Interrupts

- C++ Keywords
- Lookup Tables
- Nested Vector Interrupt Controller (NVIC)
- Interrupts Lab Assignment

C++ Keywords

Sections of a binary

`.text`

Assembly instructions are placed within this section. When loaded onto a board, this will section will be placed into the flash memory (ROM) of the board. The binary file (.bin) that you load onto you board includes all of this information in it. The more code you write, the bigger the binary size gets.

`.data`

All global initialized variables are placed in this section. When the binary is created, in order to know what those global variables were at compile time, they are put into the binary which takes space on the ROM. At runtime, when the embedded system turns on, it moves the .data contents from the ROM to RAM so it can be used and modified by the application.

`.bss`

Section contains information about all uninitialized global variables. This will take up a small section in ROM, in that it only includes the start position in RAM and its length. The embedded platform will write zeros to to the start of that RAM location, extending its whole length. The `.bss` section must be cleared before using any **newlib** (stdclib and stdc++lib) libraries:

Reference: https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html#id2717944

Const

Out of all of the discussed keywords, **const** is probably the most commonly known keyword because of its simplicity as well as its ubiquity across many languages. But does it REALLY mean for a variable, class, or structure to be **const**?

Resource: https://en.cppreference.com/w/cpp/language/cv

As a Global Variable on an Embedded Platform

Take the following code:

```
const uint32_t kConstantVariable = 5;
int main(void)
{
  // ...
  return 0;}
```

What effect does the const in front of the variable type change the variable?

1. Does not allow the variable to be modified. Will throw a compiler error.
2. **Will be place the variable in ROM (or .text section).**
3. Removing the const will place

As a Local Variable on an Embedded Platform

1. Compiler will not allow modification of the variable.
2. Will be placed in the **STACK** as per a typical function call.

Cheating the system

```
const uint32_t kConstVariable = 5;
void AttemptToModifyConst()
{
  uint32_t * const_pointer = const_cast<uint32_t*>(&kConstVariable);
  *const_pointer = 10;  // Should cause system fault. Do not try this.
  const uint32_t kLocalConstVariable = 15;
```

```
  uint32_t * local_const_pointer = const_cast<uint32_t*>(&kLocalConstVariable);
  *local_const_pointer = 10;  // Should NOT cause system fault, but defeats the purpose of using const
}
```

If you cast away the const of a variable, you can attempt to change it. If the variable is global and placed in ROM, this will attempt a write access to the ROM which will cause a system fault. Doing so to local variables, which exists on the STACK which is within ram does not cause any faults, because the memory was always mutable. In the case of local variables, const is a means to keep you from compiling code that changes a constant.

Benefits of using Const in Embedded

If you evaluate many **MCUs**, you will see that most of them have a decent amount of flash but small amounts of RAM. If you have information that does not need to be modified at run time, you can shift the information into the ROM by placing it in global space (or make it static, see static section). This is why you should make you character strings, lookup tables, bitmaps and anything else const.

```
// Examples of good use cases for
const char kIpAddress = "192.168.1.5";
const char kUrl = "http://example.com/index.html";
const uint32_t kBitMasks[] = { 0xF0F0F0F, 0x55555555, 0xAAAAAAAA };
const uint8_t kMaximumRetries = 10;
```

Volatile

Every access to volatile variable will be treated as a visible side-effect. This means that:

1. The compiler CANNOT do optimizations such as out-of-order instruction reordering, link time garbage collection of the object or as-if transformations.
2. Access to object WILL bypass CPU cache and generate a bus cycle.

```
volatile uint32_t * pin_address = &LPC_GPIO0->PIN;
// Pin address will be loaded from the system bus and written back to the system bus.
// No caching will take place.*pin_address |= (1 << 5);
```

Benefits of using Volatile in Embedded

- Required in order to insure that access of registers is not optimized out.
- Prevents the register information from being access through the cache. To change a register, you need to write to it, but if you make the changes to cache memory, the actual hardware isn't read or written to.

Inline

Many people make the mistake in C++ in thinking that the **inline** keyword means that the contents of a function call will be inlined at the call site. This is not correct. In order to make this happen you must use the `always_inline` compiler attribute.

How define a function that will be inlined at its call site

```
// Modern C++17 and above attribute
[[gnu::always_inline]]
void CallsiteInlinedFunction(int a, int b)
{
 return a + b;
}
// Older version of GCC will require this
// Must seperate the attributes from the declaration
__attribute__((always_inline))
void CallsiteInlinedFunctionOld(int a, int b);
void CallsiteInlinedFunctionOld(int a, int b)
{
 return a + b;}
```

Inline functions and variables within header files

Inline functions and variables can be defined within a header file without the need to define them in a .cpp file. The will keep linker errors from appearing

> TODO: Add more detail as the above lacks it.

For member variables in class/structs

Static variables can be defined within a class if their declaration is preceded by the `inline` keyword.

Static

The **static** keyword has a load of different meanings depending on where it is used.

Static global variable or function

```
// Objects below are not visible outside of the .cpp file.
// Techincally works in .h/.hpp files but it defeats the purpose of putting it in there.
static uint8_t kHiddenBuffer[256];
static void FunctionPrivateToThisFile()
{
  // ...}
```

Stay away from using static this way in C++. If you would like to make a variable, object, type, etc private to a file you can use an anonymous namespace.

```
namespace
{
// Objects below are not visible outside of the .cpp file.
// Techincally works in .h/.hpp files but it defeats the purpose of putting it in there.
uint8_t kHiddenBuffer[256];
void FunctionPrivateToThisFile()
{
  // ...
}
}  // namespace
```

Static local function/method variable

Static variables within a function are actually apart of the `.data` section and not the stack. The also retain values across calls. This variable can be considered the state of the function.

```
uint32_t FunctionWithInternalStateVariable()
{
  static uint32_t call_count = 0;
  return ++call_count;}
```

Static class/struct member variable

```
class ClassWithStaticVariable
{
 public:
  ClassWithStaticVariable()
  {
    // NOTE: that this is NOT thread safe!
    id = next_id++;
  }
  // ...
 private:
  // This variable is common and accessable by all objects of this class
  // So if one class alters it, each class will see that change.
  inline static uint32_t next_id = 0;
  uint32_t id;};
```

Enum Class

You may be familiar with enumerations in C and C++. What enumeration class does is make

enumerations strong types.

```
// A clever way to force the user to your enumeration
// vs them potentially putting in an invalid value.
enum class TransferSpeed : uint32_t
{
  kHigh = 0b011,
  kFast = 0b010,
  kLow = 0b001,
  kDisabled = 0b111,
};
void SetTransferSpeed(TransferSpeed speed)
{
  // Because speed is not a integer type, you need to cast it into that type.
  *transfer_speed_register = static_cast<uint32_t>(speed);
}
// Usage ...
SetTransferSpeed(TransferSpeed::kHigh);  // OK
SetTransferSpeed(TransferSpeed::kFast);  // OK
SetTransferSpeed(0b00);  // Compiler Error!
```

Constexpr

Variables and function declared with the `constexpr` keyword exist only at compile time.

```
// Should return a mask like so:
//   AlternatingPatternMask(1) => 0b...0101'0101'0101'0101
//   AlternatingPatternMask(2) => 0b...1011'0110'1101'1011
//   AlternatingPatternMask(3) => 0b...0111'0111'0111'0111
//   etc ...
constexpr uint32_t AlternatingPatternMask(uint8_t number_of_ones_in_sequence)
{
  uint32_t result = 0;
  for (int i = 0; i < sizeof(uint16_t)*8; i++)
```

```
  {
    uint32_t set_this_bit = ((i % number_of_ones_in_sequence) == 0) ? 0 : 1;
    result |= set_this_bit << i;
  }
  return result;
}
// ...
// This value of this global variable is figured out at compile
// time and not at runtime.
// NOTE: this depends on the situation in which it is used.
uint32_t three_ones_in_sequence_mask = AlternatingPatternMask(3);
```

Using

> TODO: Fill this out later

?

Lookup Tables

Objective

To discuss lookup tables and how to use them to sacrifice storage space to increase computation time.

What Are Lookup Tables

Lookup tables are static arrays that sacrifices memory storage in place of a simple array index lookup of precalculated values. In some examples, a lookup table is not meant to speed a process, but simply an elegant solution to a problem.

Lets look at some examples to see why these are useful.

