Title: The arm Cortex M4, tools and debugging
Author: Craig Duffy 23/09/21, 13/10/22 16/11/22 29/08/24
Module: Embedded Systems Programming
Awards: BSc Computer Science
Prerequisites: Some Linux skills, and basic computer architecture

This worksheet introduces you to the arm cortex M4 CPU. make and Makefiles are introduced, and we look at various compiler options and tools. You will also understand something of the Cortex memory layout and booting. Finally, we will look at debugging code using gdb-multiarch.

The arm Cortex-M4 processor

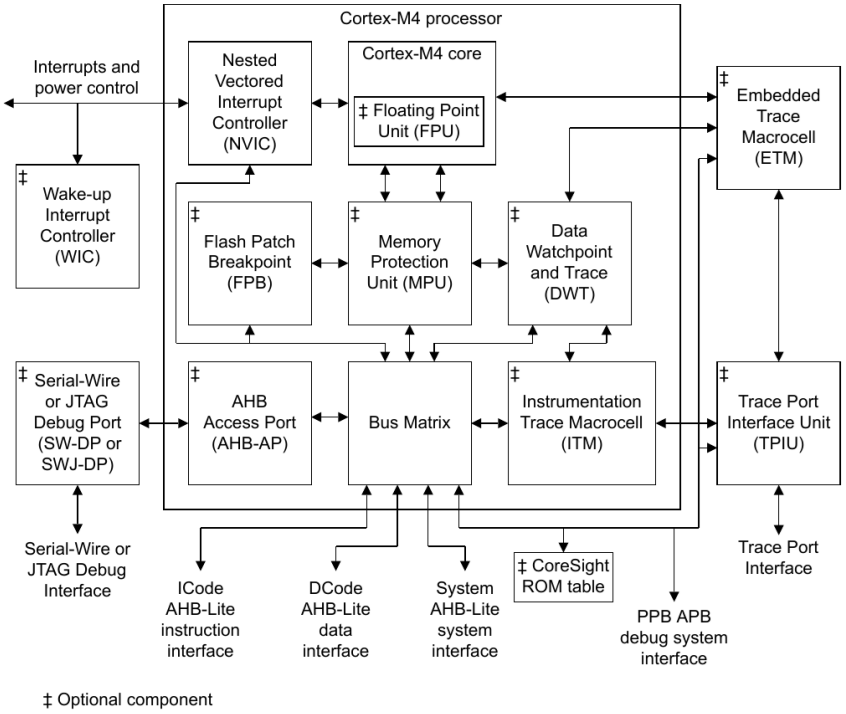
Before we look a little at the CPU, we need to understand a little about arm and their business model. arm design CPUs but don't actually make them – in this way they differ from most of the other CPU firms – Intel, Nvidia ETC. arm license their designs to other firms who then make chips, these chips can then be used in various boards, sometimes by the company making the chip sometimes by a third party. The chip we will be using is the STM32L47x series made by STMicro, they also made the board we are using, the discovery (AKA disco) IoTboard.

arm, as a company found a niche in low powered CPUs especially during the growth in mobile phones around the late 1990s to the early 2000s. The earlier designs, many of which are still in active use, were a little ad-hoc. From the mid 2000s arm introduced 3 new processor types – the Cortex A, R and M. The A series, for application processors, is the type of CPU found in modern mobile phones – it has a 64-bit instruction set, MMU and can run complex OSs such as Windows or Linux. The R series is for real-time operations and is used more in military, avionics and medical applications. The M series, microcontroller, is more for lower end applications with a 32-bit instruction set, rather more limited memory and access control, and is used to run either without or with a more limited OS.

The aim of this more standardised approach was to allow developers to move code from one core to another with the least problems. Obviously running Cortex-M3 code on an M4 would not be able to access some of the extra features, such as an FPU, but could none the less execute and would be easier to modify. The memory maps between the various models are forwards compatible and thus better to work with.

