ENCE464

Embedded Software & Advanced Computing

Lecture 1: Introduction

ENCE464

The team:

Steve Weddell (Course coordinator & Lecturer in Term-3)
steve.weddell@canterbury.ac.nz (office: Link Building, Room 510)
Le Yang (Course Lecturer in Term-3)
Le.Yang@canterbury.ac.nz (office: Link Building, Room 305)
Mike Hayes (Lecturer in Term-4)
michael.hayes@elec.canterbury.ac.nz (office: Link Building, Room 504)
Dave van Leeuwen (Support, git accounts, etc.)
dave@elec.canterbury.ac.nz (office: ECE Wing Building, Room 217)

Assessment:

Individual Project (Term-3): 30% (Demo & Code: 22 August; Final report and critique is due 25 August.)

Team Project (Term-4): 20% (Details TBA.)

Final Exam : 50% (2.5 hours)

ENCE464: Overview & Lecture Schedule over Term-3*

ENCE464 is a relatively new course that combines software engineering practice for embedded systems (old ENCE463) with advanced computer architectures and memory systems (old ENCE462). State machines form a unifying mechanism to understand hardware and software. Software design procedures and methodologies are used to develop reliable coding practices used on high-performance multi-core systems with real-time constraints. Testing and debugging on high-performance concurrent systems, where inter-task communication support is required, are analyzed using industry standard metrics and test platforms.

LY				
SJW	**ENCE464 Lecture V.2.0, Tutorial, & Demo Schedule, 2019**			
MPH		**(approx. only)**	14/07/19	
Week	**Lecture b (Mon.)**	**Lecture C (Tue.)**	**Lecture A (Wed.)**	**Comments/Tutorials/Labs**
13	Intro - embedded SW eng.	Specification & Arch.	FreeRTOS Intro	Project intro (Thu) ESL
14	FreeRTOS Basics (1 of 2)	FreeRTOS Adv. (2 of 2)	scheduling methodologies	FreeRTOS Tut. (venue TBA)
15	Make it scale (intro)	Object orientated design	Object Patterns	"Make it Scale" series
16	Testing (intro)	Testing (Cont.)	Charts (UML & State)	Lab/Tutorial in ESL
17	Make it Right (Intro)	Concurrency (Ch.2)	Concurrency (Ch.3)	Lab/Proj demo brief in ESL
18	Concurrency (Ch.3 cont.)	Mutex-OPG Ex (Ch.4)	Tutorial	Demos: Thu (Group.)
	*** 2 wk term break ***	*** 2 wk term break ***	*** 2 wk term break ***	
19	MPH - TBA	MPH - TBA	MPH - TBA	

*Subject to change

Previous courses covered the basics of how to program

ENCE361 covered developing embedded software:

- The embedded tool-chain
- The basics of embedded debugging and testing
- The concept of data abstraction
- How to interact with a variety of peripherals at a low level
- Interrupts and latency
- Simple real-time operating systems
- Simple concurrency (+ from ENEL373 and ENCE360)

This course

This course goes beyond simply knowing how to program. In Term-3 we will focus on software engineering for embedded systems. Software engineering in general is about the application of a *systematic*, *disciplined*, *quantifiable* approach to the *development*, *operation* and *maintenance* of software.

In Term-4 you will learn about advanced computer architectures, in addition to metrics with which to compare and measure performance.

FreeRTOS

"FreeRTOS is a market leading real time operating system (or RTOS) from Real Time Engineers Ltd. that supports 34 architectures and receives 103000 downloads a year. It is professionally developed, strictly quality controlled, robust, supported, and free to use in commercial products without any requirement to expose your proprietary source code. Why would you choose anything else?" - www.freertos.org

Features:
FreeRTOS is designed to be simple and easy to use: Only 3 source files that are common to all RTOS ports, and one microcontroller specific source file are required, and its API is designed to be simple and intuitive.

Supports:
Tasks and co-routines
Tasks are implemented as C functions
Fixed priority **pre-emptive scheduling** (based on ticks from SysTick)
Optional cooperative scheduling
Queues
Binary and counting **semaphores**
Mutexes and recursive **mutexes**
Stack overflow detection

Why do we care about ***engineering*** embedded software?

The lawyers are coming!

The quality of a lot of embedded software is abysmal. And lawyers are on to it. If you don't want your source code to show up in court, you better get your act together.

By Michael Barr Embedded.com (2009)

Listing 1 A single line of unintelligible mystery code.

```
y = (x + 305)/146097 * 400 + (x + 305) % 146097 / 36524 * 100 * (x + 305)
    % 146097 % 36524 / 1461 * 4 + (305) % 146097 % 36524 % 1461 / 365;
```

Why do we care about ***engineering*** embedded software?

Listing 1 A single line of uninteligible mystery code.

```
y = (x + 305)/146097 * 400 + (x + 305) % 146097 / 36524 * 100 * (x + 305)
    % 146097 % 36524 / 1461 * 4 + (305) % 146097 % 36524 % 1461 / 365;
```

"The original listing had no comments on or around this line to help. I eventually learned that this code computes the year, with accounting for extra days in leap years, given the number of days since a known reference date (such as January 1, 1970). But I note that we still don't know if it works in all cases, despite it being present in an FDA-regulated medical device."

