

Embedded System Primer

Version 1.0

Date: 8 October 2001-10-11

© 2001 by GenerExe

www.generexe.com

SUMMARY

Embedded system software development practices are in many ways different from other software engineering disciplines. This paper aims at being an introduction to software engineers from other disciplines. Besides presenting the technical differences and problems, it also presents some cultural elements, personal attitude and requirements shared by successful embedded software engineers.

1) INTRODUCTION

Some facts:

- 1) *Embedded Systems include all computer-systems which you cannot see or identify when looking at an appliance/product.*
- 2) *Computers in Embedded systems greatly outnumber all other computers on desks or in businesses.*

Embedded systems often share the following attributes:

- They are tightly connected to the outside world via sensors and actuators (i.e. motors, relais, switches). The outside world is not as clean as the digital world inside your software.

(Noise, disturbance, debouncing of switches).

- They often monitor and control processes.
- Data logging and transformations is less important
- They have a minimal user interface.
- They start on powerup and HAVE TO REMAIN operational until power is removed. (Reliability is important; A VCR, which crashes while you are watching a movie is not acceptable);
- The environment in which the system is going to operate and the production method/run pose requirements on embedded systems, regarding:
 - Price of parts
 - In-circuit programmability
 - Environmental requirements: temperature ranges, vibrations, humidity (close to the “action”)

In the rest of this document I will briefly describe the issues Software Engineers have to deal with when shifting towards Embedded Systems. I assume that the reader has some basic programming experience and knowledge of how a computer works.

2) HARDWARE ENVIRONMENTS

2.1) ELECTRONICS

The target-hardware often is customized for the tasks it has to perform in a certain device. As a result, the variety in hardware, interfaces and drivers is very high. (See appendix B for an overview of the different markets, where you can encounter embedded systems). Also on smaller systems there might not be a central operating system, providing general services to an application. As a matter of fact, most of the times your application is the **ONLY** software running on the controller.

If you are lucky your code is stored on a flash-card and loaded into program memory (RAM) at startup by some ROM-bootloader. However in many cases, your software is statically linked and stored on fixed addresses in some kind of non-volatile ROM. There is no boot-loader at all. Your main-function is the first code to run and it is your responsibility to configure the controller and its peripherals.

The hardware often interface to the outside world via several interfaces. These interfaces can be very noisy. I.e. pulling a switch actually results in a bouncing digital signal on a millisecond time-scale. The inputs of your systems are not constrained to the keys on a keyboard, but can be any (analog) value, the sensor is able to expose. Signal conditioning often needs to be performed in software, before the information can be interpreted. (I.e. debouncing of a switch). Hardware noise

(radiation from your powersupply or a nearby radio), can lead to seemingly erratic behavior, indicating a hardware problem instead of software problem. (Unfortunately, the hardware design has to be frozen before the software development, sometimes forcing you to “program” a solution for a problem, which essentially is caused by faulty hardware design).

Since the hardware environment is often in a prototype-stage when software development starts, it is recommended to check your wiring and cables regularly! Don't hesitate to ask someone to explain the use of a multi-meter to measure a cable, connection or discrete output-pin.

2.2) PROCESSORS

The hardware used in embedded systems, ranges from single-chip small/8pin 8bit RISC-controllers to 100pin+ 32bit CISC-processors, with supporting chipsets.

The lower end controllers are characterized by:

- having no operating system, requiring you to design your own startup-code and scheduling-executive;
- relatively small amounts of RAM (for data/variables);
- using non-volatile ROMs in which the software code has been burned;
- are being used in customized different environments every time, requiring you to write custom low level drivers every time.

The higher end controllers are hardware-wise more similar to desktop-machines, except for probably having some kind of solid-state disk-on-chip file-system instead of mechanical disks and tapes. Memory resources are more abundant and third-party operating systems and drivers can be purchased with APIs (APplication Interfaces) which will look familiar to any Windows-programmer.

In appendix A, you will find a more detailed description of the different ranges of Embedded Systems you might encounter.

