand 2 simple facts:~~

1. ~~This book cannot proceed far without a general discussion of microprocessor architecture~~
2. ~~Many of the concepts discussed in this book apply equally well, if not better, to Desktop computers than to embedded computers.~~

~~In the general interests of completeness, this book will cover a number of topics that have general relevance to all computers in general. Many of the lessons in this book will even be better applied by a desktop computer programmer than by an embedded systems engineer. It might be more fair to say that this book is more about "Low Level Computing" than "Embedded Systems".~~

~~This book will, of course, cover many embedded systems topics that are irrelevant when programming desktop computers, such as cross-compilers, Real-Time Operating Systems, EEPROM storage, code compression, bit-banging serial ports, umbilical development, etc.~~

- ~~Python compilers are available for some popular microcontrollers. Pyastra[2] compiles for all Microchip PIC12, PIC14 and PIC16 microcontrollers. PyMite[3] compiles for "any device in the AVR family that has at least 64 KiB program memory and 4 KiB RAM". PyMite also targets (some) ARM microcontrollers. Notice that these embedded Python compilers typically can only compile a subset of the Python language for these devices.~~

~~further reading:~~

- ~~Robotics: Design Basics: Design software#Programming Languages~~
- ~~Embedded Systems/PIC Programming#Compilers.2C_Assemblers~~

~~References~~

~~[1] http://imaginetools.com/support/downloads/~~
~~[2] http://pyastra.sourceforge.net/~~
~~[3] http://pymite.python-hosting.com/wiki/~~

Embedded Systems/Terminology

This page will try to discuss some of the different, important terminology, and it may even contain a listing of some of the acronyms used in this book.

Types of Chips

There are a number of different types of chips that we will discuss here.

Microprocessors

These chips contain a processing core, and occasionally a few integrated peripherals. In a different sense, Microprocessors are simply CPUs found in desktops.

Microcontrollers

These chips are all-in-one computer chips. They contain a processing core, memory, and integrated peripherals. In a broader sense, a microcontroller is a CPU that is used in an embedded system.

Digital Signal Processor (DSP)

DSPs are the "best of the best" when it comes to microcontrollers. DSPs frequently run very quickly, and have immense processing power (for an embedded chip). Digital Signal Processors, and the field of Digital Signal Processing is so large and involved, that it warrants its own book -- Digital Signal Processing.

Grades of Microcontrollers

Microcontrollers can be divided up into different categories, depending on several parameters such as bus-width (8 bit, 16 bit, etc...), amount of memory, speed, and the number of I/O pins:

Low-end

Low-end chips are frequently used in simple situations, where speed and power are not a factor. Low-end chips are the cheapest of the bunch, and can usually cost *less than a dollar*, depending on the quantity in which they are purchased. Low-end chips rarely have many I/O pins (4 or 8, total), and rarely have any special capabilities. Almost all low-end chips are 8 bits or smaller.

Mid-level chips

mid-level chips are the "basic" microcontroller units. They don't suffer from the drawbacks of the low-end chips, but at the same time they are more expensive and larger. Mid-level chips are 8 bits or 16 bits wide, and frequently have a number of available I/O pins to play with. Mid-level chips may come with ADC, voltage regulators, OpAmps, etc... Mid-level chips can cost anywhere between $1 and $10 for reasonable chips.

High-end chips

High end chips are used in situations where power and speed are a must, but a conventional microprocessor board (think a computer motherboard) is too large or expensive. High-end chips will have a number of fancy features, more available memory, and a larger addressable memory range. High end chips can come in 8 bit, 16 bit, 32 bit or even 64 bit, and can cost anywhere between $10 to $100 each.

Acronyms

This will be a functional list of most of the acronyms used in this book

ADC

ADC stands for **Analog to Digital Converter**. ADCs are also written as "A/D" or "A2D" in other literature.

DAC

The exact opposite of an ADC, a DAC stands for **Digital to Analog Converter**. May also be called "D/A" or "D2A"

RAM

Random-Access Memory. RAM is the memory that a microcontroller uses to store information when the power is on. When the power goes off, RAM is erased.

ROM

Read-Only Memory, ROM is memory that can be read, but it cant be written or erased. ROM is cheaper than RAM, and it doesnt lose its information when the power is turned off.