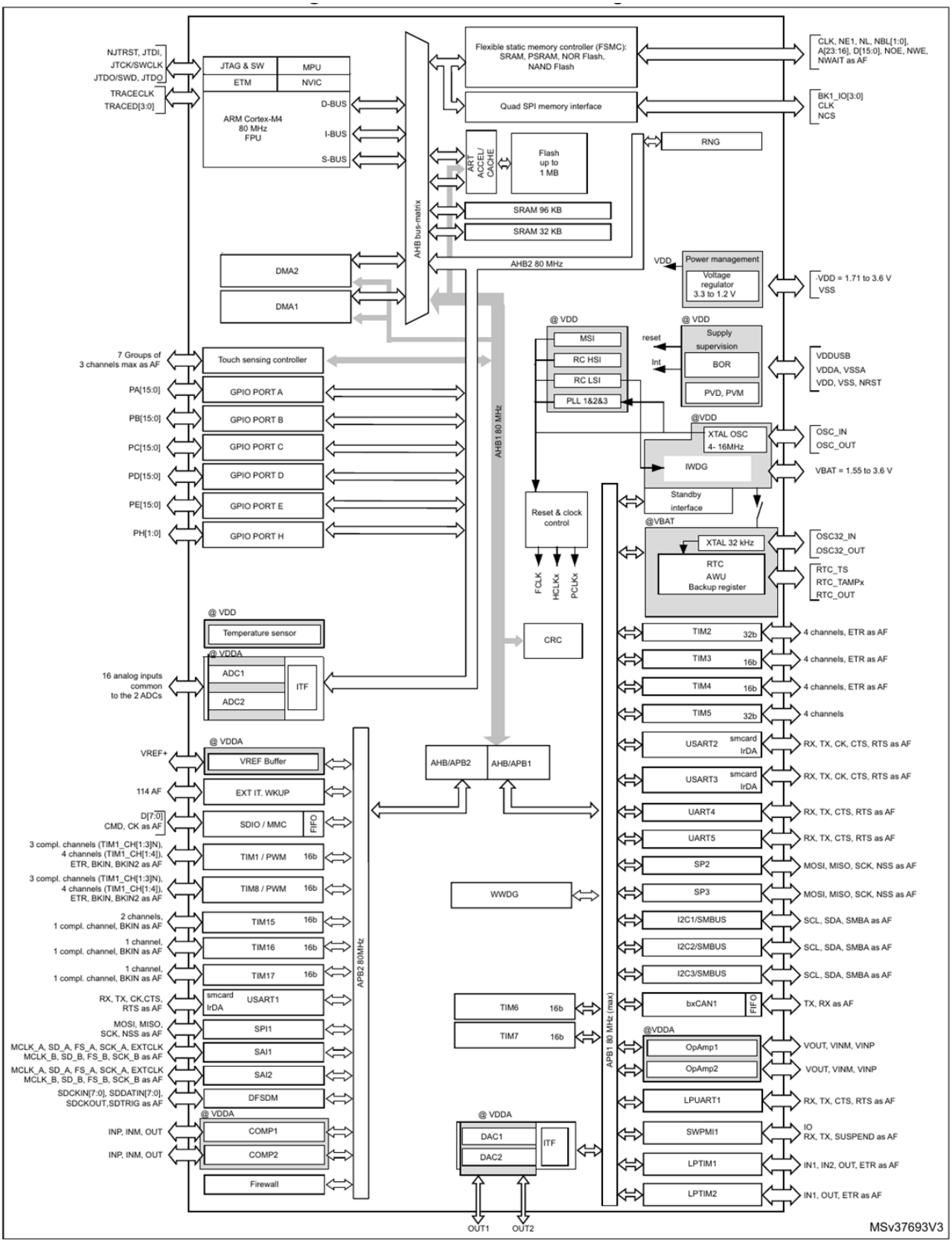
The Cortex-M4

The diagram below shows the various components of an M4 processor. In later lectures and worksheets we will focus upon things such as the Nested Vectored Interrupt Controller, when we work on interrupts. If we get to work on porting an operating system, FreeRTOS, then we will look at the Memory Protection Unit in more detail. We have already used the JTAG port in accessing the board via openocd.



Block diagram of the arm Cortex-M4

The next diagram shows some of the function interconnections of the M4. Be aware that we won't be using all of these, and that some may not be implemented on our board. What it does illustrate is the breadth and depth of the interconnections and protocols available to the developer using this device and board. In our work we will look in detail at some of these peripherals and learn how to program them.



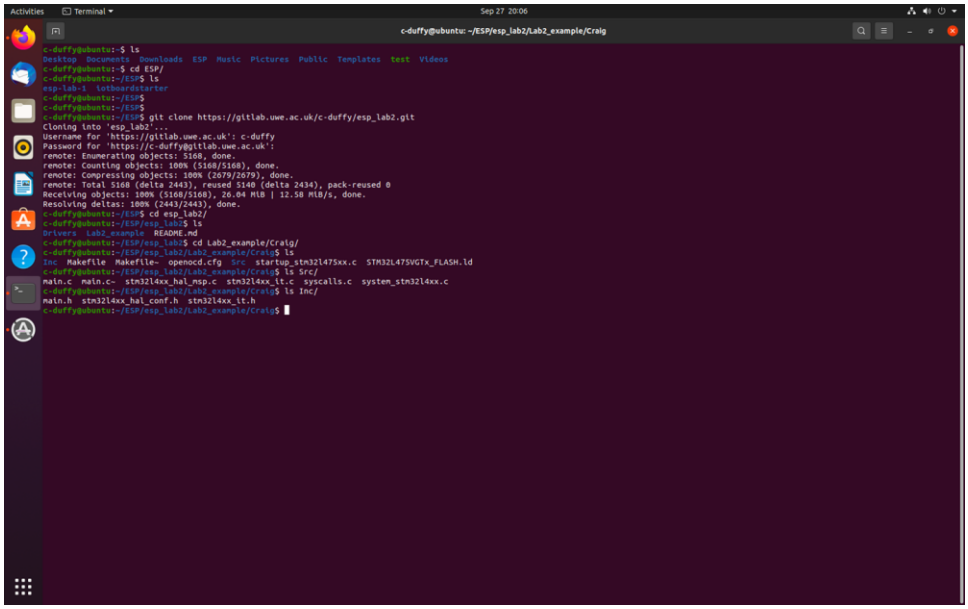
Note: AF: alternate function on I/O pins.

Functionality of arm Cortex-M4

Building your first program

Clone the repo for this worksheet – https://gitlab.uwe.ac.uk/c-duffy/esp_lab2.git you won't need to fork it as again we won't be saving any changes. If you have any problems cloning the repo refer to the lab 1 worksheet for information.

Once you have cloned the repo *cd* into the esp-lab2 directory. You will notice there are 2 sub directories – Drivers and Lab2_example. The Drivers directory contains 2 further sub directories of specific device drivers for the Cortex-M4 and the disco board. The BSP directory contains some low level code to control the various sensors and I/O of the board. The CMSIS directory is essentially the BIOS for the M4 chip itself – CMSIS is arm's name for a BIOS. Finally, there is the STM32L4xx_HAL_Driver directory which has the hardware application layer (HAL) code for the board – so there are drivers for all the protocols and sub-systems – I2C, SPI, timers and so on. As we go through the labs we will look in more detail at the code in these sub-directories.



