edX (https://www.edx.org)

UTAustinX: UT.6.01x Embedded Systems - Shape the World

KarenWest (/dashboard) ▼

Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) **Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)**

Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) **Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)**

Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)

Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

Embedded Systems Community (/courses/UTAustinX/UT.6.01x/1T2014/e3df91316c544d3e8e21944fde3ed46c/)

In this next video we present an introduction to **modular design**, using the Lab 15 hand-held game as an example.

VIDEO 15.2 MODULAR DESIGN

C15 2 Game design

Help

21:31 / 21:31 **1.0x**

DR. RAMESH YERRABALLI: So in this module,

we're going to look at the game from a modular design standpoint.

What we mean by that is we will design our system using three important tools.

The first one will be a call graph.

The second will be a data flow graph.

And the third will be a low level detail of flow charts.

DR. JONATHAN VALVANO: Professor Yerraballi,

why is modular design important?

DR. RAMESH YERRABALLI: So modular design allows

us to manage the complexity of a large project.

This project is going to be one of the largest

that you will handle in this class.

So what we want to do is be able to manage by modularizing, which means that we break down our system into small modules.

This will allow us to manage change.

It will allow us to debug easily because if you break them up

into small modules, we test one module; make sure it works.

And then we can hide that module as if it were a box,

and look around it and see other modules.

As I said, it allows us to manage complexity.

So now we'll look at each of these in turn.

We will look at the game overall and break it down into modules

where each module will be specified either as a circle,

if it's a software, or a square, if it's hardware.

And we will describe which module calls which module by using arrows.

This is what our call graph is going to be like.

DR. JONATHAN VALVANO: What are the three files that one normally

creates for a software module?

DR. RAMESH YERRABALLI: The software module

is going to be managed by three files.

So let's take an example of the ADC module.

So when we look at a module like ADC, we will

have the specification of the module in a ADC.h.

DR. JONATHAN VALVANO: And what's in the H file?

DR. RAMESH YERRABALLI: The H file lists the public functions as prototypes.

DR. JONATHAN VALVANO: So these are the functions

that another module could call?

DR. RAMESH YERRABALLI: That is correct.

So this is what the module exposes to the outside world.

So we'll list all the functions.

And not just functions.

If there is any other data that needs to be exposed,

that's also going to be exposed in the header, as we call it.

And then we will have an ADC.c, which will be the implementation.

DR. JONATHAN VALVANO: So this is where the code that actually runs exists.

DR. RAMESH YERRABALLI: That is correct.

And we want to hide the implementation and expose only the public function.

As we saw, this allows us the flexibility

of improving the implementation or changing the implementation

without changing how the users call this module.

DR. JONATHAN VALVANO: But what's the third file?

DR. RAMESH YERRABALLI: And the third file

is not specific to the module itself, but how is the module used.

So the main.c, in our case, is going to be

the driver that will call this module.

The driver is also sometimes the tester of the module.

DR. JONATHAN VALVANO: So this file explicitly shows

how we're going to use the module, and how we tested the module?

DR. RAMESH YERRABALLI: That is correct.

And so at times when we're developing the code,

we will have in the main.c specific testing

code for each module that we are building.

And we test that module, and then we discard that piece of code

and we keep writing new testing software.

DR. JONATHAN VALVANO: Professor Yerraballi,

are there any guidelines in developing modular code?

DR. RAMESH YERRABALLI: There are guidelines.

These are something that you learn by practice.

But the general idea is to maximize the number of modules

by making sure that you minimize the

interdependency.

DR. JONATHAN VALVANO: Give me some examples.

What you mean by interdependency?

DR. RAMESH YERRABALLI: So the first idea is the idea of maximizing modules.

Means that I'm going to have as many modules as it

takes to break down the problem in a divide and conquer sort of way.

That is I'm going to break it into one module, another module.

If I feel like some functionality needs its own module,

I'm going to build another module for it.

But by interdependency we mean modules are going to exchange information.

So there's going to be modules communicating with each other.

This module may call this module.

This module may in turn call this module.

But part of this is how much data do they exchange.

The volume of the data, if you will, that they exchange.

Now we want to make sure that these modules are not

exchanging huge amounts of data.

And if two modules are exchanging a lot of data,

maybe there's some reason to think of them as one single module.

And so we want to maximize number of modules,

but we also want to minimize interdependency

because exchange of data between modules where

it gets too much that communication becomes bottle-neck.

DR. JONATHAN VALVANO: So let's look at modularity

from the perspective of our game.

Where do we begin?

DR. RAMESH YERRABALLI: So as I said, the first thing we're going to look at it

is the call graph.

So for our game, the heart of our game is going to be our game engine.

DR. JONATHAN VALVANO: So this is the main code

that controls all of the game function?

DR. RAMESH YERRABALLI: That is correct and so this game engine

is going to be using a bunch of different modules.