Michael Barr Embedded.com (2009)

Why do we care about ***engineering*** embedded software?

"Consider the case of the *Alcotest 7110*. After a two-year legal fight, seven defendants in New Jersey drunk driving cases successfully won the right to have their experts review the source code for the Alcotest firmware. Expert reports evaluating the quality of the firmware source code were produced:

- Of the available 12 bits of ADC precision, just 4 bits (most-significant) are used in the actual calculation. This sorts each raw blood-alcohol reading into one of 16 buckets.
- Out of range A/D readings are forced to the high or low limit. This must happen with at least 32 consecutive readings before any flags are raised.
- There is no feedback mechanism for the software to ensure that actuated devices, such as an air pump and infrared sensor, are actually on or off when they are supposed to be.
- The software first averages the initial two readings. It then averages the third reading with that average, then the fourth reading is averaged in, and so on. Thus the final reading has a weight of 0.5 in the "average", the one before that 0.25, and so on.
- Out of range averages are forced to the high or low limit, too.
- Static analysis with `lint` produced over 19,000 warnings about the code (that's about three errors for every five lines of source code).

What would you infer about the reliability of a defendant's blood-alcohol reading if you were on that jury?"

Michael Barr Embedded.com (2009)

What makes an embedded system different from a desktop system?

- Interaction with the physical world implies a need for meeting **real-time deadlines**. Total data throughput is less important than meeting deadlines. More importantly, we need to understand the physical processes that the software is interacting with, and **how the physical processes and the software will work together**.

- **Interaction with the physical world** is often through application-specific sensors or actuators, rather than a standard mouse, keyboard, display or network port.

- The software must run on a non-desktop processor, often on a custom PCB. Insight into the internal state of the software is more difficult to come by. **Standard operating system services may not be available** (or appropriate).

- The application may **require high reliability**. Systems often must work without fail or with minimal interference. **What can go wrong**? How can we think through the possibilities? How can we recover?

- The **software must interact simultaneously with multiple external processes that may produce continuous signals or sporadic events**. This implies that some kind of concurrency will be required.

- **Embedded systems are generally resource-constrained and cost-constrained**.

What we're trying to avoid:

Avoiding the Top 43 Embedded Software Risks
Philip Koopman (Carnegie Mellon University)
Embedded Systems Conference, Silicon Valley, 2011

Koopman's identified software risk areas included:

#3. **No written requirements**
#4. Requirements omit extra-functional aspects
#5. **Requirements with poor measurability**
#6. **No defined software architecture**
#7. **Poor code modularity**
#8. Too many global variables
#10. Design skipped or is created after code is written
#11. **Flowcharts are used in place of statecharts**
#15. **No real time schedule analysis**
#19. **No test plan**
#31. High requirements churn
#32. No version control

Avoiding the Top 43 Embedded Software Risks
Philip Koopman (Carnegie Mellon University)
Embedded Systems Conference, Silicon Valley, 2011

Of Koopman's 43 risks, the following are particularly relevant:

2. Define requirements in a measurable, traceable way. If you can't measure a requirement, you don't know if you've met it.
3. Document your software architecture. Make your code modular and avoid global variables.
7. Do real time scheduling analysis and use a 3rd party RTOS if you are doing preemptive task switching.
8. Pay attention to concurrency to avoid tricky timing bugs. Home-brew methods are risky.
11. Ensure that problems found both in test and at runtime are identified and tracked.
12. Explicitly specify and plan for performance, dependability, security and safety up-front. Most embedded systems involve all these and it is painful to try to add them at the end of the project.
16. Leave plenty of slack resources. Use as little assembly language as possible (zero assembly code is a good amount).

What things matter in embedded software development?

Jack's Top Ten (Identifying the 10 most important issues in embedded software development)

Jack Ganssle *(jack@ganssle.com)*

December 1, 2006

"Embedded systems defy conventional test techniques. How do you build automatic tests for a system which has buttons some human must push and an LCD someone has to watch? A small number of companies use virtualization. Some build test harnesses to simulate I/O. But any rigorous test program is expensive.

"Programming is a human process subject to human imperfections. The usual tools, which are usually not used, can capture and correct all sorts of unexpected circumstances. These tools include checking pointer values; range checking data passed to functions; using asserts and exception handlers, and checking outputs (I have a collection of amusing pictures of embedded systems displaying insane results, like an outdoor thermometer showing 505 degrees, and a parking meter demanding $8 million in quarters).

"For very good reasons of efficiency, C does not check, well, pretty much anything. It's up to us to add those [checks] that are needed."

Final words (in this lecture anyway!)

It's worth emphasizing (and constantly bearing in mind) that one of the essential features of software, and one of its key contributors of value in a system design, is **that it is soft**. That is, software can be changed and adapted to meet changing customer needs or to respond to our evolving understanding of the problem space. How can we design our systems to best take advantage of this fact?

Outside of the books and articles cited in this course, one of the best references for embedded software engineers is the articles at http://www.embedded.com/