The repo and contents

The Lab2 directory contains a two sub directories called Src and Inc , which houses all the source code we will require for this worksheet. Also in the Lab2 directory are some other files: openocd.cfg (you should know that this one is for!), Makefile, STM32L475VGTx_FLASH.ld, startup_stm32l475xx.c. Ignore any file with ~ as part of their name – they are backup files created by the emacs editor. The Src directory has: main.c stm32l4xx_hal_msp.c stm32l4xx_it.c syscalls.c system_stm32l4xx.c and finally, Inc has: main.h stm32l4xx_hal_conf.h stm32l4xx_it.h.

Make and Makefiles

We need to understand **make** and Makefiles before we go much further. The directories described above have all of the code required to build a simple application on our board, the problem is to build that code in a manner that can be executed on the board. make and the Makefile manages this process for us. make, as the manual says, 'will determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.' In fact, make isn't only used for programs but can be used for web pages, databases, script or whatever.

You can think of make as the chef and the Makefile the recipe, and like all recipes it has to be written by someone. The Makefile is given in appendix A for you to refer to. One of the first things in it is the directive

```
ELF=demo.elf
```

This is specifying that the 'target' of the Makefile is to create a file called demo.elf. Then we get a specification of a toolroot, library paths and tools. This is because we are doing cross development – we can't use the standard compiler or libraries – they will generate or be made from 64-bit Pentium assembler and also be assuming that Linux is running the show – our board is a 32-bit arm CPU without an operating system. In fact, you will notice that the CC variable (C Compiler) is set to **arm-none-eabi-gcc**. This is indicating that it is an arm compiler, with no OS support and using the **e**xtended **a**rm **b**inary **i**nterface. The standard compiler on Linux, gcc, is actually a link to a file called **x86_64-linux-gnu-gcc-9**, meaning it is a 64-bit Pentium, running on Linux using the gnu calling convention and is version 9 – phew!

The section between *#Processor and board specific settings* and *#Build executable* contains all of the various settings that the compiler and other tools need to build for our board. The C compiler settings includes things such as *–mcpu=cortex-m4* which is specifying the Cortex-M4 CPU – gcc has lots (hundreds?) of settings, just have a look at man gcc to get an idea of how many there are.

The actual work is done after the *#Build executable* line, and this gives the 'real' target of the Makefile in the following rule:

```
$(ELF) : $(OBJS)
```

On the left-hand side of the : is the target and on the right-hand side the prerequisite. So once the $(variables) have been resolved this means that to get a target of demo.elf you need to have this long list of .o dependencies – from startup_stm32l475xx.o through to main.o. If these files exist then make will execute the command

```
$(LD) $(LDFLAGS) -o $@ $(OBJS)
```

Which will resolve into the **arm-none-eabi-gcc** command, which is being used as a linker, followed by the various flags and the object files. This will create the executable binary, demo.elf. However, you might notice that the first time you run make there are no .o files. So what does make do? If it comes to a problem like this, make runs through its other set of rules with their targets and prerequisites, such as the following rule

```
%.o: %.c
```

This tells make that if you have a target of something.o (%.o) and you have a prerequisite of something.c (%.c) then do the following

```
$(CC) -c $(CFLAGS) $< -o $@
```

This will translate into executing the **arm-none-eabi-gcc** command with the set of parameters specified in the previous section. If

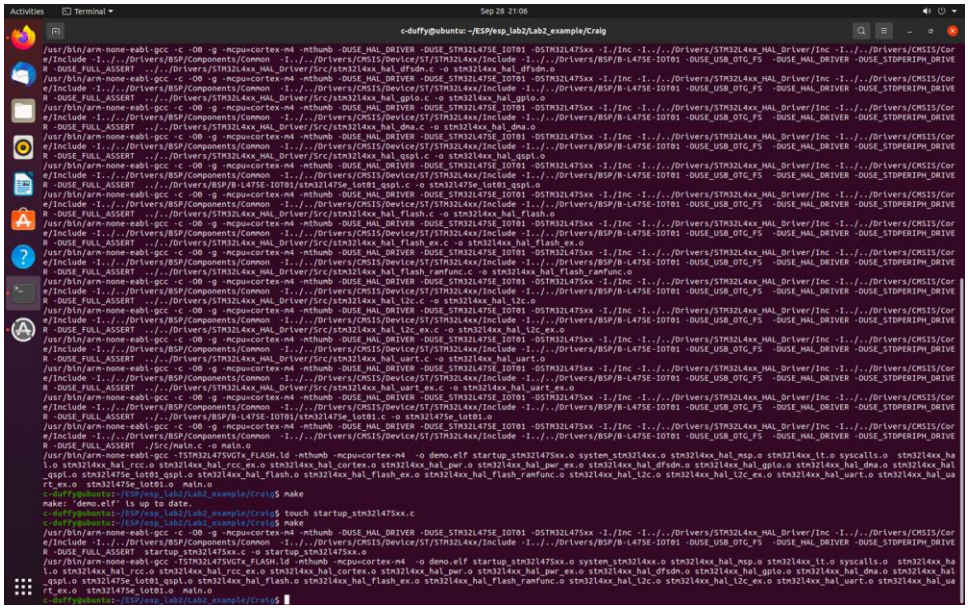
```
demo.elf : startup_stm32l475xx.o
```