The first module I'm going to look at is that ADC module.

So the ADC module will encapsulate all functionality related to the slide pot.

Remember the slide pot is a piece of hardware,

so I'm going to write it like that.

So this is encapsulating the information that we specify for the slide pot.

DR. JONATHAN VALVANO: What does that arrow mean you just drew?

DR. RAMESH YERRABALLI: So the arrow is the call, the interface,

where we say that the game engine calls the ADC module, which

calls the hardware which is our slide pot.

So our second module we're going to look at is the switch hardware.

DR. JONATHAN VALVANO: These are our buttons.

DR. RAMESH YERRABALLI: These are our buttons.

So all the buttons, all the switches, that we have on our system

are going to be managed by this module.

The third module.

Obviously we want our game to be fun, and it should sound fun.

So we're going to have a sound module.

And the sound module will interact with a piece

of hardware, which is going to be our DAC, which, as you remember,

is going to be connected to the headphones.

So the last module is going to be our Nokia display

module, which is a Nokia 5110.

So this is our display module.

This is going to be responsible for all the graphics, and it's going to be interfacing with our Nokia display 5110

hardware LCD display.

DR. JONATHAN VALVANO: Now if you haven't bought a Nokia display, no worries.

You can still build your game because we built

into the process a simulated Nokia.

So you can actually run your game even if you

don't have the hardware for the graphics.

DR. RAMESH YERRABALLI: So we are indicating that simulated hardware

by this here, which is a cross between a square and a circle.

DR. JONATHAN VALVANO: You just invented something, Professor Yerraballi.

DR. RAMESH YERRABALLI: I did.

DR. JONATHAN VALVANO: A circle-square.

DR. RAMESH YERRABALLI: A circle-square.

So now that we saw the call graph, Jon, do you

see interesting things about this graph?

DR. JONATHAN VALVANO: I see that we have five modules 1, 2, 3, 4, 5.

But there are only four arrows between this.

This means that we have a lot of modules and very little interdependence.

The switch doesn't call sound.

The Nokia doesn't call the ADC, and the ADC doesn't call the game engine.

This is a very simple call graph.

DR. RAMESH YERRABALLI: So we saw the call graph,

now let's look at the data flow graph.

DR. JONATHAN VALVANO: The data flow graph

is another way to visualize the modularity.

DR. RAMESH YERRABALLI: So we see, we kind of recreated the call graph,

with five modules.

Our five modules are our game engine, our ADC, our switch, sound,

and the Nokia 5110 module.

Now we're going to see what the real exchange is going to be.

So let's look at one of them.

I see the slide pot, which is going to give us a position.

And we take the position, and we convert it into a voltage.

And we do this at 30Hz.

And what happens as I get this voltage into my system

is it's going to convert it to a 12-bit number,

and the 12-bit number is our measure of the position on the slide pot.

DR. JONATHAN VALVANO: But in a data flow graph,

what do the arrows mean this time?

DR. RAMESH YERRABALLI: So the arrows in the data flow graph

is the passage of information or data.

So in our case, the position, which is resistance if you will,

is going to be converted to a voltage.

Which is moving from the slide pot to the ADC.

Which is a zero to three volts.

Which is moving from the ADC to the driver.

Which is then moving from the driver to the game engine.

So it's the passage of information.

DR. JONATHAN VALVANO: And I can see we're passing 12 bits, 30 times

a second.

DR. RAMESH YERRABALLI: That is correct.

And so the other modules we'll see, we can quickly list.

So there are two inputs, our two switches.

And that is also being read at 30Hz, which

is going to be the two bits of information.

The two switches are two bits.

We pass that to the game engine.

These two together are what I think of as a interface.

And the interface is being sampled at 30 Hz by using the SysTick interrupt.

And this is what comes into the system.

And this is what goes out of the system, which is a sound.

Which in our case is going to be 11kHz.

Obviously, sound requires a more demanding timing.

So we have that flowing the other way.

So you see that there's a 4-bit DAC.

So the four bits go from the sound routine to the sound module

to the DAC, which is then fed to the headphones as a single analog signal.

And so this is our sound, which will also

be managed by a timing module-- which is a timer, which

we're going to look at later.

DR. JONATHAN VALVANO: Well so far, you've

just put the pieces together from the last three labs.

But what's new in this lab?

DR. RAMESH YERRABALLI: So the new thing in this lab is the graphics.

And the graphics are part of the LCD display, which is a Nokia display.

And we will try and abstract all the low level details

so that we can think of this as simply calling certain routines that either

display an image, clear the buffer, or redisplay a new buffer.

DR. JONATHAN VALVANO: Also in this data flow graph,

we have shown you the explicit hardware connections

that you must perform in order to share your game with other students.