Why Use Lookup Tables

Simple Example: Convert Potentiometer Voltage to Angle

Lets make some assumptions about the system first:

1. Using an 8-bit ADC
2. Potentiometer is linear
3. Potentiometer sweep angle is 180 degrees
4. Potentiometer all the way left is 0 deg and 0V
5. Potentiometer all the way right (180 deg) is ADC Reference Voltage
6. Using a processor that does NOT have a FPU (**F**loating **P**oint arithmetic **U**nit) like the Arm Cortex M3 we use in the LPC1756.

```
double potADCToDegrees(uint8_t adc)
{
  return ((double)(adc))*(270/256);}
```

Code Block 1. Without Lookup

```
const double potentiometer_angles[256] =
{
//  [ADC] = Angle
  [0]  = 0.0,
  [1]  = 1.0546875,
  [2]  = 2.109375,
  [3]  = 3.1640625,
  [4]  = 4.21875,
  [5]  = 5.2734375,
  [6]  = 6.328125,
  [7]  = 7.3828125,
  [8]  = 8.4375,
  [9]  = 9.4921875,
  [10]  = 10.546875,
  [11]  = 11.6015625,
  [12]  = 12.65625,
  [13]  = 13.7109375,
  [14]  = 14.765625,
  [15]  = 15.8203125,
  [16]  = 16.875,
  [17]  = 17.9296875,
  [18]  = 18.984375,
  [19]  = 20.0390625,
  [20]  = 21.09375,
  [21]  = 22.1484375,
  [22]  = 23.203125,
  [23]  = 24.2578125,
  [24]  = 25.3125,
  [25]  = 26.3671875,
  [26]  = 27.421875,
```

□[27] □= 28.4765625,
□[28] □= 29.53125,
□[29] □= 30.5859375,
□[30] □= 31.640625,
□[31] □= 32.6953125,
□[32] □= 33.75,
□[33] □= 34.8046875,
□[34] □= 35.859375,
□[35] □= 36.9140625,
□[36] □= 37.96875,
□[37] □= 39.0234375,
□[38] □= 40.078125,
□[39] □= 41.1328125,
□[40] □= 42.1875,
□[41] □= 43.2421875,
□[42] □= 44.296875,
□[43] □= 45.3515625,
□[44] □= 46.40625,
□[45] □= 47.4609375,
□[46] □= 48.515625,
□[47] □= 49.5703125,
□[48] □= 50.625,
□[49] □= 51.6796875,
□[50] □= 52.734375,
□[51] □= 53.7890625,
□[52] □= 54.84375,
□[53] □= 55.8984375,
□[54] □= 56.953125,
□[55] □= 58.0078125,
□[56] □= 59.0625,
□[57] □= 60.1171875,
□[58] □= 61.171875,
□[59] □= 62.2265625,
□[60] □= 63.28125,
□[61] □= 64.3359375,
□[62] □= 65.390625,

□[63] □= 66.4453125,

□[64] □= 67.5,

□[65] □= 68.5546875,

□[66] □= 69.609375,

□[67] □= 70.6640625,

□[68] □= 71.71875,

□[69] □= 72.7734375,

□[70] □= 73.828125,

□[71] □= 74.8828125,

□[72] □= 75.9375,

□[73] □= 76.9921875,

□[74] □= 78.046875,

□[75] □= 79.1015625,

□[76] □= 80.15625,

□[77] □= 81.2109375,

□[78] □= 82.265625,

□[79] □= 83.3203125,

□[80] □= 84.375,

□[81] □= 85.4296875,

□[82] □= 86.484375,

□[83] □= 87.5390625,

□[84] □= 88.59375,

□[85] □= 89.6484375,

□[86] □= 90.703125,

□[87] □= 91.7578125,

□[88] □= 92.8125,

□[89] □= 93.8671875,

□[90] □= 94.921875,

□[91] □= 95.9765625,

□[92] □= 97.03125,

□[93] □= 98.0859375,

□[94] □= 99.140625,

□[95] □= 100.1953125,

□[96] □= 101.25,

□[97] □= 102.3046875,

□[98] □= 103.359375,

□[99] □= 104.4140625,
□[100] □= 105.46875,
□[101] □= 106.5234375,
□[102] □= 107.578125,
□[103] □= 108.6328125,
□[104] □= 109.6875,
□[105] □= 110.7421875,
□[106] □= 111.796875,
□[107] □= 112.8515625,
□[108] □= 113.90625,
□[109] □= 114.9609375,
□[110] □= 116.015625,
□[111] □= 117.0703125,
□[112] □= 118.125,
□[113] □= 119.1796875,
□[114] □= 120.234375,
□[115] □= 121.2890625,
□[116] □= 122.34375,
□[117] □= 123.3984375,
□[118] □= 124.453125,
□[119] □= 125.5078125,
□[120] □= 126.5625,
□[121] □= 127.6171875,
□[122] □= 128.671875,
□[123] □= 129.7265625,
□[124] □= 130.78125,
□[125] □= 131.8359375,
□[126] □= 132.890625,
□[127] □= 133.9453125,
□[128] □= 135,
□[129] □= 136.0546875,
□[130] □= 137.109375,
□[131] □= 138.1640625,
□[132] □= 139.21875,
□[133] □= 140.2734375,
□[134] □= 141.328125,

□[135] □= 142.3828125,
□[136] □= 143.4375,
□[137] □= 144.4921875,
□[138] □= 145.546875,
□[139] □= 146.6015625,
□[140] □= 147.65625,
□[141] □= 148.7109375,
□[142] □= 149.765625,
□[143] □= 150.8203125,
□[144] □= 151.875,
□[145] □= 152.9296875,
□[146] □= 153.984375,
□[147] □= 155.0390625,
□[148] □= 156.09375,
□[149] □= 157.1484375,
□[150] □= 158.203125,
□[151] □= 159.2578125,
□[152] □= 160.3125,
□[153] □= 161.3671875,
□[154] □= 162.421875,
□[155] □= 163.4765625,
□[156] □= 164.53125,
□[157] □= 165.5859375,
□[158] □= 166.640625,
□[159] □= 167.6953125,
□[160] □= 168.75,
□[161] □= 169.8046875,
□[162] □= 170.859375,
□[163] □= 171.9140625,
□[164] □= 172.96875,
□[165] □= 174.0234375,
□[166] □= 175.078125,
□[167] □= 176.1328125,
□[168] □= 177.1875,
□[169] □= 178.2421875,
□[170] □= 179.296875,

□[171] □= 180.3515625,
□[172] □= 181.40625,
□[173] □= 182.4609375,
□[174] □= 183.515625,
□[175] □= 184.5703125,
□[176] □= 185.625,
□[177] □= 186.6796875,
□[178] □= 187.734375,
□[179] □= 188.7890625,
□[180] □= 189.84375,
□[181] □= 190.8984375,
□[182] □= 191.953125,
□[183] □= 193.0078125,
□[184] □= 194.0625,
□[185] □= 195.1171875,
□[186] □= 196.171875,
□[187] □= 197.2265625,
□[188] □= 198.28125,
□[189] □= 199.3359375,
□[190] □= 200.390625,
□[191] □= 201.4453125,
□[192] □= 202.5,
□[193] □= 203.5546875,
□[194] □= 204.609375,
□[195] □= 205.6640625,
□[196] □= 206.71875,
□[197] □= 207.7734375,
□[198] □= 208.828125,
□[199] □= 209.8828125,
□[200] □= 210.9375,
□[201] □= 211.9921875,
□[202] □= 213.046875,
□[203] □= 214.1015625,
□[204] □= 215.15625,
□[205] □= 216.2109375,
□[206] □= 217.265625,

□[207] □= 218.3203125,
□[208] □= 219.375,
□[209] □= 220.4296875,
□[210] □= 221.484375,
□[211] □= 222.5390625,
□[212] □= 223.59375,
□[213] □= 224.6484375,
□[214] □= 225.703125,
□[215] □= 226.7578125,
□[216] □= 227.8125,
□[217] □= 228.8671875,
□[218] □= 229.921875,
□[219] □= 230.9765625,
□[220] □= 232.03125,
□[221] □= 233.0859375,
□[222] □= 234.140625,
□[223] □= 235.1953125,
□[224] □= 236.25,
□[225] □= 237.3046875,
□[226] □= 238.359375,
□[227] □= 239.4140625,
□[228] □= 240.46875,
□[229] □= 241.5234375,
□[230] □= 242.578125,
□[231] □= 243.6328125,
□[232] □= 244.6875,
□[233] □= 245.7421875,
□[234] □= 246.796875,
□[235] □= 247.8515625,
□[236] □= 248.90625,
□[237] □= 249.9609375,
□[238] □= 251.015625,
□[239] □= 252.0703125,
□[240] □= 253.125,
□[241] □= 254.1796875,
□[242] □= 255.234375,

```
[243] = 256.2890625,
[244] = 257.34375,
[245] = 258.3984375,
[246] = 259.453125,
[247] = 260.5078125,
[248] = 261.5625,
[249] = 262.6171875,
[250] = 263.671875,
[251] = 264.7265625,
[252] = 265.78125,
[253] = 266.8359375,
[254] = 267.890625,
[255] = 270
};
inline double potADCToDegrees(uint8_t adc)
{
return potentiometer_angles[adc];}
```

Code Block 2. With Lookup

With the two examples, it may seem trivial since the *WITHOUT* case is only "really" doing one calculation, mulitplying the **uint8_t** with (270/256) since the compiler will most likely optimize this value to its result. But if you take a look at the assembly, the results may shock you.