Couldn't be resolved then make would look for a startup_stm32l475xx.c, which it can find and then execute, doing the following

```
/usr/bin/arm-none-eabi-gcc -c -O0 -g -mcpu=cortex-m4 -mthumb -DUSE_HAL_DRIVER -DUSE_STM32L475E_IOT01 -DSTM32L475xx -I./Inc -I../../Drivers/STM32L4xx_HAL_Driver/Inc -I../../Drivers/CMSIS/Core/Include -I../../Drivers/BSP/Components/Common -I../../Drivers/CMSIS/Device/ST/STM32L4xx/Include -I../../Drivers/BSP/B-L475E-IOT01 -DUSE_USB_OTG_FS -DUSE_HAL_DRIVER -DUSE_STDPERIPH_DRIVER -DUSE_FULL_ASSERT startup_stm32l475xx.c -o startup_stm32l475xx.o
```

This would, assuming the code in the C file is correct, create the .o file thus allowing make to process the next item in the list of object files ($(OBJS)).

Thus make saves us from having to type all of these complex commands in and if we add new files, we just have to add them to the list of object prerequisites. make also can manage changes in files. Look at the screen below



Running make

The first make command, which creates all the output, has done its job, and when make is run again it reports 'demo.elf' is up to date. Namely, nothing to be done here. Next the touch command is run on the startup_stm32l475xx.c file. This resets the modification time of the file, so when make is run again it notices that startup_stm32l475xx.c is newer than startup_stm32l475xx.o and therefore demo.elf. make assumes that the changed modification time means the file has been changed and it rebuilds the .o file and then relinks it with the other objects to remake demo.elf. This is really useful when rebuilding big developments – like the Linux kernel for example, as it means rather than rebuild everything just build the changes.

What is an ELF?

You will have noticed the word elf popping up in various places in this worksheet. It doesn't refer to mythical woodland creature, sadly, but is an acronym. It stands for Executable Linkable Format (in some places Executable Loadable Format), it is a standard for binary files that allows all the code and data to be represented in a portable and easy to manage way, it is especially useful for dynamic linking at run time. It is used on a large number of operating systems and it originated in the UNIX world. So the file demo.elf stores all of the code, data and loading information that is

required for the program to be executed on our disco board. Later we will see exactly how this file is created.

If you look at our main.c you will notice that as well as being non-descript and pointless there is nothing specific to our board.

```
#include <stdio.h>

int i = 0;
int off = 5;

void inc(void){
i += off;
}

int main(void)
{
printf("Welcome to the ESP Lab 1 test program \n");
printf("this program does next to nothing!\n");
/* Infinite loop */
while (1) {
inc();
}
}
```

So it can be compiled using the native gcc, try the command *gcc -static –o demo.x86 Src/main.c*. This will create a Pentium 64-bit version of the program that can run on Linux. You can run from the command line by typing *./demo.x86* - you will need to terminate it with a ctrl-c character as it goes into an infinite loop. If you type in the command *file demo.elf main.x86* you should see the following output

demo.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, with debug_info, not stripped
main.x86: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=57627ebc7eb779c3a90a084a19b335ab64993a48, for GNU/Linux 3.2.0, not stripped

Both files are executable, but demo is a 32-bit ARM and main is 64-bit Pentium, you will also note that main knows that it is running on Linux and using its calling conventions.

The final demo.elf is made up of a series of object files 0- these are also ELF files but they are reported as relocatable rather than executable by file. So *file main.o* reports

main.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), with debug_info, not stripped

This means that it is legitimate arm assembler but is not yet located into a memory area nor linked with other code that will help it run. We can see this by using the command objdump to disassemble the object file. Typing *arm-none-eabi-objdump –D main.o | more* will display the created code.

```
main.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <inc>:
   0:   b480            push    {r7}
   2:   af00            add     r7, sp, #0
   4:   4b05            ldr     r3, [pc, #20]   ; (1c <inc+0x1c>)
   6:   681a            ldr     r2, [r3, #0]
   8:   4b05            ldr     r3, [pc, #20]   ; (20 <inc+0x20>)
   a:   681b            ldr     r3, [r3, #0]
   c:   4413            add     r3, r2
   e:   4a03            ldr     r2, [pc, #12]   ; (1c <inc+0x1c>)
  10:   6013            str     r3, [r2, #0]
  12:   bf00            nop
  14:   46bd            mov     sp, r7
  16:   bc80            pop     {r7}
  18:   4770            bx      lr
  1a:   bf00            nop
…

00000024 <main>:
  24:   b580            push    {r7, lr}
  26:   af00            add     r7, sp, #0
  28:   4804            ldr     r0, [pc, #16]   ; (3c <main+0x18>)
  2a:   f7ff fffe       bl      0 <puts>
  2e:   4804            ldr     r0, [pc, #16]   ; (40 <main+0x1c>)
  30:   f7ff fffe       bl      0 <puts>
  34:   f7ff fffe       bl      0 <inc>
  38:   e7fc            b.n     34 <main+0x10>
  3a:   bf00            nop
  3c:   00000000        andeq   r0, r0, r0
  40:   00000028        andeq   r0, r0, r8, lsr #32

Disassembly of section .data:
00000000 <off>:
   0:   00000005        andeq   r0, r0, r5

Disassembly of section .bss:
00000000 <i>:
   0:   00000000        andeq   r0, r0, r0
```