And we see that in order to run your game engine with other students,

you're going to have to tie the slide pot to PE2.

The two buttons, whatever they do in your game,

need to be connected to PE1 and PE0.

The DAC, in order to create sounds that other people can hear,

we will specify the DAC must be on PB3, 2, 1, and 0.

And lastly, the Nokia needs port A. And so we're

going to use these five bits on PA 2, 3, 5, 6, and 7

to interface a Nokia display.

DR. RAMESH YERRABALLI: This is the standard that we're talking about.

This will allow everybody to play everybody else's game.

So the third piece of our modular design is flow charts.

DR. JONATHAN VALVANO: The flow chart is a way

to describe the software algorithm.

And you don't have to follow this explicitly,

but this is our advice on how we build our games.

Every software needs a main loop, so we have our main loop

which will begin when we hit the reset button.

As all embedded systems, we will begin our main loop with initializations.

So we'll initialize the PLL, the graphics, the DAC, the switches, the A

to D converter.

We'll initialize all of the components of our game.

And then after we've initialized everything,

we will enable interrupts In this way, all the initializations will

be complete before the interrupts are turned on.

In order to make the game extremely interactive,

our recommendation is that you do the LCD outputs, the graphical outputs,

do all the graphical output in the main loop in what we call the foreground.

So this is the foreground thread, here, which

executes when there's no interrupts.

Of all the things we do, the LCD output will be the slowest.

It'll take on the order of milliseconds to generate actual output

to the graphics.

The other two things we're going to do is things that occur very, very slowly.

And this will be a SysTick interrupt occurring at 30Hz.

30 hertz is fast enough that our eyes can't tell it's moved.

30 hertz is right at that range where it will look smooth.

And in our SysTick interrupt service routine

is where the actual game engine will occur.

And so your game, whatever it is, is going to look at the switches

and do whatever it is supposed to happen with those switches.

You'll look at the A to D converter, the slide pot

and do whatever your game is supposed to do with a slide pot.

DR. RAMESH YERRABALLI: Maybe movement.

DR. JONATHAN VALVANO: Yeah, like moving your ship around,

firing missiles, whatever is supposed to happen in your game.

And then we will decide to move sprites.

And a sprite we'll define later, but a sprite

is one of the objects of your game, one of the virtual elements of your game.

We will decide to move it.

Now we're not doing the graphics, we're just

deciding where to draw all the sprites.

DR. RAMESH YERRABALLI: So the movement is a virtual movement.

A physical movement is always done by the main program

by rendering this movement on the screen.

DR. JONATHAN VALVANO: Yes, and this is a virtual movement.

And you notice the flow chart of my SysTick interrupt service routine

follows the classic; arrows only go down.

This way the execution time of this interrupt service routine

will be short and bounded.

One of the reasons we have threads is so we can separate or define

individual software processes.

And the other process, the other action, that we want to create is sound.

And so we are going to separate the threads which produces sound from the thread which plays

the game by using two interrupt service routines.

This interrupt service routine is going to run a lot faster.

It's going to run at 11kHz and generate a sound.

And so if it's time or if we wish to play a sound,

we will do the output to a DAC.

And if we wish to not play a sound, we won't do one output to the DAC.

And notice that this interrupt service routine

separates the action of playing the sound

from the action of playing the game.

DR. RAMESH YERRABALLI: So you're saying that it's possible

that when I choose to fire, I want to produce a sound.

And so I schedule the action, but the action

itself is performed in the Timer2A ISR.

DR. JONATHAN VALVANO: Excellent.

So we could start a sound in our game, but the actual sound

is going to occur with a second interrupt,

occurring at a much faster rate.

DR. RAMESH YERRABALLI: This will also allow

us to change sounds from one sound to another sound,

depending upon what action has caused the sound.

DR. JONATHAN VALVANO: Absolutely.

DR. RAMESH YERRABALLI: So we've looked at the design.

We've delved into the three aspects of design,

which are call graph, our data flow graph, and flow charts.

And we are now going to look at some details

that we haven't covered yet in the previous chapters.

Help

The design process involves the conversion of a problem statement into hardware and software components. **Successive refinement** is the transformation from the general to the specific. In this section, we introduce the concept of modular programming and demonstrate that it is an effective way to organize our software projects. There are four reasons for forming modules. First, **functional abstraction** allows us to reuse a software module from multiple locations. Second, **complexity abstraction** allows us to divide a highly complex system into smaller less complicated components. The third reason is **portability**. If we create modules for the I/O devices, then we can isolate the rest of the system from the hardware details. This approach is sometimes called a hardware abstraction layer. Since all the software components that access an I/O port are grouped together, it will be easier to redesign the embedded system on a machine with different I/O ports. Finally, another reason for forming modules is **security**. Modular systems by design hide the inner workings from other modules and provide a strict set of mechanisms to access data and I/O ports. Hiding details and restricting access generates a more secure system.