Look up Table Disassembly

```
00016e08 <main>:
main():
/var/www/html/SJSU-Dev/firmware/Experiements/L5_Application/main.cpp:322
    [254]   = 268.9411765,
    [255]   = 270
};
int main(void)
{
   16e08:	b082      	sub	sp, #8
/var/www/html/SJSU-Dev/firmware/Experiements/L5_Application/main.cpp:323
    volatile double a = potentiometer_angles[15];
```

```
   16e0a:	a303      	add	r3, pc, #12	; (adr r3, 16e18 <main+0x10>)
   16e0c:	e9d3 2300 	ldrd	r2, r3, [r3]
   16e10:	e9cd 2300 	strd	r2, r3, [sp]
   16e14:	e7fe      	b.n	16e14 <main+0xc>
   16e16:	bf00      	nop
   16e18:	c3b9a8ae 	.word	0xc3b9a8ae 16e1c:	402fc3c3 	.word	0x402fc3c3
```

Code Block 3. Dissassembly of Look up Table

Looks about right. You can see at **16e0a** the software is retrieving data from the lookup table, and then it is loading it into the double which is on the stack.

Double Floating Point Disassembly

```
00017c64 <__adddf3>:
__aeabi_dadd():
   17c64:	b530      	push	{r4, r5, lr}
   17c66:	ea4f 0441 	mov.w	r4, r1, lsl #1
   17c6a:	ea4f 0543 	mov.w	r5, r3, lsl #1
   17c6e:	ea94 0f05 	teq	r4, r5
   17c72:	bf08      	it	eq
   17c74:	ea90 0f02 	teqeq	r0, r2
   17c78:	bf1f      	itttt	ne
   17c7a:	ea54 0c00 	orrsne.w	ip, r4, r0
   17c7e:	ea55 0c02 	orrsne.w	ip, r5, r2
   17c82:	ea7f 5c64 	mvnsne.w	ip, r4, asr #21
   17c86:	ea7f 5c65 	mvnsne.w	ip, r5, asr #21
   17c8a:	f000 80e2 	beq.w	17e52 <__adddf3+0x1ee>
   17c8e:	ea4f 5454 	mov.w	r4, r4, lsr #21
   17c92:	ebd4 5555 	rsbs	r5, r4, r5, lsr #21
   17c96:	bfb8      	it	lt
   17c98:	426d      	neglt	r5, r5
   17c9a:	dd0c      	ble.n	17cb6 <__adddf3+0x52>
   17c9c:	442c      	add	r4, r5
   17c9e:	ea80 0202 	eor.w	r2, r0, r2
   17ca2:	ea81 0303 	eor.w	r3, r1, r3
   17ca6:	ea82 0000 	eor.w	r0, r2, r0
```

```
17caa:␣ea83 0101 ␣eor.w␣r1, r3, r1
17cae:␣ea80 0202 ␣eor.w␣r2, r0, r2
17cb2:␣ea81 0303 ␣eor.w␣r3, r1, r3
17cb6:␣2d36      ␣cmp␣r5, #54␣; 0x36
17cb8:␣bf88      ␣it␣hi
17cba:␣bd30      ␣pophi␣{r4, r5, pc}
17cbc:␣f011 4f00 ␣tst.w␣r1, #2147483648␣; 0x80000000
17cc0:␣ea4f 3101 ␣mov.w␣r1, r1, lsl #12
17cc4:␣f44f 1c80 ␣mov.w␣ip, #1048576␣; 0x100000
17cc8:␣ea4c 3111 ␣orr.w␣r1, ip, r1, lsr #12
17ccc:␣d002      ␣beq.n␣17cd4 <__adddf3+0x70>
17cce:␣4240      ␣negs␣r0, r0
17cd0:␣eb61 0141 ␣sbc.w␣r1, r1, r1, lsl #1
17cd4:␣f013 4f00 ␣tst.w␣r3, #2147483648␣; 0x80000000
17cd8:␣ea4f 3303 ␣mov.w␣r3, r3, lsl #12
17cdc:␣ea4c 3313 ␣orr.w␣r3, ip, r3, lsr #12
17ce0:␣d002      ␣beq.n␣17ce8 <__adddf3+0x84>
17ce2:␣4252      ␣negs␣r2, r2
17ce4:␣eb63 0343 ␣sbc.w␣r3, r3, r3, lsl #1
17ce8:␣ea94 0f05 ␣teq␣r4, r5
17cec:␣f000 80a7 ␣beq.w␣17e3e <__adddf3+0x1da>
17cf0:␣f1a4 0401 ␣sub.w␣r4, r4, #1
17cf4:␣f1d5 0e20 ␣rsbs␣lr, r5, #32
17cf8:␣db0d      ␣blt.n␣17d16 <__adddf3+0xb2>
17cfa:␣fa02 fc0e ␣lsl.w␣ip, r2, lr
17cfe:␣fa22 f205 ␣lsr.w␣r2, r2, r5
17d02:␣1880      ␣adds␣r0, r0, r2
17d04:␣f141 0100 ␣adc.w␣r1, r1, #0
17d08:␣fa03 f20e ␣lsl.w␣r2, r3, lr
17d0c:␣1880      ␣adds␣r0, r0, r2
17d0e:␣fa43 f305 ␣asr.w␣r3, r3, r5
17d12:␣4159      ␣adcs␣r1, r3
17d14:␣e00e      ␣b.n␣17d34 <__adddf3+0xd0>
17d16:␣f1a5 0520 ␣sub.w␣r5, r5, #32
17d1a:␣f10e 0e20 ␣add.w␣lr, lr, #32
17d1e:␣2a01      ␣cmp␣r2, #1
```

```
17d20:␣fa03 fc0e ␣lsl.w␣ip, r3, lr
17d24:␣bf28      ␣it␣cs
17d26:␣f04c 0c02 ␣orrcs.w␣ip, ip, #2
17d2a:␣fa43 f305 ␣asr.w␣r3, r3, r5
17d2e:␣18c0      ␣adds␣r0, r0, r3
17d30:␣eb51 71e3 ␣adcs.w␣r1, r1, r3, asr #31
17d34:␣f001 4500 ␣and.w␣r5, r1, #2147483648␣; 0x80000000
17d38:␣d507      ␣bpl.n␣17d4a <__adddf3+0xe6>
17d3a:␣f04f 0e00 ␣mov.w␣lr, #0
17d3e:␣f1dc 0c00 ␣rsbs␣ip, ip, #0
17d42:␣eb7e 0000 ␣sbcs.w␣r0, lr, r0
17d46:␣eb6e 0101 ␣sbc.w␣r1, lr, r1
17d4a:␣f5b1 1f80 ␣cmp.w␣r1, #1048576␣; 0x100000
17d4e:␣d31b      ␣bcc.n␣17d88 <__adddf3+0x124>
17d50:␣f5b1 1f00 ␣cmp.w␣r1, #2097152␣; 0x200000
17d54:␣d30c      ␣bcc.n␣17d70 <__adddf3+0x10c>
17d56:␣0849      ␣lsrs␣r1, r1, #1
17d58:␣ea5f 0030 ␣movs.w␣r0, r0, rrx
17d5c:␣ea4f 0c3c ␣mov.w␣ip, ip, rrx
17d60:␣f104 0401 ␣add.w␣r4, r4, #1
17d64:␣ea4f 5244 ␣mov.w␣r2, r4, lsl #21
17d68:␣f512 0f80 ␣cmn.w␣r2, #4194304␣; 0x400000
17d6c:␣f080 809a ␣bcs.w␣17ea4 <__adddf3+0x240>
17d70:␣f1bc 4f00 ␣cmp.w␣ip, #2147483648␣; 0x80000000
17d74:␣bf08      ␣it␣eq
17d76:␣ea5f 0c50 ␣movseq.w␣ip, r0, lsr #1
17d7a:␣f150 0000 ␣adcs.w␣r0, r0, #0
17d7e:␣eb41 5104 ␣adc.w␣r1, r1, r4, lsl #20
17d82:␣ea41 0105 ␣orr.w␣r1, r1, r5
17d86:␣bd30      ␣pop␣{r4, r5, pc}
17d88:␣ea5f 0c4c ␣movs.w␣ip, ip, lsl #1
17d8c:␣4140      ␣adcs␣r0, r0
17d8e:␣eb41 0101 ␣adc.w␣r1, r1, r1