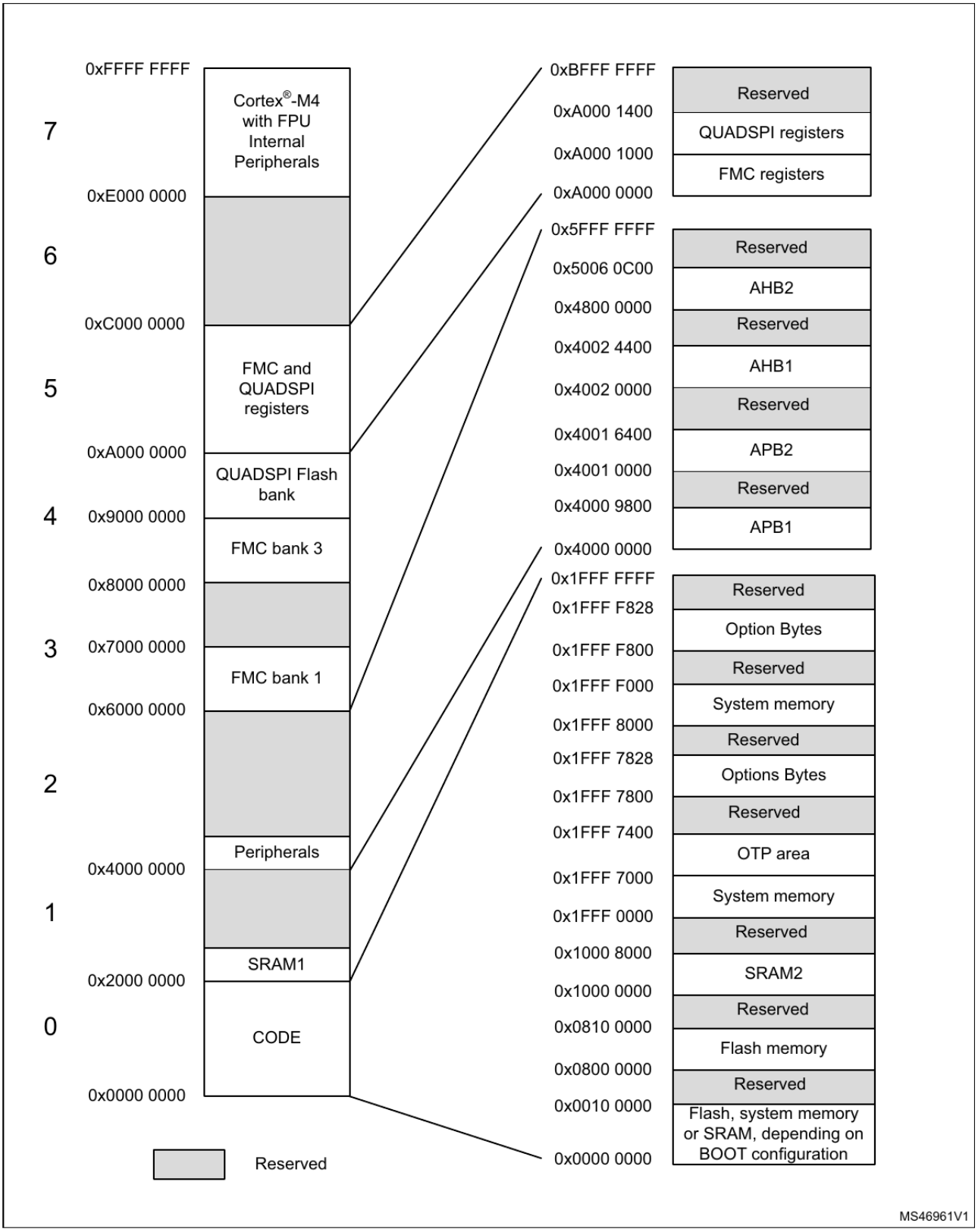
You will notice that the code seems to start at address 0 – not a great choice to execute code. Also the branches to *puts* (2a and 30) seem to be branching to address 0. Really this just means that these labels are unresolved and are awaiting linkage. Note that the last 2 sections are disassembling the initialised and uninitialized data (AKA BSS) - the *andeq* instructions are just objdump trying to make sense of the data (0 and 5) and displaying them as instructions.

The linking is done by the final command in the make sequence - which links all the object files into the demo.elf:

```
/usr/bin/arm-none-eabi-gcc -TSTM32L475VGTx_FLASH.ld -mthumb -mcpu=cortex-m4  -o demo.elf
startup_stm32l475xx.o system_stm32l4xx.o stm32l4xx_hal_msp.o stm32l4xx_it.o syscalls.o
stm32l4xx_hal.o stm32l4xx_hal_rcc.o stm32l4xx_hal_rcc_ex.o stm32l4xx_hal_cortex.o
stm32l4xx_hal_pwr.o stm32l4xx_hal_pwr_ex.o stm32l4xx_hal_dfsdm.o stm32l4xx_hal_gpio.o
stm32l4xx_hal_dma.o stm32l4xx_hal_qspi.o stm32l475e_iot01_qspi.o stm32l4xx_hal_flash.o
stm32l4xx_hal_flash_ex.o stm32l4xx_hal_flash_ramfunc.o stm32l4xx_hal_i2c.o stm32l4xx_hal_i2c_ex.o
stm32l4xx_hal_uart.o stm32l4xx_hal_uart_ex.o  stm32l475e_iot01.o  main.o
```

This is calling the linker from the gcc command, and it is using the flag -T to use the STM32L475VGTx_FLASH.ld as a linker description file. This file contains information for the linker about the layout of the device's memory map and where to place the various code sections. A stripped-down version of the linker script is in appendix B. You will notice that it creates a number of variables that are available to the C code – _estack and so on, these allow the start-up routines to initialise various segments.

