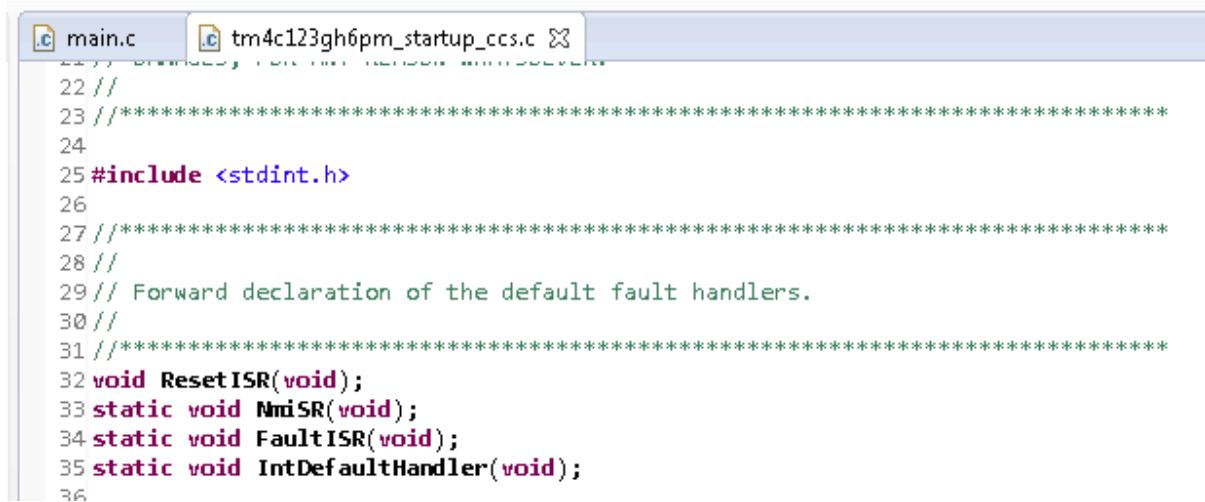


Viewing the tm4c123gh6pm_startup_ccs.c file, the first piece of code we see after the comments is shown in Figure 1.



```
22 //
23 //*****
24
25 #include <stdint.h>
26
27 //*****
28 //
29 // Forward declaration of the default fault handlers.
30 //
31 //*****
32 void ResetISR(void);
33 static void NmiISR(void);
34 static void FaultISR(void);
35 static void IntDefaultHandler(void);
36
```

Figure 1 Forward declaration for the default handlers

These are the forward declarations, or function prototypes, for the three high-priority exceptions (**ResetISR**, **NmiISR** and **FaultISR**), and a default Interrupt Service Routine (ISR) – **IntDefaultHandler** for the other interrupt sources. Taking each of these in turn...

Line 32.

void ResetISR(void);

This is the highest priority exception (priority -3 in the tm4c123gh6pm datasheet, p103, Table 2-8, "Exception types"). In the event of power-up or a reset, **ResetISR** is invoked, by virtue of its position at the start of the vector table (see later). The code listing for **ResetISR** is shown in Figure 2:

```
229 // This is the code that gets called when the processor first starts execution
230 // following a reset event. Only the absolutely necessary set is performed,
231 // after which the application supplied entry() routine is called. Any fancy
232 // actions (such as making decisions based on the reset cause register, and
233 // resetting the bits in that register) are left solely in the hands of the
234 // application.
235 //
236 //*****
237 void
238 ResetISR(void)
239 {
240     //
241     // Jump to the CCS C initialization routine. This will enable the
242     // floating-point unit as well, so that does not need to be done here.
243     //
244     __asm(".global _c_int00\n"
245           "      b.w      _c_int00");
246 }
247
```

Figure 2 **ResetISR** code listing

This is the first (although not last) instance of code whose precise function is not clear to me at the moment. The **__asm** keyword invokes the inline assembler, indicating that the code in brackets is written in assembly language rather than C. My attempts to research this would seem to indicate that it has something to do with loading low-level code from the micro ROM into SRAM ("bootloading"), however at that point my understanding ends - and even that might be incorrect. Although I would like to know what this code actually *does*, for the moment I am content to assume that it's there for a good reason and is doing whatever it needs to do.

Line 33

static void NmiSR(void);

The non-maskable interrupt (NMI) is the next highest priority exception with a priority of -2, and the second entry in the vector table. An NMI is generally an exception/interrupt which for whatever reason should not be ignored or “masked”. The code for **NmiSR** is shown in Figure 3:

```
248 //*****
249 //
250 // This is the code that gets called when the processor receives a NMI. This
251 // simply enters an infinite loop, preserving the system state for examination
252 // by a debugger.
253 //
254 //*****
255 static void
256 NmiSR(void)
257 {
258     //
259     // Enter an infinite loop.
260     //
261     while(1)
262     {
263     }
264 }
265
```

Figure 3 **NmiSR** code listing

As the comments indicate, this exception handler does nothing other than put the micro into an infinite **while** loop. The programmer is free to write his or her **NmiSR** handler in place of the existing default handler.