17d92:␣f411 1f80 ␣tst.w␣r1, #1048576␣; 0x100000
17d96:␣f1a4 0401 ␣sub.w␣r4, r4, #1
17d9a:␣d1e9      ␣bne.n␣17d70 <__adddf3+0x10c>
```

```
17d9c:	f091 0f00 	teq	r1, #0
17da0:	bf04      	itt	eq
17da2:	4601      	moveq	r1, r0
17da4:	2000      	moveq	r0, #0
17da6:	fab1 f381 	clz	r3, r1
17daa:	bf08      	it	eq
17dac:	3320      	addeq	r3, #32
17dae:	f1a3 030b 	sub.w	r3, r3, #11
17db2:	f1b3 0220 	subs.w	r2, r3, #32
17db6:	da0c      	bge.n	17dd2 <__adddf3+0x16e>
17db8:	320c      	adds	r2, #12
17dba:	dd08      	ble.n	17dce <__adddf3+0x16a>
17dbc:	f102 0c14 	add.w	ip, r2, #20
17dc0:	f1c2 020c 	rsb	r2, r2, #12
17dc4:	fa01 f00c 	lsl.w	r0, r1, ip
17dc8:	fa21 f102 	lsr.w	r1, r1, r2
17dcc:	e00c      	b.n	17de8 <__adddf3+0x184>
17dce:	f102 0214 	add.w	r2, r2, #20
17dd2:	bfd8      	it	le
17dd4:	f1c2 0c20 	rsble	ip, r2, #32
17dd8:	fa01 f102 	lsl.w	r1, r1, r2
17ddc:	fa20 fc0c 	lsr.w	ip, r0, ip
17de0:	bfdc      	itt	le
17de2:	ea41 010c 	orrle.w	r1, r1, ip
17de6:	4090      	lslle	r0, r2
17de8:	1ae4      	subs	r4, r4, r3
17dea:	bfa2      	ittt	ge
17dec:	eb01 5104 	addge.w	r1, r1, r4, lsl #20
17df0:	4329      	orrge	r1, r5
17df2:	bd30      	popge	{r4, r5, pc}
17df4:	ea6f 0404 	mvn.w	r4, r4
17df8:	3c1f      	subs	r4, #31
17dfa:	da1c      	bge.n	17e36 <__adddf3+0x1d2>
17dfc:	340c      	adds	r4, #12
17dfe:	dc0e      	bgt.n	17e1e <__adddf3+0x1ba>
17e00:	f104 0414 	add.w	r4, r4, #20
```

```
17e04:	f1c4 0220 	rsb	r2, r4, #32
17e08:	fa20 f004 	lsr.w	r0, r0, r4
17e0c:	fa01 f302 	lsl.w	r3, r1, r2
17e10:	ea40 0003 	orr.w	r0, r0, r3
17e14:	fa21 f304 	lsr.w	r3, r1, r4
17e18:	ea45 0103 	orr.w	r1, r5, r3
17e1c:	bd30      	pop	{r4, r5, pc}
17e1e:	f1c4 040c 	rsb	r4, r4, #12
17e22:	f1c4 0220 	rsb	r2, r4, #32
17e26:	fa20 f002 	lsr.w	r0, r0, r2
17e2a:	fa01 f304 	lsl.w	r3, r1, r4
17e2e:	ea40 0003 	orr.w	r0, r0, r3
17e32:	4629      	mov	r1, r5
17e34:	bd30      	pop	{r4, r5, pc}
17e36:	fa21 f004 	lsr.w	r0, r1, r4
17e3a:	4629      	mov	r1, r5
17e3c:	bd30      	pop	{r4, r5, pc}
17e3e:	f094 0f00 	teq	r4, #0
17e42:	f483 1380 	eor.w	r3, r3, #1048576	; 0x100000
17e46:	bf06      	itte	eq
17e48:	f481 1180 	eoreq.w	r1, r1, #1048576	; 0x100000
17e4c:	3401      	addeq	r4, #1
17e4e:	3d01      	subne	r5, #1
17e50:	e74e      	b.n	17cf0 <__adddf3+0x8c>
17e52:	ea7f 5c64 	mvns.w	ip, r4, asr #21
17e56:	bf18      	it	ne
17e58:	ea7f 5c65 	mvnsne.w	ip, r5, asr #21
17e5c:	d029      	beq.n	17eb2 <__adddf3+0x24e>
17e5e:	ea94 0f05 	teq	r4, r5
17e62:	bf08      	it	eq
17e64:	ea90 0f02 	teqeq	r0, r2
17e68:	d005      	beq.n	17e76 <__adddf3+0x212>
17e6a:	ea54 0c00 	orrs.w	ip, r4, r0
17e6e:	bf04      	itt	eq
17e70:	4619      	moveq	r1, r3
17e72:	4610      	moveq	r0, r2
```

```
17e74:␣bd30      ␣pop␣{r4, r5, pc}
17e76:␣ea91 0f03 ␣teq␣r1, r3
17e7a:␣bf1e      ␣ittt␣ne
17e7c:␣2100      ␣movne␣r1, #0
17e7e:␣2000      ␣movne␣r0, #0
17e80:␣bd30      ␣popne␣{r4, r5, pc}
17e82:␣ea5f 5c54 ␣movs.w␣ip, r4, lsr #21
17e86:␣d105      ␣bne.n␣17e94 <__adddf3+0x230>
17e88:␣0040      ␣lsls␣r0, r0, #1
17e8a:␣4149      ␣adcs␣r1, r1
17e8c:␣bf28      ␣it␣cs
17e8e:␣f041 4100 ␣orrcs.w␣r1, r1, #2147483648␣; 0x80000000
17e92:␣bd30      ␣pop␣{r4, r5, pc}
17e94:␣f514 0480 ␣adds.w␣r4, r4, #4194304␣; 0x400000
17e98:␣bf3c      ␣itt␣cc
17e9a:␣f501 1180 ␣addcc.w␣r1, r1, #1048576␣; 0x100000
17e9e:␣bd30      ␣popcc␣{r4, r5, pc}
17ea0:␣f001 4500 ␣and.w␣r5, r1, #2147483648␣; 0x80000000
17ea4:␣f045 41fe ␣orr.w␣r1, r5, #2130706432␣; 0x7f000000
17ea8:␣f441 0170 ␣orr.w␣r1, r1, #15728640␣; 0xf00000
17eac:␣f04f 0000 ␣mov.w␣r0, #0
17eb0:␣bd30      ␣pop␣{r4, r5, pc}
17eb2:␣ea7f 5c64 ␣mvns.w␣ip, r4, asr #21
17eb6:␣bf1a      ␣itte␣ne
17eb8:␣4619      ␣movne␣r1, r3
17eba:␣4610      ␣movne␣r0, r2
17ebc:␣ea7f 5c65 ␣mvnseq.w␣ip, r5, asr #21
17ec0:␣bf1c      ␣itt␣ne
17ec2:␣460b      ␣movne␣r3, r1
17ec4:␣4602      ␣movne␣r2, r0
17ec6:␣ea50 3401 ␣orrs.w␣r4, r0, r1, lsl #12
17eca:␣bf06      ␣itte␣eq
17ecc:␣ea52 3503 ␣orrseq.w␣r5, r2, r3, lsl #12
17ed0:␣ea91 0f03 ␣teqeq␣r1, r3
17ed4:␣f441 2100 ␣orrne.w␣r1, r1, #524288␣; 0x80000
17ed8:␣bd30      ␣pop␣{r4, r5, pc}
```

```
17eda:□bf00      □nop
```

Code Block 4. Arm Software Floating Point Addition Implementation

This isn't even the full code. This is a function that our calculation function has to run each time it wants to add two doubles together. Also, note that it is not just a straight shot of 202 instructions, because you can see that there are loops in the code where ever you see an instruction's mnemonic that starts with the letter **b** (stands for branch).