Below is the memory map for the Cortex-M4, taken from the ST datasheets – this shows the placement of all the memory regions for the device. You will notice that the FLASH memory starts at address 0x0800 0000 and RAM at 0x2000 0000 – you will also see in the linker script that these addresses are passed to the linker along with the size of the memory regions. The actual memory sizes will vary from board to board.



Cortex-M4 memory map

If you look at the file *startup_stm32l475xx.c* you will notice a few important features. Firstly, you will notice that the series of variable created in the linker script are referenced and they are used in the *Reset_Handler* function to initialise the run-time code. Secondly, there is a section called *.isr_vector* which is a very large jump table which is forced into the first FLASH memory

address by the linker script. This ensures that the first 2 32-bit words of memory are the stack address (_estack) and the address of the *Reset_Handler*, which must be called first to set up the run time environment. *Reset_Handler* as well as initialising the run time also calls some system set up routines and then calls *main()* to execute our code.

```
void Reset_Handler(void) {
unsigned long *src, *dst;

src = &_sidata;
dst = &_sdata;

// Copy data initializers
while (dst < &_edata)
*(dst++) = *(src++);

// Zero bss
dst = &_sbss;
while (dst < &_ebss)
*(dst++) = 0;

SystemInit();
Hal_start();
__libc_init_array();
main();
while(1) {}
}
```

Note that if *main()* returns then *Reset_Handler* just goes into an infinite loop, on a system with an operating system control would be returned to the OS.

gdb-multiarch – remote debugging

One big problem in any development is debugging code – code is relatively straightforward to write but often very hard to debug. This is especially true for embedded systems where, unlike on a 'normal' computer there may be no or a very limited interface. Quite often flashing LEDs or beeping is the only way for the code to communicate – hence the use of beeps by BIOS code. So being able to remotely debug is really useful. Our compiler tool chain comes with a debugger as standard – **gdb** and for cross development this is now packaged in the program **gdb-multiarch**.
In order to run gdb-multiarch we need to have a server for it to connect to – often this can be the program gdbserver, however that normally assumes that you are running something like Linux on your target. Fortunately for us openocd has a gdbserver built in to its code and can act as the server we need. So before starting you will need to run openocd in your demo.elf directory, and it is also good to then run telnet to connect to the board (via openocd) to flash the demo.elf onto the board but then don't *reset run.*

Now in another terminal in the demo.elf directory, run *gdb-multiarch* from the command line on giving the *demo.elf* as an argument.

gdb-multiarch demo.elf

You should now get some output form gdb and a prompt (gdb). Now we need to connect gdb-multiarch with openocd – this is set up on the TCP port 3333, you can find this in the openocd.cfg file.

*(*gdb*) target extended-remote :3333*

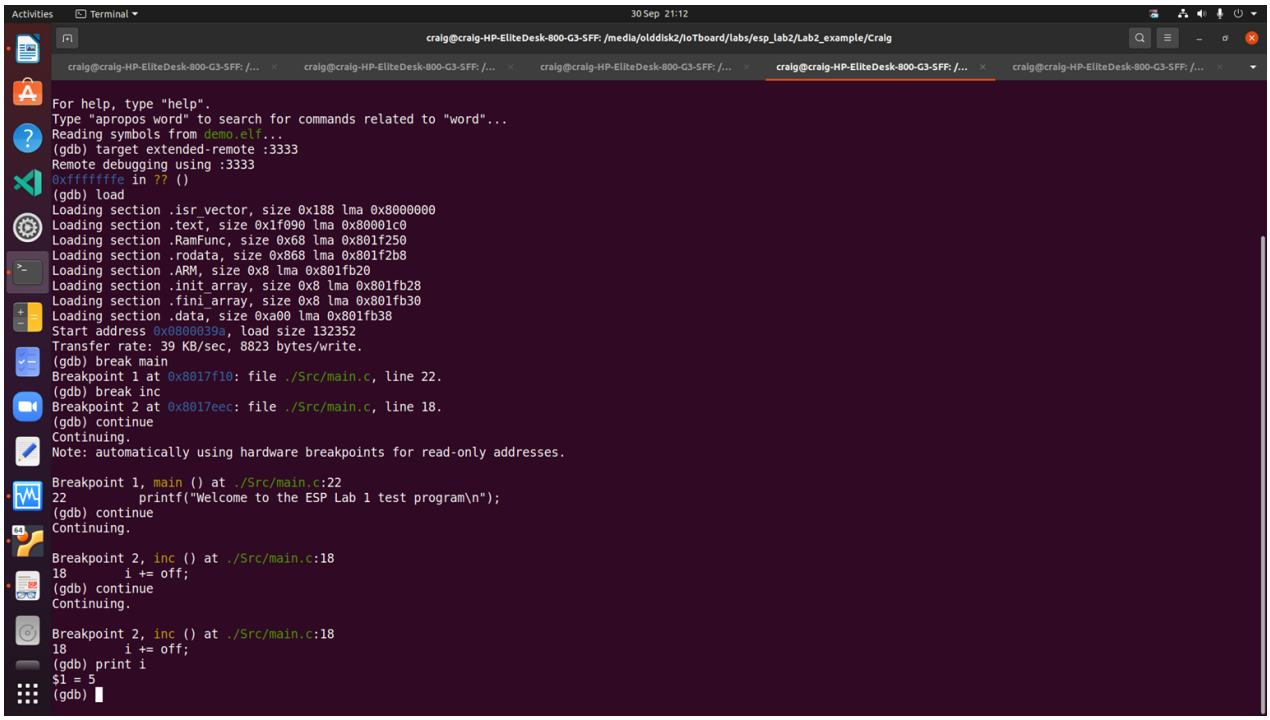
If you look at the openocd output it should register the connection.