This is also the first instance of the **static** keyword, with **static** being an example of a C “[storage class](#)”. I *think* the use of the keyword **static** in this function definition limits the function to being callable *only* within the startup file and nowhere else. In other words, declaring the **NmiSR** function as **static** hides the function from **main** or any code module. The only reason I can think of that one might do this is to prevent the exception handler from being called, perhaps accidentally, from another code module. Another point... if this is indeed true, does it mean that the “non-**static**” exception handler **ResetISR** can be called from other modules...?

Line 34

static void FaultISR(void);

This is the exception handler which is called due to a “Hard Fault” where Hard Fault is defined as “...an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.” The code for **FaultISR** is shown in Figure 4

```
266 //*****
267 //
268 // This is the code that gets called when the processor receives a fault
269 // interrupt. This simply enters an infinite loop, preserving the system state
270 // for examination by a debugger.
271 //
272 //*****
273 static void
274 FaultISR(void)
275 {
276     //
277     // Enter an infinite loop.
278     //
279     while(1)
280     {
281     }
282 }
283
```

Figure 4 **FaultISR** code listing

FaultISR has a priority of -1 and is the lowest priority of the fixed-priority exception handlers, third entry in the vector table. This code is very similar to that for the **NmiISR** handler, simply putting the micro into an infinite loop. Again, the programmer is free to modify this code as required, and the handler can only be “seen” from within the startup file.

Line 35

```
static void IntDefaultHandler(void);
```

Unlike **ResetISR**, **NmiISR** and **FaultISR**, **IntDefaultHandler** is a general default handler for the remaining 86 exceptions and interrupts not already explicitly handled. Unless a given exception or interrupt has its ISR overwritten by the programmer, it will simply invoke the **IntDefaultHandler** code shown in Figure 5:

```
284 //*****
285 //
286 // This is the code that gets called when the processor receives an unexpected
287 // interrupt. This simply enters an infinite loop, preserving the system state
288 // for examination by a debugger.
289 //
290 //*****
291 static void
292 IntDefaultHandler(void)
293 {
294     //
295     // Go into an infinite loop.
296     //
297     while(1)
298     {
299     }
300 }
301
```

Figure 5 **IntDefaultHandler** code listing

Again, this is the familiar infinite-loop code which we have seen in the **NmiISR** and **FaultISR** handlers. In typical usage, which would involve the micro taking action following on from one of more exceptions or interrupts, the particular exception or interrupt would have its own corresponding handler/ISR.

Line 43.

After the function prototypes for **ResetISR**, **NmiISR**, **FaultISR** and **IntDefaultHandler** we next have this code (Figure 6):

```
--  
37 //*****  
38 //  
39 // External declaration for the reset handler that is to be called when the  
40 // processor is started  
41 //  
42 //*****  
43 extern void _c_int00(void);  
44
```

Figure 6 **_c_int100** declaration

This is a declaration or prototype for the function **_c_int00**, which is declared externally to the startup code, as dictated by the **extern** keyword. The **_c_int00** function is provided by the linker and called by the **ResetISR** function described above. This is part of the startup (or boot) routine and is described in Section 3.3.1 of the CCS Help system:

*“...The function **_c_int00** is the startup routine (also called the boot routine) for C/C++ programs. It performs all the steps necessary for a C/C++ program to initialize itself.*

*The name **_c_int00** means that it is the interrupt handler for interrupt number 0, RESET, and that it sets up the C environment. Its name need not be exactly **_c_int00**, but the linker sets **_c_int00** as the entry point for C programs by default. The compiler's run-time-support library provides a default implementation of **_c_int00**.”*

This is consistent with what we have seen so far: **_c_int00** is called by **ResetISR**, which is the handler which runs after a hard reset. Further detailed information regarding **_c_int00** seems a little hard to come by, although the question is [often asked](#). For the moment I am content that it exists and does what it needs to do without further investigation.

Line 50.

This is a declaration of an externally-defined symbolic constant, `__STACK_TOP` (Figure 7)

```
45 //*****  
46 //  
47 // Linker variable that marks the top of the stack.  
48 //  
49 //*****  
50 extern uint32_t __STACK_TOP;  
51
```

Figure 7 `__STACK_TOP` declaration

This constant is defined by the linker and is used in the definition of the vector table, beginning at Line 66.

