Minimize coupling | 15.2 Modular Design | UT... https://courses.edx.org/courses/UTAustinX/UT...

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/)

Information hiding is similar to minimizing coupling. It is better to separate the mechanisms of software from its policies. We should separate “what the function does” from “how the function works”. What a function does is defined by the relationship between its inputs and outputs. It is good to hide certain inner workings of a module and simply interface with the other modules through the well-defined input/output parameters. For example we could implement a variable size buffer by maintaining the current byte count in a global variable, **`Count`**. A good module will hide how **`Count`** is implemented from its users. If the user wants to know how many bytes are in the buffer, it calls a function that returns the count. A badly written module will not hide **`Count`** from its users. The user simply accesses the global variable **`Count`**. If we update the buffer routines, making them faster or better, we might have to update all the programs that access **`Count`** too. Allowing all software to access **`Count`** creates a security risk, making the system vulnerable to malicious or incompetent software. The object-oriented programming environments provide well-defined mechanisms to support information hiding. This separation of policies from mechanisms is discussed further in the section on layered software.

Help

Maintenance Tip: It is good practice to make all permanently-allocated data and all I/O devices private. Information is transferred from one module to another through well-defined function calls.

The **Keep It Simple Stupid** approach tries to generalize the problem so that the solution uses an abstract model. Unfortunately, the person who defines the software specifications may not understand the implications and alternatives. As a software developer, we always ask ourselves these questions:

“How important is this feature?”

“What if it worked this different way?”

Sometimes we can restate the problem to allow for a simpler and possibly more powerful solution. We begin the design of the game by listing possible modules for our system.

ADC	The interface to the joystick
Switch	User interaction with LEDs and switches
Sound	Sound output using the DAC
Nokia5110	Images displayed on the LCD
Game engine	The central controller that implements the game

Figure 15.1 shows a possible **call graph** for the game. An arrow in a call graph means software in one module can call functions in another module. This is a very simple organization with one master module and four slave modules. Notice the slave modules do not call each other. This configuration is an example of good modularization because there are 5 modules but only 4 arrows.

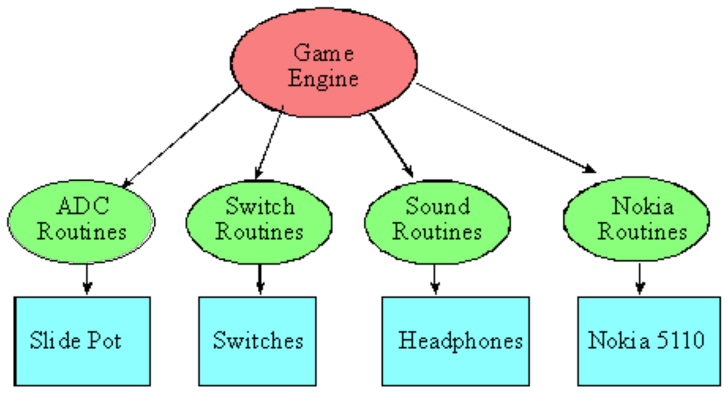

Figure 15.1 Possible call graph for the game.

Figure 15.2 shows on possible data flow graph for the game. Recall that arrows in a data flow graph represent data passing from one module to another. Notice the high bandwidth communication occurs between the sound module and its hardware, and between the Nokia5110 module and its hardware. We will design the system such that software modules do not need to pass a lot of data to other software modules.

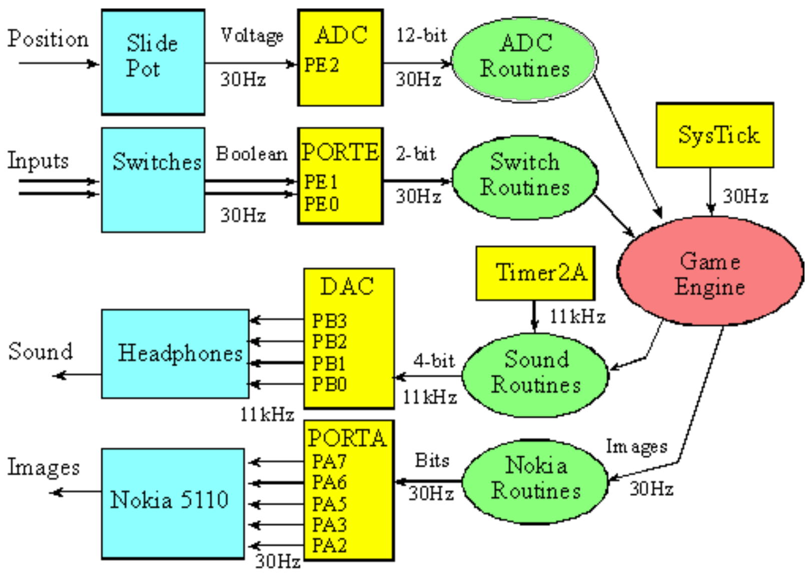

Figure 15.2. Possible data flow graph for the game. If you wish to add LEDs place them on PB4 and PB5.

