Introduction

This application note is a short tutorial on the following topics.

1) "Bare Metal" firmware development.
2) "CMSIS" and "HAL" oriented ARM microcontroller firmware techniques:
   Our example is for ST Microelectronics STM32 series parts.
3) STM-32 firmware techniques, for PWM firmware:
   Exploring a C code PWM macro provided in the ST Microelectronics HAL.

Background and terminology:

For firmware engineers, the term "bare metal" means "bare silicon chip". In the past firmware engineers typically wrote their own firmware programs completely from scratch, without using third party source code provided by the chip manufactures (like ST and ARM) to access the control registers on the chip. This style of programming is known as "bare metal" programming. But presently firmware is often designed, to utilize code provided by the chip manufacturers which acts as a layer (jacket) between the most detailed "bare metal" access and control over what the chip does and how it does it. The reason for this jacket layer - "Hardware abstraction layer" or "HAL" is to make the task of the firmware engineer easier, and to provide consistent code for accessing the functionality of the chip.

For engineers who prefer "bare metal" programming, it is important to distinguish "bare metal" from "made from scratch". This is because it is possible to achieve the effect of "bare metal" coding even if it is not "made from scratch", all it takes is readable "HAL" code, even if it is created by a third party.

Advantages of "bare metal" control and understanding of the source code.

The engineer understands what the program is doing through all layers of software.

The engineer is required to fully understand the capabilities of the chip.

The local development team will be enabled to debug and test the code through all layers down to the "bare metal" i.e. down to the silicon chip control registers.

The engineer has complete control of functionality down to the silicon chip.

Advantages of "programming from scratch", or "single project programming"

The coding style can be more consistent since only one programmer or one team controls the standard. (But, they need to agree to the coding style.)

The team only needs to write code for exactly what they intend to do, and no more.
This can drastically reduce "code bloat" - code complexity

Examples of the “layered” auto generated code approach are HAL: “The Hardware abstraction layer” (a common term used by the industry) and CMSIS: “The Cortex Microcontroller Software Interface Standard” developed by ARM holdings.

Advantages of “layered /jacketed” approach.

The inner layer(s) are written once by experts at the silicon computer chip company.

End product developers who use a certain chip will all use the same software to access the control registers, and, in the case of the higher level HAL code, implement practical functionality.

HAL code provides a large amount of functionality to all product developers. This allows them to develop practical working firmware rapidly.

A final note on “bare metal” , versus third party / auto generated “HAL” code

You can still achieve “bare metal” control and understanding with “auto generated code” HAL code.

In order to do this, readable “HAL” source code must be provided (not binary libraries), and adequate local attention must be given to this third party provided code. The code needs to be reviewed, tested, and possibly optimized, locally by the end product firmware developers. Clearly, “bare metal” engineers are still need by the local engineering firms in order to create the most innovative, highest quality product.

A simple example - RGB LED Color Control Project

The “use case” (practical example), for this discussion is changing the PWM duty cycle to control the color brightness and hue in an RGB Led attached to a “STM32 value line Discovery” evaluation board using the ST Microelectronics “STM32F100RBT6B” microcontroller.

Ref: http://www.st.com/en/evaluation-tools/stm32vldiscovery.html

The author of this project largely used “auto generated code” techniques to create the firmware code to get the RGB Led to display its 3 colors each at user selectable intensity levels. This was done using the “STM32CubeMX” Code generation tool. But he was striving for “bare metal” control and understanding.

Did the code auto generation tool generate code that is “plug and play” to get the microcontroller to perform the desired task ? Well not exactly, but it got the firmware engineer started, in the right direction. For many engineers, this is not unlike being told “you were started in the right direction” if someone tosses you in a pool and asks you to swim a lap. Well, you were certainly started in the right direction! You were provided a pool, a swimsuit, (hopefully) and some initial momentum , now go swim !

About this time a non bare metal engineer may wish they had used an Arduino or a Raspberry Pi with Python, to generate PWM signals but with a little “bare metal” assistance our firmware will get there !

Once the firmware engineer looks at the code generated, they will notice that the CMSIS (the closest layer to the bare silicon chip) has been made part of the set of files that are used to build the desired firmware. Also the HAL code which is one layer up from the CMSIS and uses the CMSIS, has been included in the set of files making up the firmware. (Gone are the days when 4 files made up your firmware.) Depending on how you told the code generation tool "STM32CubeMX" - to setup peripherals, and auto generate code, the number of HAL files, will vary. Also, initialization code that sets up various areas of the chip (a.k.a. peripherals) to be configured to do what you want will be automatically injected into your "main.c" file. *You can find these auto-generated functions in "main.c" since all of these functions start with the prefix "MX_" indicating they were auto generated by "STM32CubeMX".*