Info : accepting 'gdb' connection on tcp/3333

Next we need to load the symbol tables and metadata into gdb-multiarch. This is done with the load command. You should receive some more information

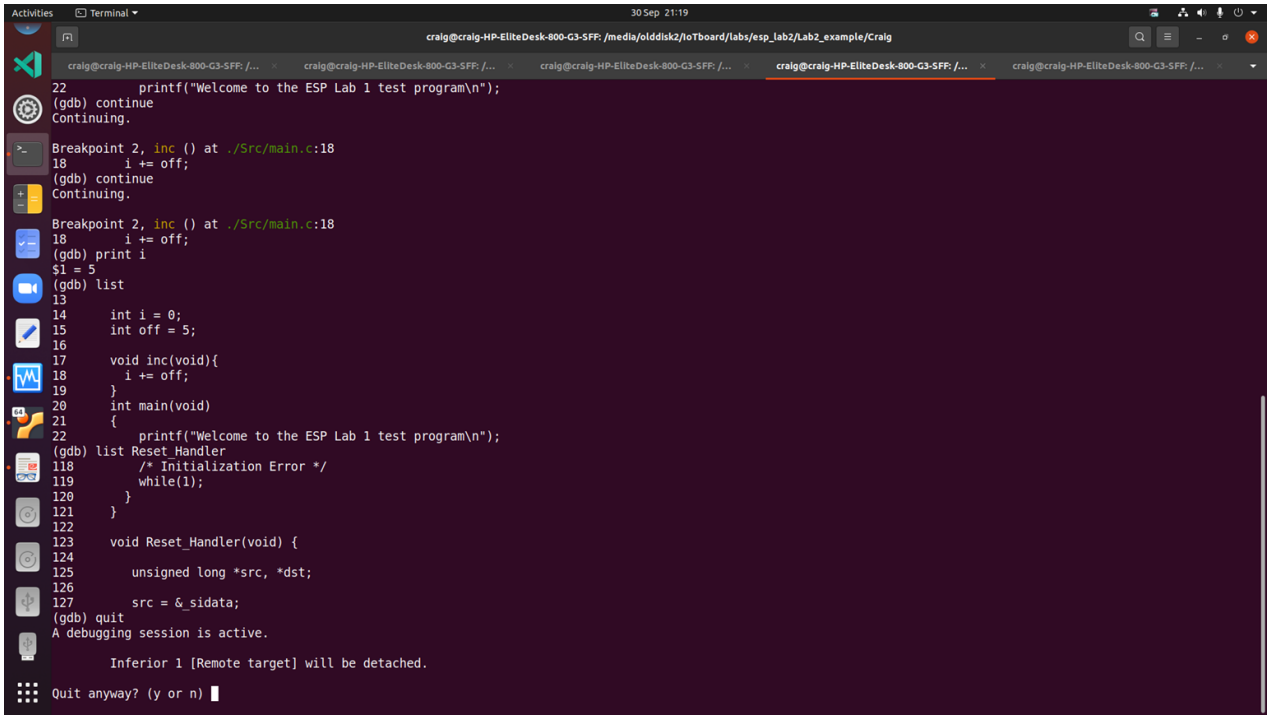
(gdb) *load*
Loading section .isr_vector, size 0x188 lma 0x8000000
Loading section .text, size 0x1f090 lma 0x80001c0
Loading section .RamFunc, size 0x68 lma 0x801f250
Loading section .rodata, size 0x870 lma 0x801f2b8
Loading section .ARM, size 0x8 lma 0x801fb28
Loading section .init_array, size 0x8 lma 0x801fb30
Loading section .fini_array, size 0x8 lma 0x801fb38
Loading section .data, size 0xa00 lma 0x801fb40
conStart address 0x0800039a, load size 132360
Transfer rate: 39 KB/sec, 8824 bytes/write.

From here we can now control the program. However if we just run the code we will not be much better off than we were using telnet. So we could issue come debug commands. One of the simplest is to set some breakpoints – places where we want the code to stop. We can put 2 into the code at main and inc – the 2 functions in main.c



A gdb-multiarch session

Once the code is loaded and gdb-multiarch is running you can list the program, examine variable, you set watchpoints where you get gdb to only report on variables when the reach a certain state or range. There is even a windowed interface to gdb called **ddd** which will allow easier display of data.