3) SOFTWARE TOOLS: COMPILER, DEBUG, TEST AND MAINTENANCE

Embedded software is often developed with a cross-compiler. These compilers run on a regular desktop system but compile code for the target-hardware. More often than not, the languages will be assembly and C on lower-end controllers and C/C++ on higher-end controllers. Remember that some lower-end controllers can store only 1024 bytes of program code and have less than 64bytes of RAM, while for the higher end controllers these numbers may run into the MegaBytes-range. These differences result in a totally different kind of challenge for the Software Engineers involved.

C is used because:

- optimized C-code approaches compactness/efficiency of manually crafted assembly code;

- the ansi-C language is standardized, allowing re-use of code and cross-platform portability.

C++ has the advantage that its OO-concepts promote maintainability, thus allow for more complex systems to be implemented. Care should be taken to be aware of some of C++ its performance issues (i.e. runtime binding). Note that you can do the same dirty things in C++ as you can in C. Just “using” C++ gives NO guarantee for higher quality.

After compilation there are several possibilities to test the code. One way or another the binary code-files have to be trusted to some kind of execution-environment where it can be tested. Some of the options are:

- use of a simulator on the desktop machine or some remote server, with which you can step through a significant portion of the functionality in order to increase confidence in your work, before execution on the real target;
- the lucky few with a bigger wallet have an emulator, which fits in the CPU-socket of the target-controller board. The emulator connects to your desktop machine and allows you to use a source-level debugger to debug your code, while being run on the target. Sometimes this approach is augmented with the use of EEPROM-emulators, to prevent the need of re-burning your program-code into ROM each time you made a change/fix. In these cases, the debug-process is very similar as with traditional

debugging in an IDE (Integrated Development Environment) for desktop software development. The big advantage is that you are working with the real hardware and can test the actual interfaces of that hardware;

- if an emulator is not available, you are often forced to include a test-interface into your code. The larger platforms may include support for such an interface already. For smaller systems you might need to implement a simple “monitor-like” function, which allows you to communicate with the target device over a simple RS232 connection or a TCP/IP cable. The monitor-program functions range from the ability to load/start programs, inspect memory and send out debug-texts to complete debug-interfaces which hook up to your debugger and allow you to set breakpoints and step through your code. However, you need to be aware of the performance impact of the monitor-program you include with your software (depending on whether your final release code will include the monitor or not).

For future problem solving many embedded systems reserve some non-volatile writable memory to serve as a log where any errors and performance statistics may be stored. When a device is in maintenance, these logs may help to diagnose a problem.

4) EMBEDDED SOFTWARE DESIGN

4.1) RELIABILITY

Many embedded systems run as long as power is applied. Since embedded systems often have a minimum user interface and are “embedded” in some bigger device, no user/operator can restart or interfere with the operation of the software in any other way than restarting the system by powering the system down.

The software needs to be able to determine itself if something is wrong and take appropriate action. Most times this means, an error-message followed by shutdown or an automated power-cycle (reset) of the controller. In rare cases, degraded functionality may be required. A **watchdog timer** is one of the most common recovery mechanisms used. This independent hardware times resets your target’s controller if it is not cleared periodically by your software. The idea is that the missing clear-signal is an indication that your software crashed.

Another common function is the **built-in tests**, which are performed at least once during startup, but sometimes are performed periodically. Typical tasks done during a built-in test are:

- verifying memory (write/read tests of RAM, checking checksums/CRCs of memory ranges in ROM);
- test communication ports if local loopback signals are available;
- many industry chipsets include built-in tests, which can be invoked from software.

4.2) PERFORMANCE

4.2.1) REALTIME and PRE-EMPTIVE

The term “**Realtime**” is often used in Embedded System environments. *Realtime performance means that a required response to an event occurs guaranteed (always) within a required period of time.* Sometimes the term latency (delay) is used to indicate the maximum time it might take for an external event to be handled by an appropriate piece of code.

As a result the scheduling/interrupt-handling mechanism of your system’s operating system or executive needs special attention during design. In these situations, a flag or event CANNOT be buffered in some kind of message-queue until the appropriate handler gets the opportunity to work on it, by the grace of another task giving up the processor (i.e. as with round-robin time-slicing, which is often used in office/multi-user environments).

Instead high priority events **pre-empt**, the running task to allow the appropriate handler to generate the response immediately. On lower-end systems, this behavior is accomplished by interrupt-routines. On higher-end systems the operating system’s or executive’s task-scheduler uses task-priorities in combination with pre-emption to give the handlers tasks the processor as soon as possible.