Other Use Cases

- Correlate degrees to radians (assuming degrees are whole numbers)
- Table of cosine or sine given radians or degrees
  - In the radians case, you will need to create your own trivial hashing function to convert radians to an index
- Finding a number of bits SET in a 32-bit number
  - Without a lookup table time complexity is O(n) where (n = 32), the number of bits you want to look through
  - With a lookup table, the time complexity is O(1), constant time, and only needs the followin operations
    - 3 bitwise left shifts operations
    - 4 bitwise ANDS operations
    - 4 load from memory addresses
    - 4 binary ADD operations
      - Total of 15 operations total

```
/* Found this on wikipedia! */
/* Pseudocode of the lookup table 'uint32_t bits_set[256]' */
 /*                    0b00, 0b01, 0b10, 0b11, 0b100, 0b101, ... */
 int bits_set[256] = {    0,    1,    1,    2,     1,     2, // 200+ more entries
 /* (this code assumes that 'int' is an unsigned 32-bits wide integer) */
 int count_ones(unsigned int x) {
    return bits_set[ x        & 255] + bits_set[(x >>  8) & 255]
         + bits_set[(x >> 16) & 255] + bits_set[(x >> 24) & 255]; }
```

Code Block 5. Bits set in a 32-bit number (Found this on wikipedia (look up tables))

> There are far more use cases then this, but these are a few.

Lookup Table Decision Tree

Lookup tables can be used as elegant ways to structure information. In this case, they may not provide a speed up but they will associate indexes with something greater, making your code more readable and easier to maintain. In this example, we will be looking at a matrix of function pointers.

Example: Replace Decision Tree

See the function below:

```
void makeADecisionRobot(bool power_system_nominal, bool no_obstacles_ahead)
{
   if(power_system_nominal && no_obstacles_ahead)
   {
      moveForward();
   }
   else if(power_system_nominal && !no_obstacles_ahead)
   {
      moveOutOfTheWay();
   }
   else if(!power_system_nominal && no_obstacles_ahead)
   {
      slowDown();
   }
   else
   {
      emergencyStop();
   }}
```

Code Block 6. Typical Decision Tree

```
void (* decision_matrix)(void)[2][2] =
{
```

```
   [1][1] = moveForward
   [1][0] = moveOutOfTheWay,
   [0][1] = slowDown,
   [0][0] = emergencyStop,
};
void makeADecisionRobot(bool power_system_nominal, bool no_obstacles_ahead)
{
   decision_matrix[power_system_nominal][no_obstacles_ahead]();}
```

Code Block 7. Lookup Table Decision Tree

The interesting thing about the decision tree is that it is also more optimal in that, it takes a few instructions to do the look up from memory, then the address of the procedure [function] is looked up an executed, where the former required multiple read instructions and comparison instructions.

Making LabGPIO Easier

In the LabGPIO assignment you were required to make the class handle multiple ports. Most likely you used a **switch case** or **if - if else - else** statements to switch between the registers that you control. BUT! A helpful way to get around this is to use a lookup table.

```
class LabGPIO
{
 public:
  // Table of GPIO ports located in LPC memory map ordered in such a way that
  // using the port number in the braces looks up the appropriate gpio register
  // For example LPC_GPIO2 can be found by using:
  //
  //    gpio[2] == LPC_GPIO2
  //    gpio[n] == LPC_GPIOn
  //
  inline static LPC_GPIO_TypeDef * gpio[6] = {
    LPC_GPIO0, LPC_GPIO1, LPC_GPIO2, LPC_GPIO3, LPC_GPIO4, LPC_GPIO5
  };

  // ...
```

```
//
// Switch case version
//
void SetHigh()
{
  switch(port)
  {
    case 0:
      LPC_GPIO0->SET = (1 << pin);
      break;
    case 1:
      LPC_GPIO1->SET = (1 << pin);
      break;
    case 2:
      LPC_GPIO2->SET = (1 << pin);
      break;
    case 3:
      LPC_GPIO3->SET = (1 << pin);
      break;
    case 4:
      LPC_GPIO4->SET = (1 << pin);
      break;
    case 5:
      LPC_GPIO5->SET = (1 << pin);
      break;
  }
}
//
// Lookup table version
//
void SetHigh()
{
  gpio[port]->SET = (1 << pin);
}
// ...};
```

https://godbolt.org/z/kdVZ4P

Here is a list of benefits you get form using a lookup table:

1. [Maintance] Minimizes redundant code
2. [Space Optimize] Minimizes binary size
3. [Time Optimize] *Eliminates comparisons & branching
4. [Readability] Easier to understand

Minimizes redundant code

If you look at each case you see that they all look exactly the same. The problem with redundant code is that maintaining it can be a problem.

For example: Lets say you need to shift, not by the pin, but by the pin+1. Now you must change all 6 lines of code to make this happen. There is a chance that you may forget one of the lines. And if you do not have a test to check this for you, it could result in a hard to find bug, with other developers have issues with your driver.

Using a lookup table makes the code easier to maintain.

Minimizes binary size

If you look at each case, you will see that each one has its own set of code to do. The compiler does not notice that there is a pattern with the code, and that the same operation is done the same way just at different addresses. So for each case, a new set of instructions is required. With the lookup table, only 1 set of instructions is needed and thus the amount of code in the **.text** section of your executable decreases compared to the switch case.

Using a lookup table, if used properly, can make your code smaller.

Eliminates Comparisons & Branching

There was an * above, because this is not always the case. Some switches cases can be converted into lookup tables. In the case above, the switch cases will be converted into a lookup table of program counter offsets. This means that the code will jump directly to the set of code they need to run without having to do any comparisons. But in some cases, the compiler cannot make the judgement and falls back to branching.

Comparisons and branching are not a bad thing, but they tend to take a few instructions to work and must check each case, one by one before actually running any of them, whereas the lookup table only need to use a few mathematical operations to achieve its goal.

Using a lookup table can eliminate comparison and branching and minimize what would normally be a worst case O(n) complexity to O(1) time complexity.

Easier to understand

One concern that an on looker to your code may have is, do each of these cases do the same thing or is there some nuance to each one. This will force the reader to have to check each case to see if they all do the same thing. Where as the single or few lines of a lookup table tends to tell you that each element of the table has the same set of operations done on it.

Using a lookup table can make reading code easier because it allows the reader to understand that each case or element has the same set of operations done it.

Points of caution

Do not overuse lookup tables and make sure your lookup table is an efficient size. If the distance between useful points in the lookup table are too far apart, you end up wasting space.

Also, note that the lookup table itself requires space and sometimes that space could be more then the space that a switch case would take up. You need to check the disassembly and the binary section sizes to make sure that for your specific use case, that lookup tables are right for you.

Nested Vector Interrupt Controller (NVIC)

Objective

This tutorial demonstrates how to use interrupts on a processor. In general, you will understand the concept behind interrupts on any processor, but we will use the SJ-One board as an example.

What is an interrupt?

An interrupt is the hardware capability of a CPU to break the normal flow of software to attend an urgent request.

The science behind interrupts lies in the hardware that allows the CPU to be interrupted. Each peripheral in a microcontroller *may be* able to assert an interrupt to the CPU core, and then the CPU core would jump to the corresponding interrupt service routine (**ISR**) to service the interrupt.

ISR Procedure

The following steps demonstrate what happens when an interrupt occurs :

- CPU manipulates the PC (program counter) to jump to the ISR
- **IMPORTANT**: CPU will disable interrupts (or that priority level's interrupts until end of ISR)
- Registers are saved before running the ISR (pushed onto the stack)
- ISR is run
- Registers are restored (popped from stack)
- Interrupts are re-enabled (or that priority level's interrupt is re-enabled)

On some processors, the savings and restoring of registers is a manual step and the compiler would help you do it. You can google "GCC interrupt attribute" to study this topic further. On SJ-One board,

which us ware.