More gdb-multiarch

Appendix A The Makefile

```
name of executable - change this to change the executables name
ELF=demo.elf

# object files
OBJS= $(STARTUP)  $(HAL_OBJS) $(DRIVERS)  main.o

HAL_OBJS= stm32l4xx_hal.o stm32l4xx_hal_rcc.o stm32l4xx_hal_rcc_ex.o stm32l4xx_hal_cortex.o
stm32l4xx_hal_pwr.o stm32l4xx_hal_pwr_ex.o stm32l4xx_hal_dfsdm.o stm32l4xx_hal_gpio.o \
stm32l4xx_hal_dma.o stm32l4xx_hal_qspi.o stm32l475e_iot01_qspi.o stm32l4xx_hal_flash.o
stm32l4xx_hal_flash_ex.o stm32l4xx_hal_flash_ramfunc.o stm32l4xx_hal_i2c.o stm32l4xx_hal_i2c_ex.o
stm32l4xx_hal_uart.o stm32l4xx_hal_uart_ex.o
DRIVERS=stm32l475e_iot01.o

# Tool path
TOOLROOT            = /opt/gcc-arm-none-eabi/bin

# Library path
HAL                 = ../../Drivers/STM32L4xx_HAL_Driver
BSP_BL-L475         = ../../Drivers/BSP/B-L475E-IOT01
CMSIS               = ../../Drivers/CMSIS/Core
DEVICE              = ../../Drivers/CMSIS/Device/ST/STM32L4xx/Include
Common_BSP          = ../../Drivers/BSP/Components/Common
APP                 = .

# Tools
CC=$(TOOLROOT)/arm-none-eabi-gcc
LD=$(TOOLROOT)/arm-none-eabi-gcc
AR=$(TOOLROOT)/arm-none-eabi-ar
AS=$(TOOLROOT)/arm-none-eabi-as
OBJCOPY=$(TOOLROOT)/arm-none-eabi-objcopy

# Code paths
APP_CODE=$(APP)/Src

# Search path for peripheral library
VPATH=$(HAL)/Src:$(BSP_BL-L475):$(APP_CODE)
vpath %.c $(APP_CODE)
vpath %.s $(APP_CODE)
vpath %.c $(HAL)/Src
vpath %.c $(BSP_BL-L475)

# Processor and board specific settings
PTYPE = STM32L475xx
LDSCRIPT = STM32L475VGTx_FLASH.ld
STARTUP=  startup_stm32l475xx.o system_stm32l4xx.o stm32l4xx_hal_msp.o stm32l4xx_it.o syscalls.o
BOARD=USE_HAL_DRIVER
BOARDTYPE=USE_STM32L475E_IOT01

# compilation flags for gdb
CFLAGS  = -O0 -g

# Compilation Flags
FULLASSERT = -DUSE_FULL_ASSERT
#not sure about last 2 vfp flags? Also not sure about interwork or PTYPE -mfloat-abi=hard -mfpu=fpv4-sp-d16
LDFLAGS+= -T$(LDSCRIPT) -mthumb -mcpu=cortex-m4
CFLAGS+= -mcpu=cortex-m4 -mthumb -D$(BOARD) -D$(BOARDTYPE) -D$(PTYPE)
CFLAGS+= -I$(APP)/Inc -I$(HAL)/Inc -I$(CMSIS)/Include -I$(Common_BSP)  -I$(DEVICE) -I$(BSP_BL-L475)
CFLAGS+=  -DUSE_USB_OTG_FS  -D$(BOARD) -DUSE_STDPERIPH_DRIVER $(FULLASSERT)
```

```
# Build executable
$(ELF) : $(OBJS)
$(LD) $(LDFLAGS) -o $@ $(OBJS)

# compile and generate dependency info
%.o: %.c
$(CC) -c $(CFLAGS) $< -o $@

%.o: %.s
$(CC) -c $(CFLAGS) $< -o $@

%.hex: %.elf
$(OBJCOPY) -Oihex $*.elf $*.hex

%.flash: %.hex
@echo FLASH $<
@openocd -f board/stm32l4discovery.cfg \
-c "init" \
-c "reset init" \
-c "stm32l4x mass_erase 0" \
-c "flash write_image $(*).hex" \
-c "reset" \
-c "shutdown" $(NULL)
@touch $@

clean:
rm -f $(OBJS) $(OBJS:.o=.d) $(ELF) $(ELF).hx $(ELF).flash

debug: $(ELF)
multiarch-gdb $(ELF)

# pull in dependencies
-include $(OBJS:.o=.d)
```

Appendix B – edited linker description file STM32L475VGTx_FLASH.ld

```
/*
******************************************************************************
**


**  File        : LinkerScript.ld
**
**  Abstract    : Linker script for STM32L475VGTx Device with
**                1024KByte FLASH, 96KByte RAM
**
**                Set heap size, stack size and stack location according
**                to application requirements.
******************************************************************************
*/

/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20018000;                    /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200;                      /* required amount of heap  */
_Min_Stack_Size = 0x400;                     /* required amount of stack */


/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 96K
RAM2 (xrw)      : ORIGIN = 0x10000000, LENGTH = 32K
FLASH (rx)      : ORIGIN = 0x8000000, LENGTH = 1024K
}

/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(8);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(8);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(8);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    . = ALIGN(8);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH

  /* Constant data goes into FLASH */
  .rodata :
  {
    . = ALIGN(8);
```

```
  *(.rodata)                /* .rodata sections (constants, strings, etc.) */
  *(.rodata*)               /* .rodata* sections (constants, strings, etc.) */
  . = ALIGN(8);
} >FLASH

/* Initialized data sections goes into RAM, load LMA copy after code */
.data :
{
  . = ALIGN(8);
  _sdata = .;               /* create a global symbol at data start */
  *(.data)          /* .data sections */
  *(.data*)         /* .data* sections */

  . = ALIGN(8);
  _edata = .;               /* define a global symbol at data end */
} >RAM AT> FLASH

/* Uninitialized data section */
. = ALIGN(4);
.bss :
{
  /* This is used by the startup in order to initialize the .bss secion */
  _sbss = .;        /* define a global symbol at bss start */
  __bss_start__ = _sbss;
  *(.bss)
  *(.bss*)
  *(COMMON)

  . = ALIGN(4);
  _ebss = .;        /* define a global symbol at bss end */
  __bss_end__ = _ebss;
} >RAM

/* User_heap_stack section, used to check that there is enough RAM left */
._user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM
```