The ARM Cortex M0 processor supports only the thumb2 assembly language instruction set. This instruction set consists of fifty 16-bit instructions and six 32-bit instructions. The general syntax for the assembly language in Keil is:

[label mnemonic operand1, operand2, ...; comments

Note that there is no colon separating the label from the mnemonic and you can put the label on a separate line by itself. The label is optional. The first operand is the destination operand if there are multiple operands. Everything after a semicolon is a comment until the next line.

In Line and Embedded Assembly

To use inline assembly code you must use the __asm directive. This allows you to place assembly code inside your c-code as in:

```c
__asm
    {movs r1, r2;
     movs r3, r2;
    }
```

Inline assembler is not available for the ARM Cortex M0 processor but is available on processors that support the full 32-bit instruction set.

Embedded assembly is more useful. Embedded assembly places an assembly function inside a C program and can be used with the ARM Cortex M0 processor.

Example 1 shows how to use an assembly function along with C code as a delay function when toggling bit PA.7. The assembly function creates an empty loop as a software delay. Example 2 uses an assembly function to set or clear a bit on port 0.

Example 1 is occasionally useful for sections of code that require lots of bit manipulation or that may require the fastest possible speed. For example the ARM processor has a barrel shifter that is available in hardware which is not used by the C compiler. It could be accessed by way of embedded assembly code.

Example 1

Note that an s has been added to many assembler mnemonics as in

```assembly
adds r1, r2, #1.
```

Most ARM instructions are available with or without the s where the instruction without the s does not alter the flags while the instruction with the s does. For our particular compiler/assembler, this is not an option and the added s is necessary.

This example uses an embedded assembler function as a delay when writing to a specific bit 7 on port A.
/* AsmLEDPA7.c              October 24, 2015
   This program blinks an LED connected to PA7.
   It uses an assembly module for the delay time.
*/
#include "stm32f407vg.h"
__asm void Delay(int x);
int main()
{int i, tmp;
 //Clock bits
RCC_AHB1ENR |= 1;         //Bit 0 is GPIOA clock enable bit
RCC_AHB1ENR |= 2;         //Bit 1 is GPIOB clock enable bit
//I/O bits  PA7
GPIOA_MODER |= 0x4000;    //Bits 15-14 = 01 for digital output on PA7
//OTYPER register resets to 0 so it is push/pull by default
GPIOA_OSPEEDER |= 0xC000;  //Bits 15-14 = 11 for high speed on PA7
//PUPDR defaults to no pull up no pull down
//Main program loop
tmp = 0;
while(1)
{if(tmp == 0)
   GPIOA_ODR &= ~(1 << 7);            //Only PA7 is set up for output so other bits
else
   tmp = ~tmp;                 // have no effect.
Delay(5000);
//for(i=0;i<200000;i++);    //Put this in to slow down toggle
}

__asm void Delay(int x)
{    //Put label in column 0
    MOVS R1, #0;       R1 = 0
loop    ADDS R1, R1, #1;   incr R1
    CMP R0, R1;        Compare R1 to x in R0
    BGT loop;          Branch greater than to loop
    BX LR;             return
};
Example 2
This example uses an assembler function to write a specified bit to PA
/*ASMWriteBit.c
 * Accepts a bit value and a bit position and writes the value
to the bit position on PA. Program sequentially changes 12
of the bits on PA to 1 followed by all to 0.
The first four parameters are passed in R0 to R3. After that
parameters are passed on the stack. Any return value is
passed back in R0.
*/
#include "stm32f407vg.h"
__asm WriteBit(int bitPos, int value);
int main()
{i = 0, j;
 //Clock bits
 RCC_AHB1ENR |= 1;         //Bit 0 is GPIOA clock enable bit
 GPIOA_MODER |= 0x00555555; //Bits 0-11 on PA set to 01 output in MODER
 //OTYPER register resets to 0 so it is push/pull by default
 GPIOA_OSPEEDER |= 0xC000;  //Bits 15-14 = 11 for high speed on PA7
 while(1)
 {for(i=0;i<12;i++)
  {j = i;
   WriteBit(j, 1);
  }
 for(i=0;i<12;i++)
  {WriteBit(i, 0);
  }
 }
 __asm WriteBit(int bitPos, int value)
 { push {r0-R5};        save the registers
   movs r2, #1;         shift a 1 to the left
   lsls r2, r0;           bitPos times
   ldr r4, =0x40020014; address of Port A
   ldr r3, [r4];        Copy port 0 r3
   cmp r1, #0;          If value is a 0
   BEQ over;              branch to over
   orrs r3, r2;         OR r2 with data from port A
   done  str r3, [r4];        Store result in port 0
   pop {r0-r5};         Restore the registers
   bx lr;
}
In general, for the ARM processors you can add an s to an instruction and it will alter the flags. Using the same instruction without an s will not alter the flags. For the compiler/assembler that we have this is often not an option. Using `add r1, r2` will generate an error which says that the "flag preserving form of the instruction is not available". Change such instructions to `adds r1, r2`