OTP

OTP means **One-Time Programmable**. OTP chips can be programmed once, and only once, usually by a physical process or burning extra wires inside the chip. If an OTP chip is programmed incorrectly, it can't be fixed, so be careful with them.

Microprocessor Basics

Embedded Systems/Microprocessor Introduction

Effectively programming an embedded system, and implementing it reliably requires the engineer to know many of the details of the system architecture. Section 1 of the Embedded Systems book will cover some of the basics of microprocessor architecture. This information might not apply to all embedded computers, and much of it may apply to computers in general. This book can only cover some basic concepts, because the actual embedded computers available on the market are changing every day, and it is the engineer's responsibility to find out what capabilities and limitations their particular systems have.

See also

LearnElectronics:Chapter9

Embedded Systems/Embedded System Basics

Embedded systems programming is not like normal PC programming. In many ways, programming for an embedded system is like programming a PC 15 years ago. The hardware for the system is usually chosen to make the device as cheap as possible. Spending an extra dollar a unit in order to make things easier to program can cost millions. Hiring a programmer for an extra month is cheap in comparison. This means the programmer must make do with slow processors and low memory, while at the same time battling a need for efficiency not seen in most PC applications. Below is a list of issues specific to the embedded field.

Tools

Embedded development makes up a small fraction of total programming. There's also a large number of embedded architectures, unlike the PC world where 1 instruction set rules, and the Unix world where there's only 3 or 4 major ones. This means that the tools are more expensive. It also means that they're lower featured, and less developed. On a major embedded project, at some point you will almost always find a compiler bug of some sort.

Debugging tools are another issue. Since you can't always run general programs on your embedded processor, you can't always run a debugger on it. This makes fixing your program difficult. Special hardware such as JTAG ports can overcome this issue in part. However, if you stop on a breakpoint when your system is controlling real world hardware (such as a motor), permanent equipment damage can occur. As a result, people doing embedded programming quickly become masters at using serial IO channels and error message style debugging.

Resources

To save costs, embedded systems frequently have the cheapest processors that can do the job. This means your programs need to be written as efficiently as possible. When dealing with large data sets, issues like memory cache misses that never matter in PC programming can hurt you. Luckily, this won't happen too often- use reasonably efficient algorithms to start, and optimize only when necessary. Of course, normal profilers won't work well, due to the same reason debuggers don't work well. So more intuition and an understanding of your software and hardware architecture is necessary to optimize effectively.

Memory is also an issue. For the same cost savings reasons, embedded systems usually have the least memory they can get away with. That means their algorithms must be memory efficient (unlike in PC programs, you will frequently sacrifice processor time for memory, rather than the reverse). It also means you can't afford to leak memory [1]. Embedded applications generally use deterministic memory techniques and avoid the default "*new*" and "*malloc*" functions, so that leaks can be found and eliminated more easily.

Other resources programmers expect may not even exist. For example, most embedded processors do not have hardware FPUs [2] (Floating-Point Processing Unit). These resources either need to be emulated in software, or avoided altogether.

Real Time Issues

Embedded systems frequently control hardware, and must be able to respond to them in real time. Failure to do so could cause inaccuracy in measurements, or even damage hardware such as motors. This is made even more difficult by the lack of resources available. Almost all embedded systems need to be able to prioritize some tasks over others, and to be able to put off/skip low priority tasks such as UI in favor of high priority tasks like hardware control.

Fixed-Point Arithmetic

Some embedded microprocessors may have an external unit for performing floating point arithmetic(FPU), but most low-end embedded systems have no FPU. Most C compilers will provide software floating point support, but this is significantly slower than a hardware FPU. As a result, many embedded projects enforce a no floating point rule on their programmers. This is in strong contrast to PCs, where the FPU has been integrated into all the major microprocessors, and programmers take fast floating point number calculations for granted. Many DSPs also do not have an FPU and require fixed-point arithemtic to obtain acceptable performance.

A common technique used to avoid the need for floating point numbers is to change the magnitude of data stored in your variables so you can utilize fixed point mathematics. For example, if you are adding inches and only need to be accurate to the hundreth of an inch, you could store the data as hundreths rather than inches. This allows you to use normal fixed point arithmetic. This technique works so long as you know the magnitude of data you are adding ahead of time, and know the accuracy to which you need to store your data.