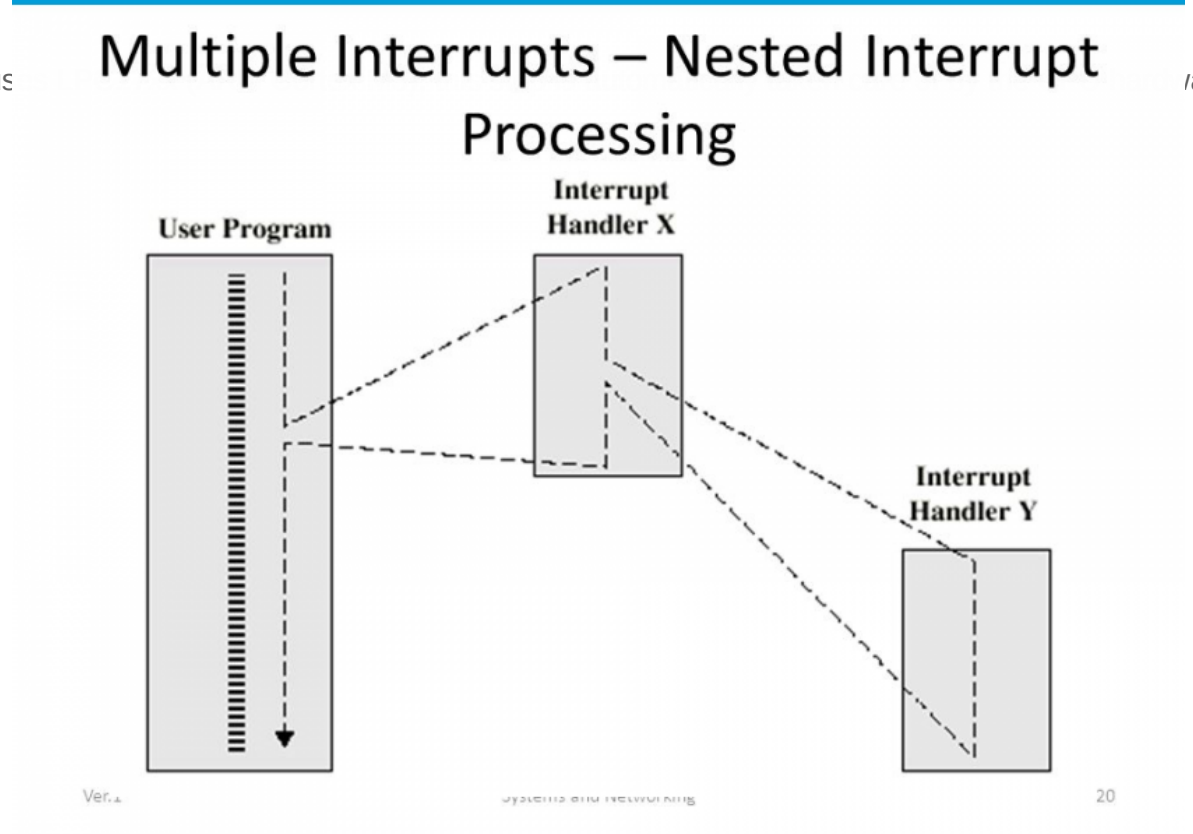


Figure 1. Nested Interrupt Processing

Nested Vector Interrupt Controller

Nested Vector Interrupt Controllers or NVIC for short, have two properties:

- Can handle multiple interrupts.
  - The number of interrupts implemented is device dependent.
- A programmable priority level for each interrupt.
  - A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Grouping of priority values into group priority and sub-priority fields.
  - This means that interrupts of the same priority are grouped together and do not preempt each other.
  - Each interrupt also has a sub-priority field which is used to figure out the run order of pending interrupts of the same priority.

- Interrupt tail-chaining.
  - This enables back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
  - This saves us from the step of having to restore and then save the registers again.
- An external N[illegible]

NVIC Interrupt Example

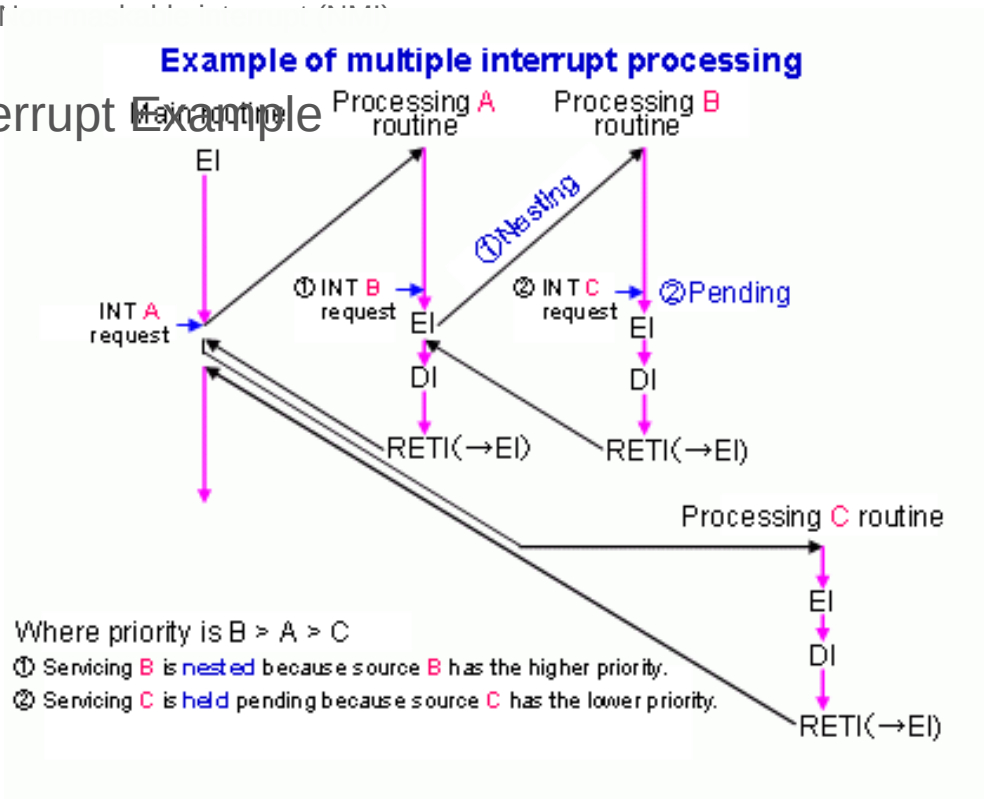


Figure 2. Multiple Interrupt Processing

The SW to HW Connection

Now that we understand how the CPU hardware services interrupts, we need to define how we inform the CPU WHERE our ISR function is located at.

Interrupt Vector Table

This table is nothing but addresses of functions that correspond to the microcontroller interrupts. Specific interrupts use specific "slots" in this table, and we have to populate these spots with our software functions that service the interrupts.

Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (external NMI input)
3	Hard fault	–1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system service
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
…	…	…	…
255	IRQ #239	Programmable	External interrupt #239

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported. In addition, the Cortex-M3 also has an NMI interrupt input. When it is asserted, the NMI-ISR is executed unconditionally.

Figure 3. HW Interrupt Vector Table

SJOne (LPC17xx) Example

The using a linker script and compiler directives (commands for the compiler), the compiler is able to place the software interrupt vector table at a specific location that the CPU expects the interrupt vector table to be located at. This connects the dots about how the CPU is able to determine WHERE your interrupt service routines are located at. From there on, anytime a specific interrupt occurs, the CPU is able to fetch the address and make the JUMP.

```
/**
 * CPU interrupt vector table that is loaded at the beginning of the CPU start
 * location by using the linker script that will place it at the isr_vector location.
 * CPU loads the stack pointer and begins execution from Reset vector.
 */
```

```
extern void (* const g_pfnVectors[])(void);
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    // Core Level - CM3
    &_vStackTop,        // The initial stack pointer
    isr_reset,          // The reset handler
    isr_nmi,            // The NMI handler
    isr_hard_fault,     // The hard fault handler
    isr_mem_fault,      // The MPU fault handler
    isr_bus_fault,      // The bus fault handler
    isr_usage_fault,    // The usage fault handler
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    vPortSVCHandler,    // FreeRTOS SVC-call handler (naked function so needs direct call - not a wrap
    isr_debug_mon,      // Debug monitor handler
    0,                  // Reserved
    xPortPendSVHandler, // FreeRTOS PendSV handler (naked function so needs direct call - not a wrappe
    isr_sys_tick,       // FreeRTOS SysTick handler (we enclose inside a wrapper to track OS overhead)
    // Chip Level - LPC17xx - common ISR that will call the real ISR
    isr_forwarder_routine,      // 16, 0x40 - WDT
    isr_forwarder_routine,      // 17, 0x44 - TIMER0
    isr_forwarder_routine,      // 18, 0x48 - TIMER1
    isr_forwarder_routine,      // 19, 0x4c - TIMER2
    isr_forwarder_routine,      // 20, 0x50 - TIMER3
    isr_forwarder_routine,      // 21, 0x54 - UART0
    isr_forwarder_routine,      // 22, 0x58 - UART1
    isr_forwarder_routine,      // 23, 0x5c - UART2
    isr_forwarder_routine,      // 24, 0x60 - UART3
    isr_forwarder_routine,      // 25, 0x64 - PWM1
    isr_forwarder_routine,      // 26, 0x68 - I2C0
    isr_forwarder_routine,      // 27, 0x6c - I2C1
    isr_forwarder_routine,      // 28, 0x70 - I2C2
    isr_forwarder_routine,      // 29, 0x74 - SPI
```

```
    isr_forwarder_routine,      // 30, 0x78 - SSP0
    isr_forwarder_routine,      // 31, 0x7c - SSP1
    isr_forwarder_routine,      // 32, 0x80 - PLL0 (Main PLL)
    isr_forwarder_routine,      // 33, 0x84 - RTC
    isr_forwarder_routine,      // 34, 0x88 - EINT0
    isr_forwarder_routine,      // 35, 0x8c - EINT1
    isr_forwarder_routine,      // 36, 0x90 - EINT2
    isr_forwarder_routine,      // 37, 0x94 - EINT3
    isr_forwarder_routine,      // 38, 0x98 - ADC
    isr_forwarder_routine,      // 39, 0x9c - BOD
    isr_forwarder_routine,      // 40, 0xA0 - USB
    isr_forwarder_routine,      // 41, 0xa4 - CAN
    isr_forwarder_routine,      // 42, 0xa8 - GP DMA
    isr_forwarder_routine,      // 43, 0xac - I2S
    isr_forwarder_routine,      // 44, 0xb0 - Ethernet
    isr_forwarder_routine,      // 45, 0xb4 - RITINT
    isr_forwarder_routine,      // 46, 0xb8 - Motor Control PWM
    isr_forwarder_routine,      // 47, 0xbc - Quadrature Encoder
    isr_forwarder_routine,      // 48, 0xc0 - PLL1 (USB PLL)
    isr_forwarder_routine,      // 49, 0xc4 - USB Activity interrupt to wakeup
    isr_forwarder_routine,      // 50, 0xc8 - CAN Activity interrupt to wakeup};
```

Code Block 1. Software Interrupt Vector Table

> **NOTE:** that a vector table is really just a lookup table that hardware utilizes.