Loading immediate data into a register:
```
ldr R3, =0x000003FF;  Loads the 32 bit hex number into R3
ldr R2, =41;          Loads 41 base 10 into R2
```

Store a register into a memory:
```
str R2, [R1];         Stores R2 into the memory whose address is in R1
```

Copy one register into another:
```
movs R2, R3;           Copies R3 into R2
movs R2, #0xFF;        Copies 255 into R2. Immediate limited to 0 to 255
```

Add:
```
adds R2, R3;           Add R2 + R3 and puts result in R2 (flags set)
adds R10, R11;         R10 = R10 + R11 but flags not set
```

Note: Flags are set only if registers are R0 to R7

```
adcs R2, R3;           R2 = R2 + R3 + carry
```

Note: Can be used only with registers are R0 to R7

Subtract:
```
subs R1, R2;           R1 = R1 - R2
subs R1, R2, R3;       R1 = R2 - R3
```

Note: Can be used only with registers are R0 to R7

```
sbcs R1, R2;           R1 = R1 – R2 and accounts for carry flag (borrow)
```

Note: Can be used only with registers are R0 to R7

Multiply:
```
mpys R1, R2;           R1 = R1*R2
```

Note: Can be used only with registers are R0 to R7

Compare:
```
cmp R2, R10;           Subtracts R10 from R2, discards result and sets flags
cmp R1, #0x20;         Subtracts 0x20 from R1, discards result and set flags
```

Note: for immediate operand value must be 0 to 255.

Logical AND, OR, and Exclusive OR
```
ands R1, R2;            R1 = R1 AND R2
orrs R1, R2;            R1 = R1 OR R2
eors R1, R2;            R1 = R1 Ex Or R2
```

Note: Can be used only with registers are R0 to R7
Negation:
\[
\text{neg } R2, R2; \quad R2 = -R2 \\
\text{neg } R2, R3; \quad R2 = -R3
\]

Left and Right Shift
\[
\text{lsls } R2, R3, #5; \quad \text{shift left } R3 \text{ 5 times and store in } R2 \\
\text{lsls } R2, R3; \quad \text{shift left } R2, R3 \text{ value times. If } R3 \text{ is 31, } R2 \text{ is cleared, if } R3 > 31 \text{ and carry are cleared.} \\
\text{lsrs } R2, R3, #5; \quad \text{shift right } R3 \text{ 5 times and store in } R2 \\
\text{lsrs } R2, R3; \quad \text{shift right } R2, R3 \text{ value times. If } R3 \text{ is 31, } R2 \text{ is cleared, if } R3 > 31 \text{ and carry are cleared.} \\
\text{asrs } R2, R3, #5; \quad \text{arithmetic shift right } R3 \text{ 5 times and store in } R2 \\
\text{asrs } R2, R3; \quad \text{arithmetic shift right } R2, R3 \text{ value times. If } R3 \text{ is 31, } R2 \text{ is cleared, if } R3 > 31 \text{ and carry are cleared.}
\]

Branch instructions
\[
\text{B } \text{Target; } \quad \text{unconditional branch to target within } \pm 2K \text{ (10 bits)} \\
\text{B(Cond) } \text{Target; } \quad \text{conditional branch to target within } -252 \text{ to } +258 \\
\text{BL(Cond) } \text{Target; } \quad \text{Branch and link. Stores return address in } R14 \text{ and conditionally branches to target. To return from subroutine move } R14(\text{link reg}) \text{ into } R15(\text{PC})
\]

<table>
<thead>
<tr>
<th>Condition</th>
<th>Flags</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>unsigned higher or same</td>
</tr>
<tr>
<td>CC/LO</td>
<td>C clear</td>
<td>unsigned lower</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>no overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>unsigned higher</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>unsigned lower or same</td>
</tr>
<tr>
<td>GE</td>
<td>N equals V</td>
<td>signed greater or equal</td>
</tr>
<tr>
<td>LT</td>
<td>N not equal to V</td>
<td>signed less than</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear AND (N equals V)</td>
<td>signed greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Z set OR (N not equal to V)</td>
<td>signed less than or equal</td>
</tr>
<tr>
<td>AL</td>
<td>(ignored)</td>
<td>always (usually omitted)</td>
</tr>
</tbody>
</table>