The Timer2A ISR will output a sequence of numbers to the DAC to create sound. Let **`explosion`** be an array of 2000 4-bit numbers, representing a sound sampled at 11 kHz. If the game engine wishes to make the explosion sound, it calls

`Sound_Play(explosion,2000);` This function call simply passes a pointer to the **`explosion`** sound array into the sound module. The Timer2A ISR will output one 4-bit number to the DAC for the next 2000 interrupts. Notice the data flow from the game engine to the sound module is only two parameters (pointer and count), causing 2000 4-bit numbers to flow from the sound module to the DAC.

The Nokia5110 module needs to send 504 bytes to the LCD to create a new image on the screen. Since the screen is updated 30 times per second, 15120 bytes/sec will flow from the Nokia5110 module to its hardware. Let **`SmallEnemy30PointA`** be an array of numbers, representing a 16 by 10 pixel image of a small enemy. If the game engine wishes to place this enemy in the center of the screen, it calls **`Nokia5110_PrintBMP(24,48,SmallEnemy30PointA,0);`** This function call simply passes four parameters, one of which is a pointer to the image array into the Nokia5110 module. If the enemy is moving, then 15120 bytes/sec are flowing from the Nokia5110 module to the LCD, but the data flow from the game engine to the Nokia5110 module is only the four parameters (location, pointer and threshold) 30 times per second, which is 16 bytes*30/sec = 480 bytes/sec. The data flow from the game engine to the Nokia5110 module will increase linearly with the number of objects moving on the screen, but remain much smaller than the data into the LCD hardware.

Figure 15.3 shows on possible flow chart for the game engine. It is important to perform the actual LCD output in the foreground. In this design there are three threads: the main program and two interrupts. Multithreading allows the processor to execute multiple tasks. The main loop performs the game engine and updates the image on the screen. At 30 Hz, which is fast enough to look continuous, the SysTick ISR will sample the ADC and switch inputs. Based on user input and the game function, the ISR will decide what actions to take and signal the main program. To play a sound, we send the Sound module an array of data and arm Timer2A. Each Timer2A interrupt outputs one value to the DAC. When the sound is over we disarm Timer2A.

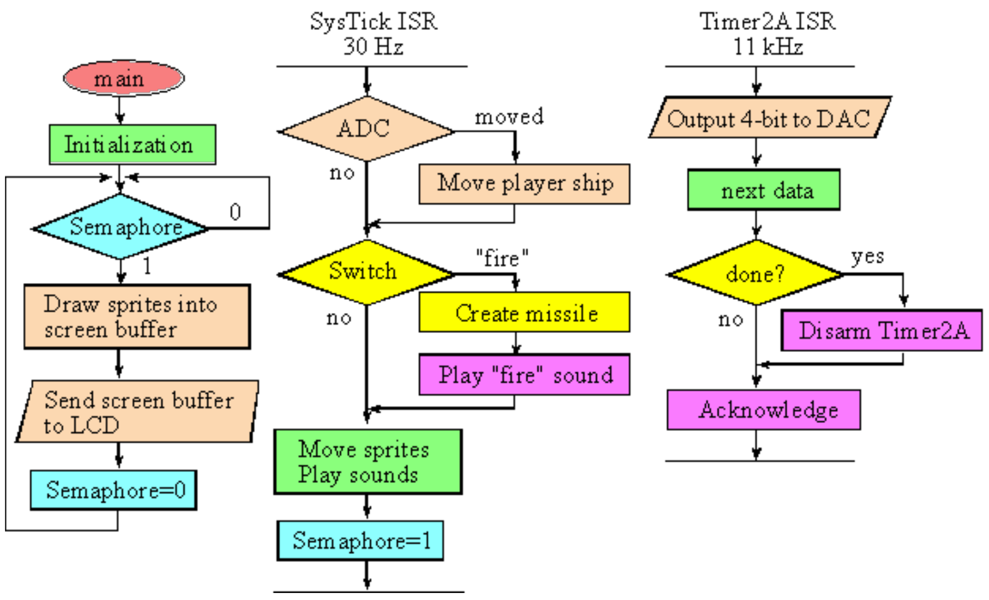

Figure 15.3. Possible flowchart for the game.

For example, if the ADC notices a motion to the left, the SysTick ISR can tell the main program to move the player ship to the left. Similarly, if the SysTick ISR notices the fire button has been pushed, it can create a missile object, and for the next 100 or so interrupts the SysTick ISR will move the missile until it goes off screen or hits something. In this way the missile moves a pixel or two every 33.3ms, causing its motion to look continuous. In summary, the ISR responds to input and time, but the main loop performs the actual output to the LCD.

CHECKPOINT 15.4

Notice the algorithm ins Figure 15.3 samples the ADC and the fire button at 30 Hz. How times/sec can we fire a missile or wiggle the slide pot? Hint: think Nyquist Theorem.

Hide Answer

According to the Nyquist Theorem we can oscillate it 15 times/sec.

CHECKPOINT 15.5

Similarly, in Figure 15.3, what frequency components are in the sound output?

Help

Hide Answer

According to the Nyquist Theorem we can faithfully represent sounds from 0 to 5.5 kHz.

Help