Two Methods to setup an ISR on the SJOne

?

All of the methods require that you run this function to allow the NVIC to accept a particular interrupt request.

NVIC_EnableIRQ(EINT3_IRQn);

Where the input is the IRQ number. This can be found in the LCP17xx.h file. Search for **enum IRQn**.

Method 1. Modify IVT

DO NOT DO THIS, unless you really know what you are doing. The ISR forwarder works with FreeRTOS to distinguish CPU utilization between ISRs and tasks.

I highly discourage modifying the **startup.cpp** and modifying the vector tables directly. Its not dynamic is less manageable in that, if you switch projects and the ISR doesn't exist, the compiler will through an error.

?

IVT modify

```
/* You will need to include the header file that holds the ISR for this to work */
#include "my_isr.h"
extern void (* const g_pfnVectors[])(void);
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    // Core Level - CM3
    &_vStackTop,        // The initial stack pointer
    isr_reset,          // The reset handler
    isr_nmi,            // The NMI handler
    isr_hard_fault,     // The hard fault handler
    isr_mem_fault,      // The MPU fault handler
    isr_bus_fault,      // The bus fault handler
    isr_usage_fault,    // The usage fault handler
    0,                  // Reserved
    0,                  // Reserved
```

```
    0,                  // Reserved
    0,                  // Reserved
    vPortSVCHandler,    // FreeRTOS SVC-call handler (naked function so needs direct call - not a wrap
    isr_debug_mon,      // Debug monitor handler
    0,                  // Reserved
    xPortPendSVHandler, // FreeRTOS PendSV handler (naked function so needs direct call - not a wrappe
    isr_sys_tick,       // FreeRTOS SysTick handler (we enclose inside a wrapper to track OS overhead)
    // Chip Level - LPC17xx - common ISR that will call the real ISR
    isr_forwarder_routine,      // 16, 0x40 - WDT
    isr_forwarder_routine,      // 17, 0x44 - TIMER0
    isr_forwarder_routine,      // 18, 0x48 - TIMER1
    isr_forwarder_routine,      // 19, 0x4c - TIMER2
    isr_forwarder_routine,      // 20, 0x50 - TIMER3
    isr_forwarder_routine,      // 21, 0x54 - UART0
    isr_forwarder_routine,      // 22, 0x58 - UART1
    isr_forwarder_routine,      // 23, 0x5c - UART2
    isr_forwarder_routine,      // 24, 0x60 - UART3
    isr_forwarder_routine,      // 25, 0x64 - PWM1
    isr_forwarder_routine,      // 26, 0x68 - I2C0
    isr_forwarder_routine,      // 27, 0x6c - I2C1
    isr_forwarder_routine,      // 28, 0x70 - I2C2
    isr_forwarder_routine,      // 29, 0x74 - SPI
    isr_forwarder_routine,      // 30, 0x78 - SSP0
    isr_forwarder_routine,      // 31, 0x7c - SSP1
    isr_forwarder_routine,      // 32, 0x80 - PLL0 (Main PLL)
    isr_forwarder_routine,      // 33, 0x84 - RTC
    isr_forwarder_routine,      // 34, 0x88 - EINT0
    isr_forwarder_routine,      // 35, 0x8c - EINT1
    isr_forwarder_routine,      // 36, 0x90 - EINT2
    runMyISR,                   // 37, 0x94 - EINT3 <---- NOTICE how I changed the name here
    isr_forwarder_routine,      // 38, 0x98 - ADC
    isr_forwarder_routine,      // 39, 0x9c - BOD
    isr_forwarder_routine,      // 40, 0xA0 - USB
    isr_forwarder_routine,      // 41, 0xa4 - CAN
    isr_forwarder_routine,      // 42, 0xa8 - GP DMA
    isr_forwarder_routine,      // 43, 0xac - I2S
```

```
    isr_forwarder_routine,      // 44, 0xb0 - Ethernet
    isr_forwarder_routine,      // 45, 0xb4 - RITINT
    isr_forwarder_routine,      // 46, 0xb8 - Motor Control PWM
    isr_forwarder_routine,      // 47, 0xbc - Quadrature Encoder
    isr_forwarder_routine,      // 48, 0xc0 - PLL1 (USB PLL)
    isr_forwarder_routine,      // 49, 0xc4 - USB Activity interrupt to wakeup
    isr_forwarder_routine,      // 50, 0xc8 - CAN Activity interrupt to wakeup};
```

Code Block 3. Weak Function Override Template

Method 2. ISR Register Function

The **EINT3_IRQn** symbol is defined in an enumeration in LPC17xx.h. All you need to do is specify the IRQ number and the function you want to act as an ISR. This will then swap out the previous ISR with your function.

> This is the best option! Please use this option almost always!

```
/**
 * Just your run-of-the-mill function
 */
void myEINT3ISR(void)
{
    doSomething();
    clearInterruptFlag();
}
int main()
{
    /**
     * Find the IRQ number for the interrupt you want to define.
     * In this case, we want to override IRQ 0x98 EINT3
     * Then specify a function pointer that will act as your ISR
     */
    RegisterIsr(EINT3_IRQn, myEINT3ISR);
    NVIC_EnableIRQ(EINT3_IRQn);}
```

Code Block 5. Weak Function Override Template

PROS	CONS

• Can dynamically change ISR during runtime.
• Does not disturb core library files in the process of adding/changing ISRs.
  ○ Always try to prevent changes to the core libraries.
• Does not cause compiler errors.
• Your ISR cpu utilization is tracked.
	• Must wait until **main** is called before ISR is registered
	  ○ Interrupt events could happen before main begins.

What to do inside an ISR

Do very little inside an ISR. When you are inside an ISR, the whole system is blocked (other than higher priority interrupts). If you spend too much time inside the ISR, then you are destroying the real-time operating system principle and everything gets clogged.

With that said, here is the general guideline:

Short as possible

DO NOT POLL FOR ANYTHING! Try to keep loops as small as possible. Note that printing data over UART can freeze the entire system, including the RTOS for that duration. For instance, printing 4 chars may take 1ms at 38400bps.

FreeRTOS API calls

If you are using FreeRTOS API, you must use **FromISR** functions only! If a FromISR function does not exist, then don't use that API.

Clear Interrupt Sources

Clear the source of the interrupt. For example, if interrupt was for rising edge of a pin, clear the "rising edge" bit such that you will not re-enter into the same interrupt function.

> ? If you don't do this, your interrupt will get stuck in an infinite ISR call loop. For the Port interrupts, this can be done by writing to the IntClr registers.

ISR processing inside a FreeRTOS Task

It is a popular scheme to have an ISR quickly exit, and then resume a task or thread to process the event. For example, if we wanted to write a file upon a button press, we don't want to do that inside an ISR because it would take too long and block the system. What we can utilize a **wait on semaphore** design pattern.

What you may argue with the example below is that we do not process the ISR immediately, and therefore delay the processing. But you can tackle this scenario by resuming a HIGHEST priority task. Immediately, after the ISR exits, due to the ISR "yield", FreeRTOS will resume the high priority task immediately rather than servicing another task

```
/* Create the semaphore in main() */
SemaphoreHandle_t button_press_semaphore = NULL;
void myButtonPressISR(void)
{
    long yield = 0;
    xSemaphoreGiveFromISR(button_press_semaphore, &yield);
    portYIELD_FROM_ISR(yield);
}
void vButtonPressTask(void *pvParameter)
{
    while(1)
    {
        if (xSemaphoreTake(button_press_semaphore, portMAX_DELAY))
        {
            /* Process the interrupt */
        }
    }
}
void main(void)
{
    button_press_semaphore = xSemaphoreCreateBinary();
    /* TODO: Hook up myButtonPressISR() using eint.h */
    /* TODO: Create vButtonPressTask() and start FreeRTOS scheduler */}
```

Code Block 6. Wait on Semaphore ISR design pattern example

Resources

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0489b/CACDDJHB.html

Interrupts Lab Assignment

Objective

To learn how to create a single dynamic user defined interrupt service routine callback driver/library.