Push and Pop
\[
\text{push } \{R1, R2\}; \quad \text{Pushes } R1 \text{ and } R2 \text{ on the stack} \\
\text{push } \{R1-R5\}; \quad \text{Pushes } R1, R2, R3, R4, \text{ and } R5 \text{ on the stack} \\
\text{push } \{R1, R2, LR\}; \quad \text{Pushes } R1, R2, \text{ and the link register on the stack} \\
\text{pop } \{R1, R2\}; \quad \text{Pops } R1 \text{ and } R2 \text{ on the stack} \\
\text{pop } \{R1-R5\}; \quad \text{Pops } R1, R2, R3, R4, \text{ and } R5 \text{ from the stack}
\]
pop (R1, R2, LR);  Pops R1, R2, and the link register from the stack
Note: Can be used only with registers are R0 to R7 and the link register.
Floating Point (Math) Library
To do programs in floating point arithmetic you need only declare variables of type float or double in your C code. If you declare a float variable it will be 32-bits long and have a range of about $0.34 \times 10^{38}$ to $+3.4 \times 10^{38}$. The float and int variable types both are 32-bits long on the Cortex M4 processor but floating point numbers require more processing code to manage the exponent. Floating point arithmetic will noticeably slow down processing and it will require more space for the libraries. The ARM Cortex M4 processor also supports the double (64-bit) data type which has a range of about $-1.7 \times 10^{308}$ to $+1.7 \times 10^{308}$. All of the standard math functions for floating point arithmetic are available in the math.h library.

PWM on the ARM M4 processor
The STM32F407VG has 14 timers and 12 of them can be used to generate Pulse Width Modulation (PWM) signals. (Timers 6 and 7 cannot be used for PWM)

In PWM we start with a base frequency of say 1,000Hz and each cycle we output a single pulse whose width varies from 0% of the period to 100% of the period.

![Figure 1](image.png)

**Figure 1**
Pulse width modulation. In this case the signal is 50% of maximum.

The base frequency of the PWM is $f = 1/T$ so that the frequency content of a PWM signal will have frequencies at $f$ and higher due to the base frequency. The signal information is transmitted by the duty cycle and will be, generally, a much lower frequency signal. Thus a relatively simple low pass filter can eliminate the frequencies at the base frequency and above and recover an analog signal from the PWM.

The ARM processor generates PWM signals using three registers: The timer count register TIMx_CNT, the auto-reload register TIMx_ARR, and the capture/compare register TIMx_CCRx. The CNT register counts up from zero until it reaches the value in the ARR register. This amount of time establishes the base frequency. The output goes from low to high at the start of each cycle. When the CNT register gets to the value in the CCR register the signal set back to low.

Figure 2 shows the functional diagram for counters 2 to 5. From the figure we note that there are four outputs associated with the timer. Each of these outputs has a capture register but they all share a common counter register and a common auto-reload register. This means the timer can generate four PWM signals as long as they all have the same base frequency. Further, Timers 2 to 5 can run simultaneously and independently of one another so these four timers could generate 16 PWM signals at 4 different base frequencies.

*Example 1*
Use Timer 3 to use PC6 (Channel 1) to generate a PWM signal which has a 12-bit resolution and produces a ramp function that runs from 0 to 4095. (Note that the pin assignments for each timer channel are listed in the data sheet¹.) This example uses an integer variable called `tmp` which goes from 0 to 4095. After `tmp` is changed to a new value it is loaded into TIM3_CCR1 which holds the pulse width count.

![Figure 134. General-purpose timer block diagram](image)

The auto-reload register is loaded with 4096 which is the maximum count for one cycle. The value of `tmp` is set back to zero when it reaches this maximum. A software delay has been added to slow the time it takes the ramp to grow.

Example 2
Create a PWM signal using Timer 2 to output a sinusoid at 1 KHz which has a sample frequency of 20000 Hz. The PWM output will appear on PA5 which is channel 1 of Timer 2 is chosen. See the data sheet. Since the sample frequency is 20000 Hz and the signal frequency is 1000 Hz there will be 20 samples per signal cycle. The while loop increments `t` by the sample time, $T$, twenty times before starting it again at 0.