Synchronous and Asynchronous

Data can be transmitted either synchronously or asynchronously. **synchronous** transmissions are transmissions that are sent with a clock signal. This way the receiver knows exactly where each bit begins and ends. This way, there is less susceptability to noise and **jitter**. Also, synchronous transmissions frequently require extensive hand-shakeing between the transmitter and receiver, to ensure that all timing mechanisms are synchronized together. Conversely, **asynchronous** transmissions are sent without a clock signal, and often without much hand-shaking.

References

[1] "Technical Report on C++ Performance" (http://www.research.att.com/~bs/performanceTR.pdf) by Dave Abrahams et. al. 2003

Embedded Systems/Serial and Parallel IO

The Serial and Parallel IO page of the Embedded Systems wikibook is a stub. You can help by expanding this section.

Data Transmission

Data can be sent either serially, one bit after another through a single wire, or in parallel, multiple bits at a time, through several parallel wires. Most famously, these different paradigms are visible in the form of the common PC ports "serial port" and "parallel port". Early parallel transmission schemes often were much faster than serial schemes (more wires = more data faster), but the added cost and complexity of hardware (more wires, more complicated transmitters and receivers). Serial data transmission is much more common in new communication protocols due to a reduction in the I/O pin count, hence a reduction in cost. Common serial protocols include SPI, and I^{2}C. Surprisingly, serial transmission methods can transmit at much higher clock rates per bit transmitted, thus tending to outweigh the primary advantage of parallel transmission. Parallel transmission protocols are now mainly reserved for applications like a CPU bus or between IC devices that are physically very close to each other, usually measured in just a few centimeters. Serial protocols are used for longer distance communication systems, ranging from shared external devices like a digital camera to global networks or even interplanetary communication for space probes, however some recent CPU bus architechtures are even using serial methodologies as well.

Common embedded operating systems

In this book, we discuss these operating systems commonly used in embedded systems:

- Palm OS
- Windows CE
- MS-DOS or DOS Clones
- Linux, including RTLinux and MontaVista Linux

~~For further reading~~

~~A variety of embedded systems and RTOS are based on Linux -- see Embedded Systems/Linux for details.~~

- ~~Embedded Control Systems Design/Operating systems~~
- ~~Wikipedia:INTEGRITY: A small, message passing, hard real-time micro kernel with memory protection designed for safety critical and high security devices.~~
- ~~Wikipedia:Contiki: a small, open source, operating system developed for use on a number of smallish systems ranging from 8-bit computers to embedded microcontrollers.~~
- ~~Wikipedia: eCos (embedded Configurable operating system): an open source, royalty-free, real-time operating system intended for embedded systems and applications. ... eCos was designed for devices with memory footprints in the tens to hundreds of kilobytes, or with real-time requirements.~~
- ~~Wikipedia:DSP/BIOS: a royalty-free real-time multi-tasking kernel (mini-operating-system) created by Texas Instruments.~~
- ~~Wikipedia:QNX~~
- ~~Wikipedia:VxWorks: A small footprint, scalable, high-performance RTOS~~
- ~~Wikipedia:Windows CE~~
- ~~Wikipedia:Palm OS~~
- ~~"pico]OS" [2] has been ported to the Atmel AVR, the ARM, and the 80x86~~
- ~~Wikipedia: OSEK is not an OS, but an open standard for automotive real-time operating systems.~~
- ~~MaRTE OS - Minimal Real-Time Operating System for Embedded Applications [3] *(Is this related to Wikipedia: MARTE ?)*~~
- ~~Wikipedia: TinyOS is an open-source operating system designed for wireless embedded sensor networks ("networked sensors").~~
- ~~Wikipedia: ChibiOS/RT is an open-source real-time operating system that supports LPC214x, AT91SAM7X, STM32F103x and AVRmega processors.~~
- ~~Wikipedia: Fusion RTOS is a license-free embedded operating system that supports ARM, Analog Devices Blackfin, Motorola StarCore and Motorola DSP 56800E.~~
- ~~Wikipedia: FreeRTOS is an open-source embedded operating system kernel that supports ARM, Atmel AVR, AVR32, HCS12, MicroBlaze, MSP430, PIC18, dsPIC, Renesas H8/S, x86, 8052 processors. FreeRTOS can be configured for both preemptive or cooperative operation. FreeRTOS, SafeRTOS, and OpenRTOS are based on the same code base.~~
- ~~Wikipedia: RTEMS (Real-Time Executive for Multiprocessor Systems) is a free open source real-time operating system (RTOS) designed for embedded systems.~~
- ~~Wikipedia: MicroC/OS-II is an embedded RTOS intended for safety critical embedded systems such as aviation, medical systems and nuclear installations; it supports a wide variety of embedded processors.~~