4.2.2) ALGORITHMS AND LIBRARIES

If you are using third party or compiler-provided libraries make sure that you are aware of the following:

- execution-cost of the function;
- code-space required by the library (using one function may link the complete library into your application!);
- is the library multi-thread safe (re-entrant) if your system is using multiple threads/task.

For example, the use of a popular function like “printf”, makes you pay a high price in resources (both in execution time and code-space). Using a “puts” is many times more efficient.

4.3) RESOURCE MANAGEMENT

Don’t use the heap! Use arrays instead or otherwise pre-allocate all the memory you will ever use during startup. This way you reduce the likelihood of memory-leaks (when you forget to free an unused memory block) and don’t waste precious execution time managing the heap. Memory-leaks, which may go unnoticed in a desktop-application, may cause an embedded system to suddenly crash after a week, since these systems may run uninterruptedly for very long periods of time, during which a desktop machine might have been restarted already.

5) CONCLUSIONS

5.1) GUIDELINES

Below follows a list of some of the guidelines discussed in the previous sections:

- Remember: reliability and performance are the key aspects!
- Simple is beautiful and improves reliability. (Remember that most applications, once “embedded” in their target-environment are not very easily accessed/interrogated/diagnosed);
- Reserve a serial link for debugging/testing;
- Don’t use the heap! Use arrays instead or otherwise pre-allocate all the memory you will ever use during startup;
- Use a watchdog timer to force reboot on software failure;
- Include an error-log, which can be accessed during maintenance to help diagnosing problems.

You will also benefit greatly from some low level processor and operating system knowledge, like:

- event-driven systems;
- realtime performance,;
- schedulers: threads, timers, pre-emption, priorities and interrupts;
- drivers: memory mapped I/O, resource allocation, deadlock, etc;
- inter-process/task/thread communication: semaphores, mailboxes.

5.2) SOFTWARE ENGINEERING CULTURE

Perhaps an important feature need for embedded systems development is: ATTITUDE.

Since many embedded systems are closely connected to the electronics of a product and for the smaller systems, traditional software development concepts do not apply very well, traditionally a lot of programming is done by the electronics engineers, which designed the hardware.

As long as the systems are small, this may work very well. A big portion of microcontrollers just replaces logic, which formerly was implemented in hardware logic (digital and/or analog electronic components). The microcontrollers just simplify hardware design and make the system more dynamic, allowing for more elaborate changes, later in the design cycle. More complex software systems tend to grow in complexity, beyond the complexity of the hardware itself and require a different set of skills.

If you wish, software can be viewed as highly dynamic and configurable hardware. The first computers -general purpose calculators- were exactly this! They were programmed by setting a number of switches in a certain position. The purpose of the machine is the same as the all-hardware version of the same device, only now we could reuse the same hardware for multiple purposes.

Why then, does it seem that we are way more lenient towards the way software is

developed and tested then we are when hardware development is involved? The answer is: the software can be changed relatively “fast” at a later time, while hardware changes may involve a new production cycle or at least a soldering iron. This “fast” (in time) is certainly true if you just consider replacing the software in a device, but the catch is that the opposite is true when you consider the responsibilities of the functionality trusted to the software!

The functions now performed in software are of the same importance as they were, when they were still hardwired in hardware. Furthermore, today, the dynamics/complexity of software far outrun the dynamics going on in the hardware it executes on. Increasing size and speed of controllers result in more and more tasks being implemented in software. So, software complexity increased way beyond just “replacing hardware”. However, the thoroughness adopted in hardware design often seems to be a good distance away when you walk into the software lab. Sure we know some tricks to keep things relatively stable, but you will still feel the attitude of: “we can always fix it at a later time!”, sometimes called “trial and fix”. The market promotes this attitude, because time-to-markets reduce all the time.

The ease at which we accept that a product is faulty (contains “bugs”) would not be acceptable in any other engineering discipline. In my opinion, most software development is still the “trial and fix” stage (CMM level 1) and has NOTHING to do with engineering, nor does it have

anything to do with science or math. It is just sculpting functionality as you go, dealing with problems as they come and/or fix them later. This attitude does NOT work in embedded system development!!!