A sinusoid varies from -1 to +1. We add one to the sine wave so that it goes from 0 to 2. We then multiply by 2000 so that it ranges from 0 to 4000. Since the auto-reload register has the number 4200 in it we are very near the maximum. Since 4000/4200 is about 95% this will be the maximum duty cycle of the PWM output.

Example 1

//PWMEx1.c
/* This program generates a PWM signal on PC6 using Timer 3. The
   signal has 12-bits of resolution and outputs a ramp function
   which goes from 0 to 4095.
*/
#include "stm32f407vg.h"
int main()
{
    int tmp, i;
    //Clock bits
    RCC_AHB1ENR |= 4;         //Bit 2 is GPIOC clock enable bit
    RCC_APB1ENR |= 2;         //Enable peripheral timer for timer 3 (bit 1)
    //I/O bits
    GPIOC_MODER |= 0x2000;    //Bits 13-12 = 10 for Alt Funct Mode on PC6
    //OTYPER register resets to 0 so it is push/pull by default
    GPIOC_OSPEEDER |= 0x3000;  //Bits 13-12 = 11 for high speed on PC6
    //PUPDR defaults to no pull up no pull down
    //Timer 3 bits
    GPIOC_AFRL = 0x02000000; //Sets PC6 to Timer 3
    TIM3_CCMR1 |= 0x60;      //Timer 3 in PWM Mode bits 6,5,4 = 110
    TIM3_CCMR1 |= 0x0C;      //Timer 3 Preload enable and fast enable
    TIM3_CR1 |= (1 << 7);    //Auto reload is buffered
    TIM3_PSC = 0;            //Don't use prescaling
    TIM3_ARR = 4096;         //((168 MHz/2)/4096 = 20508 Hz
    TIM3_CCR1 = 0;           //Duty cycle starts at 0
    TIM3_CCRER |= 1;         //Compare and capture output enable
    TIM3_EGR |= 1;           //Enable event
    TIM3_CR1 |= 1;           //Enable Timer 3
    //Main program loop
    tmp = 0;
    while(1)
    {
        TIM3_CCR1 = tmp;
        for(i=0;i<10000;i++); //Delay
        tmp++;
        if(tmp >= TIM3_ARR)
            tmp = 0;
    }
}
*************** Example 2 ***************

//PWMEx2.c
/* Create a PWM signal using Timer 2 to output a sinusoid at 1 KHz
which has a sample frequency of 20000 Hz. The PWM output appears
on PA5. */

#include "stm32f407vg.h"
#include <math.h>        //Need this for sine function
const float w = 2*3.14159265359*1000;
const float T = 1.0/20000;
int main()
{
  float t;
  //Clock bits
  RCC_AHB1ENR |= 1;         //Bit 0 is GPIOA clock enable bit
  RCC_APB1ENR |= 1;         //Enable peripheral timer for timer 2 (bit 0)
  //I/O bits
  GPIOA_MODER |= 0x800;    //Bits 11-10 = 10 for Alt Funct Mode on PA5
  GPIOA_OSPEEDER |= 0xC00;  //Bits 11-10 = 11 for high speed on PA5
  //OTYPER register resets to 0 so it is push/pull by default
  //PUPDR defaults to no pull up no pull down
  //Timer 2 bits
  GPIOA_AFRL = 0x01000000;  //Sets PA5 to Timer 2
  TIM2_CCMR1 |= 0x01000000;  //Timer 2 in PWM Mode bits 6,5,4 = 110
  TIM2_CCMR1 |= 0x00000000;  //Timer 2 Preload enable and fast enable
  TIM2_CR1 |= (1 << 7);    //Auto reload is buffered
  TIM2_PSC = 0;            //Don't use prescaling
  TIM2_ARR = 4200;         //((168 MHz/2)/4200 = 20000 Hz
  TIM2_CCR1 = 0;           //Duty cycle starts at 0
  TIM2_CCER |= 1;          //Compare and capture output enable
  TIM2_EGR |= 1;           //Clear counter on update
  TIM2_CR1 |= 1;           //Enable Timer 2
  //Main program loop
  t = 0;
  while(1)
  {
    t = 0;
    while(t < 20*T)       //20 samples make one cycle at 1 KHz
    {
//maximum number of counts is 4200
      TIM2_CCR1 = (int)(2000*(1 + sin(w*t)));
      t += T;
    }
  }
}