Particular Microprocessor Families

Embedded Systems/Particular Microprocessors

This module of *Embedded Systems* is a very brief review of the most popular microprocessor families used in embedded systems. We will go into more detail in the next few modules. Each one of these microprocessor families has an entire module dedicated to that family of processors.

The microprocessor families we will discuss are:

- 8051 Microcontroller 8 bit
- Atmel AVR 8 bit
- Atmel AVR32
- Microchip PIC Microcontroller (this family includes the code-compatible Parallax SX chips) 8 bit
- Microchip dsPIC microcontroller 16 bit: review: Circuit Cellar: "Are You Up for 16 Bits? A look at Microchip's Family of 16-Bit Microcontrollers [1] by Jeff Bachiochi 2007; example application: µWatch D-I-Y open source scientific calculator watch [2]
- Freescale Microcontrollers
- The Zilog Z8 Series (Z8, Z8encore, Z8XP)
- Cypress PSoC Microcontroller
- Texas Instruments MSP430 microcontrollers 16 bit
- ARM Microprocessors (ARM) (this family includes the Philips LPC210x ARM microcontrollers, the ~~discontinued Intel w:StrongARM~~, Atmel AT91RM9200, and the Intel XScale microprocessors)
- x86 microprocessors

~~brief selection guide~~

~~For many embedded systems, any of these microcontrollers would be more than adequate.~~

- ~~TI MSP430 has the lowest power consumption. In sleep mode, 0.3 µW = 3 V * 0.1 µA. Some chips in 2xx and 4xx series include 12-bit DACs.~~
- ~~The Cypress PSoC has more than one true analog output. Using sleep mode, power consumption as low as 21 µW = 5 V * 4.2 µA[3]. (You can get analog output from the other chips by using an external ADC, or by faking it with a PWM output and some low-pass filtering.) Most Cypress PSoC microcontrollers come in both DIP and SMT versions.~~
- ~~Many of these series include microcontrollers with integrated 10 bit ADCs, but Atmel AVR 8 bit series (as of early 2006) had the lowest-price chip that included such an ADC, as well as another chip with the lowest cost/ADC. Most Atmel AVR 8 bit microcontrollers come in both DIP and SMT versions.~~
- ~~If you need a very tiny chip, the Atmel AVR, PIC, and Freescale microcontroller lines all include tiny 8-pin SOIC microprocessors.~~

Embedded Systems/8051 Microcontroller

The Intel 8051 microcontroller is one of the most popular general purpose microcontrollers in use today. The success of the Intel 8051 spawned a number of clones which are collectively refered to as the MCS-51 family of microcontrollers, which includes chips from vendors such as Atmel, Philips, Infineon, and Texas Instruments.

About the 8051

The Intel 8051 is an 8-bit microcontroller which means that most available operations are limited to 8 bits. There are 3 basic "sizes" of the 8051: Short, Standard, and Extended. The Short and Standard chips are often available in DIP form, but the Extended 8051 models often have a different form factor, and are not "drop-in compatable". All these things are called 8051 because they can all be programmed using 8051 assembly language, and they all share certain features (although the different models all have their own special features).

Some of the features that have made the 8051 popular are:

- 8-bit data bus
- 16-bit address bus
- 32 general purpose registers each of 8 bits
- 16 bit timers (usually 2, but may have more, or less).
- 3 internal and 2 external interrupts.
- Bit as well as byte addressable RAM area of 16 bytes.
- Four 8-bit ports, (short models have two 8-bit ports).
- 16-bit program counter and data pointer

8051 models may also have a number of special, model-specific features, such as UARTs, ADC, OpAmps, etc...