Software must deal with **complexity**. Most real systems have many components, which interact in a complex manner. The size and interactions will make it difficult to conceptualize, abstract, visualize, and document. In this chapter we will present data flow graphs and call graphs as tools to describe interactions between components. Software must deal with **conformity**. All design, including software design, must interface with existing systems and with systems yet to be designed. Interfacing with existing systems creates an additional complexity. Software must deal with **changeability**. Most of the design effort involves change. Creating systems that are easy to change will help manage the rapid growth occurring in the computer industry.

The key to completing any complex task is to break it down into manageable subtasks. Modular programming is a style of software development that divides the software problem into distinct well-defined modules. The parts are as small as possible, yet relatively independent. Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent in **maintenance**. All five aspects of software maintenance

- Correcting mistakes,
- Adding new features,
- Optimizing for execution speed or program size,
- Porting to new computers or operating systems, and
- Reconfiguring the software to solve a similar related program

are simplified by organizing the software system into modules. The approach is particularly useful when a task is large enough to require several programmers.

A **program module** is a self-contained software task with clear entry and exit points. There is a distinct difference between a module and a C language function. A module is usually a collection of functions that in its entirety performs a well-defined set of tasks. A collection of 32-bit trigonometry functions is an example of a module. A device driver is a software module that facilitates the use of I/O. In particular it is collection of software functions for a particular I/O device. Modular programming involves both the specification of the individual modules and the connection scheme whereby the modules are interfaced together to form the software system. While the module may be called from many locations throughout the software, there should be well-defined **entry points**. In C, the entry point of a module is defined in the header file and is specified by a list of function prototypes for the public functions.

Common Error: In many situations the input parameters have a restricted range. It would be inefficient for the module and the calling routine to both check for valid input. On the other hand, an error may occur if neither checks for valid input.

An **exit point** is the ending point of a program module. The exit point of a function is used to return to the calling routine. We need to be careful about exit points. Similarly, if the function returns parameters, then all exit points should return parameters in an acceptable format. If the main program has an exit point it either stops the program or returns to the debugger. In most embedded systems, the main program does not exit.

In this section, an object refers to either a function or a data element. A **public** object is one that is shared by multiple modules. This means a public object can be accessed by other modules. Typically, we make the most general functions of a module public, so the functions can be called from other modules. For a module performing I/O, typical public functions include initialization, input, and output. A **private** object is one that is not shared. I.e., a private object can be accessed by only one module. Typically, we make the internal workings of a module private, so we hide how a private function works from user of the module. In an object-oriented language like C++ or Java, the programmer clearly defines a function or data object as public or private. The software in this course uses the naming convention of using the module name followed by an underline to identify the public functions of a module. For example if the module is ADC, then **`ADC_Init`** and **`ADC_Input`** are public functions. Functions without the underline in its name are private. In this manner we can easily identify whether a function or data object as public or private.

At a first glance, I/O devices seem to be public. For example, Port D resides permanently at the fixed address of 0x400073FC, and the programmer of every module knows that. In other words, from a syntactic viewpoint, any module has access to any I/O device. However, in order to reduce the complexity of the system, we will restrict the number of modules that actually do access the I/O device. From a “what do we actually do” perspective, however, we will write software that considers I/O devices as private, meaning an *I/O device should be accessed by only one module*. In general, it will be important to clarify which modules have access to I/O devices and when they are allowed to access them. When more than one module accesses an I/O device, then it is important to develop ways to arbitrate or synchronize. If two or more want to access the device simultaneously arbitration determines which module goes first. Sometimes the order of access matters, so we use synchronization to force a second module to wait until the first module is finished. Most microcontrollers do not have architectural features that restrict access to I/O ports, because it is assumed that all software burned into its ROM was designed for a common goal, meaning from a security standpoint one can assume there are no malicious components. However, as embedded systems become connected to the Internet, providing the power and flexibility, security will become important issue.

CHECKPOINT 15.1

Multiple modules may use Port F, where each module has an initialization. What conflict could arise around the initialization of a port?

Hide Answer

A conflict will occur if the modules had unfriendly access to the direction register, enable register or data register.

CHECKPOINT 15.2

What modules did you use in Lab 14? How could you tell it was a module?

Hide Answer

My solution has an ADC module and a Nokia module. Each module had a header file, with prototypes to public functions, and code file with the implementations of the functions.

CHECKPOINT 15.3

Help

How will modular design in Labs 13 and 14 affect your ability to do Lab 15?

Hide Answer

It will be simple to use the DAC and ADC in Lab 15, simply by including the DAC.c and ADC.c into the Lab 15 project. When we **#include** the DAC.h and ADC.h at the top of a file, we can now call the DAC and ADC functions.

Help