This lab will utilize:

- Lookup table structures
- Function pointers
- Constexpr and static
- Interrupts
  - LPC40xx MCU gpio supports rising and falling edge interrupts on certain pins on port 0 and 2
  - These port/pin interrupts are actually OR'd together and use a single CPU interrupt called GPIO_IRQ

Port Interrupts

You will configure GPIO interrupts. This is supported for Port 0 and Port 2 and the following registers are relevant.

Table 103. GPIO interrupt register map

Generic Name	Description	Access	Reset value[1]	PORTn Register Name & Address
IntEnR	GPIO Interrupt Enable for Rising edge.	R/W	0	IO0IntEnR - 0x4002 8090 IO2IntEnR - 0x4002 80B0
IntEnF	GPIO Interrupt Enable for Falling edge.	R/W	0	IO0IntEnR - 0x4002 8094 IO2IntEnR - 0x4002 80B4
IntStatR	GPIO Interrupt Status for Rising edge.	RO	0	IO0IntStatR - 0x4002 8084 IO2IntStatR - 0x4002 80A4
IntStatF	GPIO Interrupt Status for Falling edge.	RO	0	IO0IntStatF - 0x4002 8088 IO2IntStatF - 0x4002 80A8
IntClr	GPIO Interrupt Clear.	WO	0	IO0IntClr - 0x4002 808C IO2IntClr - 0x4002 80AC
IntStatus	GPIO overall Interrupt Status.	RO	0	IOIntStatus - 0x4002 8080

Assignment

Part 0: Simple Interrupt

The first thing you want to do is get a single Port/Pin's interrupt to work.

```
void GpioInterruptCallback()
{
  // 4) For the callback, do anything such as printf or blink and LED here to test your ISR
  // 5) MUST! Clear the source of the GPIO interrupt
}
void main(void)
{
  // 1) Setup a GPIO on port 2 as an input
  // 2) Configure the GPIO registers to trigger an interrupt on P2.0 rising edge.
  // 3) Register your callback for the GPIO_IRQn
  RegisterIsr(GPIO_IRQn, GpioInterruptCallback);

  while (1)
  {
    continue; // Empty loop just to test the interrupt
  }}
```

Code Block 1. Basic Interrupt Test

Part 1: Extend the **LabGPIO** driver

You are designing a library that will allow the programmer using your library to be able to "attach" a function callback to any and each pin on port 0 or port 2.

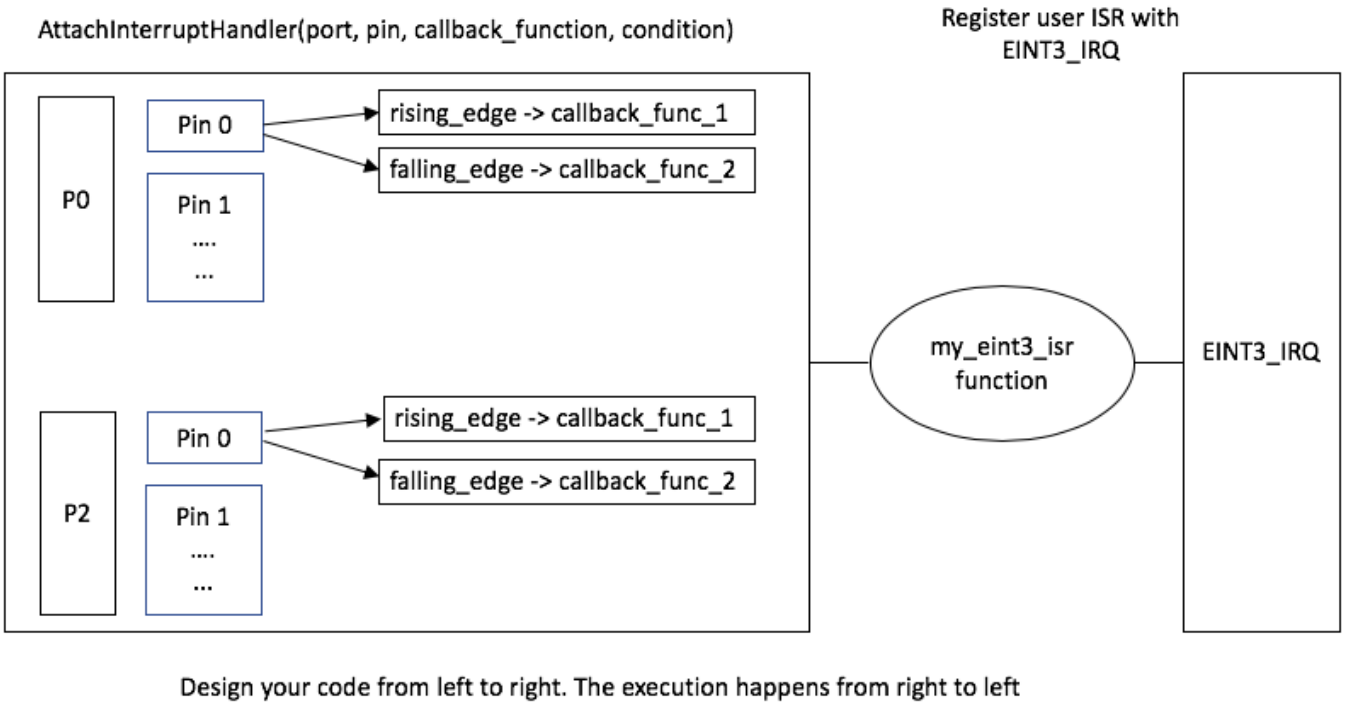
1. Add and implement ALL class methods.
2. All methods must function work as expected by their comment description.

```
#pragma once
// Gives you access to
```

```
#include "L0_LowLevel/interrupts.hpp"
class LabGPIO
{
 public:
  enum class Edge
  {
    kNone = 0,
    kRising,
    kFalling,
    kBoth
  };
  static constexpr size_t kPorts = 2;
  static constexpr size_t kPins = 32;
  // This handler should place a function pointer within the lookup table for
  // the GpioInterruptHandler() to find.
  //
  // @param isr  - function to run when the interrupt event occurs.
  // @param edge - condition for the interrupt to occur on.
  void AttachInterruptHandler(IsrPointer isr, Edge edge);
  // Register GPIO_IRQn here
  static void EnableInterrupts();
 private:
  // Statically allocated a lookup table matrix here of function pointers
  // to avoid dynamic allocation.
  //
  // Upon AttachInterruptHandler(), you will store the user's function callback
  // in this matrix.
  //
  // Upon the GPIO interrupt, you will use this matrix to find and invoke the
  // appropriate callback.
  //
  // Initialize everything to nullptr.
  static IsrPointer pin_isr_map[kPorts][kPins] = { nullptr };
  // This function is invoked by NVIC via the GPIO peripheral asynchronously.
  // This ISR should do the following:
  //  1) Find the Port and Pin that caused the interrupt via the IO0IntStatF,
```

```
  //     IO0IntStatR, IO2IntStatF, and IO2IntStatR registers.
  //  2) Lookup and invoke the user's registered callback.
  //
  // VERY IMPORTANT!
  //  - Be sure to clear the interrupt flag that caused this interrupt, or this
  //    function will be called repetitively and lock your system.
  //  - NOTE that your code needs to be able to handle two GPIO interrupts
  //    occurring at the same time.
  static void GpioInterruptHandler();
};
// ...
int main(void)
{
  // This is just an example, use which ever pins and ports you like
  Gpio gpio(2, 3);
  gpio.EnableInterrupts();
  while(true)
  {
    continue;
  }
  return 0;}
```

Code Block 2. GPIO Interrupt Driver Template Class



Design your code from left to right. The execution happens from right to left

Requirements

- Should be able to specify a callback function for any port/pin for an exposed GPIO given a rising, falling, or both condition.
  - We may ask you to change which port and pin causes a particular callback to be executed in your code and then recompile and re-flash your board to and prove it works with any port 0 or port 2 pin.
- You will need to use two external switches for this lab.

> Note that printing 4 chars inside an ISR can take 1ms, and this is an eternity for the processor and should never be done, unless other than debug.

What to turn in:

- Place all relevant source files within a .pdf file.
- Turn in the **screenshots** of terminal output.