The remainder of the startup file defines the vector table and begins at Line 66. This code section is quite lengthy, running to over 150 lines of code, however the general construction and function of the vector table can be demonstrated through viewing the first 30 or so lines as shown in Figure 8:

```

59 //*****
60 //
61 // The vector table. Note that the proper constructs must be placed on this to
62 // ensure that it ends up at physical address 0x0000.0000 or at the start of
63 // the program if located at a start address other than 0.
64 //
65 //*****
66 #pragma DATA_SECTION(g_pfnVectors, ".intvecs")
67 void (* const g_pfnVectors[])(void) =
68 {
69     (void (*)(void))((uint32_t)&__STACK_TOP),
70     // The initial stack pointer
71     ResetISR, // The reset handler
72     NmiISR, // The NMI handler
73     FaultISR, // The hard fault handler
74     IntDefaultHandler, // The MPU fault handler
75     IntDefaultHandler, // The bus fault handler
76     IntDefaultHandler, // The usage fault handler
77     0, // Reserved
78     0, // Reserved
79     0, // Reserved
80     0, // Reserved
81     IntDefaultHandler, // SVCcall handler
82     IntDefaultHandler, // Debug monitor handler
83     0, // Reserved
84     IntDefaultHandler, // The PendSV handler
85     IntDefaultHandler, // The SysTick handler
86     IntDefaultHandler, // GPIO Port A
87     IntDefaultHandler, // GPIO Port B
88     IntDefaultHandler, // GPIO Port C
89     IntDefaultHandler, // GPIO Port D
90     IntDefaultHandler, // GPIO Port E
91     IntDefaultHandler, // UART0 Rx and Tx
92     IntDefaultHandler, // UART1 Rx and Tx
93     IntDefaultHandler, // SSI0 Rx and Tx
94     IntDefaultHandler, // I2C0 Master and Slave
95     IntDefaultHandler, // PWM Fault
96     IntDefaultHandler, // PWM Generator 0
97     IntDefaultHandler, // PWM Generator 1
98     IntDefaultHandler, // PWM Generator 2
99     IntDefaultHandler, // Quadrature Encoder 0
100    IntDefaultHandler, // ADC Sequence 0
101    IntDefaultHandler, // ADC Sequence 1
102    IntDefaultHandler, // ADC Sequence 2

```

Figure 8 Vector table, partial code listing

The first four lines of this section are a little complex, and I'm not sure I fully understand what they are doing, but here goes...

Line 66.

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
```

#pragma is a pre-processor directive whose meaning is specific to the particular implementation of compiler in use. In the case of CCS, **#pragma DATA_SECTION** is described in the CSS Help file as follows:

“The DATA_SECTION pragma allocates space for the symbol in C, in a section named *section name*.

The syntax of the pragma in C is:

```
#pragma DATA_SECTION (symbol, "section name")
```

So Line 66 creates a section to be incorporated in the assembly process with the name “.intvecs” from the array g_pfnVectors, which is defined in Lines 67 onwards.

Line 67.

```
void (* const g_pfnVectors[])(void) =
```

The square brackets [and] indicate that this is the definition of an array named g_pfnVectors. An array is a contiguous group of data items of the same name and type. The asterisk * dictates that this is also an array of “pointers”, or memory locations, to a group of functions. **void** and **(void)** determine that those functions neither accept nor return any parameters. The **const** keyword is a type qualifier which prevents the array from being modified.

So this definition creates an array “g_pfnVectors” of pointers to memory locations for an array of functions which neither receive nor return any parameters. Those functions are the various exception and interrupt handlers.

Line 69.

```
(void (*)(void))((uint32_t)&_STACK_TOP)
```

The exact working of this line is unclear to me at this time, but the purpose is clearly defined by the accompanying comment. This code inserts the address of the stack (the “stack pointer”) in the first vector table location. At power-up the micro loads the stack pointer from the first location in the vector table and so, even before the reset handler **ResetISR** is invoked, the address of the stack is defined. This is necessary because it is possible that, immediately after **ResetISR** has run, the micro could be required to service an exception or interrupt handler. If that were to happen, the stack would be needed in order to preserve the present state of the micro. Had the stack not been defined, then the handler could not be run. Hence the need to make the stack definition the very first thing that the micro does after reset.

Line 71+

The remainder of the vector table is defined from Lines 71 onwards. Each entry in the vector table occupies its position in memory by virtue of its position in this list. So after the stack pointer, the next entry is ResetISR, then NmiSR, and so on. This is consistent with Figure 2.6, Vector Table in the TM4C123GH6PM data sheet (Figure 9).

The default entries in the vector table can be modified by simply replacing the name of the default handler with one of a custom handler which may have been written to deal with a particular eventuality. So for instance if the programmer has written a handler named “GPIOPortAIntHandler” to handle the GPIO Port A interrupt (line 66 of Figure 8) then the IntDefaultHandler entry on this line would simply be changed to GPIOPortAIntHandler. This is exactly the procedure detailed in Lab 4 of the [Workbook](#), which I will be examining shortly.

Figure 2-6. Vector Table

Exception number	IRQ number	Offset	Vector
154	138	0x0268	IRQ131
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCcall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Figure 9 Vector table illustrated in the TM4C123GH6PM data sheet