I realize that the last 10 years a lot has improved, but the fact remains that as an engineering discipline “Software Engineering” is still in its infancy. And the nature of embedded systems, requires more “engineering”, then most programmers in other disciplines are accustomed to.

To summarize, engineers working on embedded systems will need to:

- increase their thoroughness (be an engineer!);
- always know what they are doing;
- not try, but ask or look it up in a reference-book or on the internet;
- study risks before attacking them with a keyboard;
- anticipate changing requirements;
- document not only for the sake of “thinking before typing”, but also to support future maintenance and communication with team-members and customers in larger more complex projects;
- not be afraid for electronics and measurement tools;

Appendix A: Differences in Embedded Systems

The table below gives an overview of third party products, systems and tools for different kind of embedded systems.

	Low End (microcontrollers)	Medium Sized	High End larger systems
Hardware description	<ul style="list-style-type: none"> • 4kByte or less RAM and ROM • 8bit RISC (harvard architecture) 20mHz or less • Simple interrupt system • 20 pin or less single chip packages 	<ul style="list-style-type: none"> • 1kByte RAM and around 64kByte of ROM • 8bit or 16bit CISC architecture (20mHz or less) • Elaborate interrupt system • 100 pin or less microcontroller, supported by external chip-logic (memory and I/O features) 	<ul style="list-style-type: none"> • 64kByte or more RAM and 256kByte or more ROM • 16 or 32bit RISC or CISC architecture; 20mHz or higher up to several 100mHz • 100+ pin packages in a highly integrated chip-set (5 or less) often optimized for a particular application (i.e. cellphone or general purpose data-acquisition system) • OR generic industry-standard external bus-system which allows use of third-party I/O boards. (I.e. PCI, VME, VXI).
Operating Systems	None. The application needs its own startup/boot	None or timer-interrupt based custom multitasking	Third-party supplied Realtime Operating System:

	code and includes time/resource allocation functionality to perform its work in timely manner.	scheduler. All peripherals and I/O often are still customly written, since hardware-design often customized to applications	<ul style="list-style-type: none"> • VxWorks • pSOS • Windows-CE • Various Embedded Linux Distributions Often conform to POSIX compatible API, supplying built-in support for multi-processing/threading, interprocess synchronization and communication and an elaborate pool of industry standard I/O drivers.
Languages	Assembler-code and sometimes C.	C language and assembly for driver-like peripheral/IO	C and C++
Development Tools	<ul style="list-style-type: none"> • Simulator of microcontroller • Assembler software • Optimizing C-compiler • Hardware emulator 	<ul style="list-style-type: none"> • Prototype generic platform with target-CPU and console-access and/or debug-monitor from remote workstation (via serial port or TCP/IP connection) • Assembler and C-compiler 	<ul style="list-style-type: none"> • Workstation simulators • Extensive resources on the target are dedicated to helping debugging (higher speed communication with workstations, stepping through code at source-level) • Optimizing compilers
Background Software Engineers	Electronics engineer	Technical Computer Science engineer with knowledge of electronics	Workstation C/C++ programmers who know how to increase reliability and are aware of

			performance issues
--	--	--	--------------------

Appendix B: Embedded System markets

Market	Product Examples
Consumer Electronics Entertainment equipment (audio, video, game consoles) Kitchen equipment Heating Home computers Toys	Stereo, Television, Gameboy Refrigerator, Laundry-machine Thermostat, boiler control Personal computer (PC) Robot-dogs
Measurement Systems	Digital Multi Meters TestFrame like automated test systems
Aerospace and transportation	Control systems, display units, in-flight entertainment
Industry Process control	PLCs,
Telecommunication	Telephone systems
Computer peripherals	Printers, network-switches, etc.
Defense equipment	Communication systems
Security systems	Sensory systems, X10 parts
Medical systems	Measurement, scanning and control equipment
Vending machines/retail/MKB	Public Information Access Points/terminals; Vending: drinks/ice-cream etc. Horeca cash-register/ordering systems
Handheld organizers	PalmOS, Symbian (Smartphones)
Gaming	Handheld or TV consolders, Casino games.