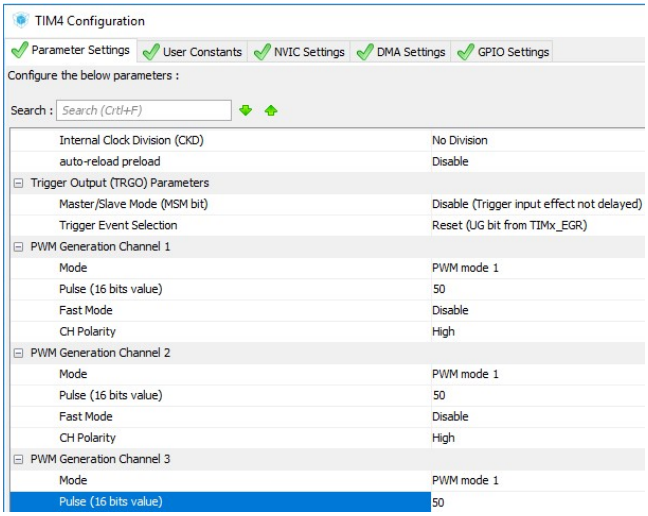
After all this instant code generation, the firmware engineer will realize that to achieve "bare metal" understanding and control, they must understand the HAL functions used and, the CMSIS code files that were added to his project file structure by STM32CubeMX. Time to swim !

The general process of acquiring "bare metal" understanding involves using .pdf and DOxygen HAL documentation, web tutorials, and auto generated source code review, but now let's discuss a specific example of this process: creating PWM functionality for setting color and intensity of an RGB LED.

For this little project we need 3 PWM signals. It turns out that the "STM32F100RBT6B" microcontroller has an area on the chip (peripheral) that is devoted to sending or receiving timing signals on certain pins of the part. This peripheral is called a TIMER and there are 4 of them, not only that but each timer may have as many as 4 channels. I used the ST Microelectronics code generation tool "STM32CubeMX" to configure the TIMER4 to output a PWM timing signal to pins "PB6", "PB7" and "PB8" for controlling intensity on the Red, Green and Blue wires going to the LED.

The design goal for this project was to create a PWM repetition rate of 200 Hz for all 3 channels (1,2, 3) of TIMER4; this is feasible and can be set up using the STM32CubeMX, but it requires "bare metal" hardware information… in order to do it.) By the way, 200 Hz is a good choice so that the customer will see no flicker in the LED , but only changes in intensity.

It also turns out that it is possible to auto generate code to initialize each of the 3 TIMER#4 channels to start at 50% on time by configuring STM32CubeMX.



But when the engineer wants finally to implement user control to change the PWM on time percentage on the fly while the program is running, this is not something the STM32CubeMX can do for you.

File searches in my auto generated project, looking at web tutorials, data sheets etc, yielded the tip: “Use this HAL macro”, located at:

/<yourEclipseProjectName>/Drivers/STM32F1xx_HAL_Driver/Inc/stm32f1xx_hal_tim.h

```
1386 /**
1387   * @brief  Sets the TIM Capture Compare Register value on runtime without
1388   *         calling another time ConfigChannel function.
1389   * @param  __HANDLE__: TIM handle.
1390   * @param  __CHANNEL__ : TIM Channels to be configured.
1391   *          This parameter can be one of the following values:
1392   *            @arg TIM_CHANNEL_1: TIM Channel 1 selected
1393   *            @arg TIM_CHANNEL_2: TIM Channel 2 selected
1394   *            @arg TIM_CHANNEL_3: TIM Channel 3 selected
1395   *            @arg TIM_CHANNEL_4: TIM Channel 4 selected
1396   * @param  __COMPARE__: specifies the Capture Compare register new value.
1397   * @retval None
1398   */
1399 #define __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__) \
1400 (*(__IO uint32_t *)(&((__HANDLE__)->Instance->CCR1) + ((__CHANNEL__) >> 2U)) = (__COMPARE__))
1401
```

But where in the Doxygen comments above the macro does it say “Application note - Use this macro to change PMW percentage on time on the fly.” ? – Maybe somebody will write a fully annotated manual to supplement this information one day, or maybe such a manual already exists – if this is so let me know !

This macro accesses the “bare metal” registers of the TIMER peripheral that affect the duty cycle of the PWM. It is coded in such a way to work for all channels of all 4 timers.

So let’s analyze this Macro and C code:

“How and Why a HAL Macro ?”

Here is my utilization of this macro to adjust the red channel pwm of my RGB LED:

```
// Which as utilized in my wrapper code file of "ledIntensity.c" as follows:

extern TIM_HandleTypeDef htim4; // inside timerPeripheral.h

void setRedPercent(int32_t percent)
{
    percent = percent <= 100  ? percent : 100U; // Limit to <=100 ########
    percent = percent > 0     ? percent : 0;    // Limit to >=0

    // htim4 is in timerPeripheral module.
    // ##### Using the "super convenient" (but obscuring) HAL Macro:
    __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, percent);
    redLEDIntensity = percent;
}
```

Some caveats: 1) The "percent" parameter for the macro is not restricted by the macro to be 0 to 100, but for this example the timers were previously initialized so that 0 to 100 gives us a duty cycle of 0 to 100% . 2) TIM_CHANNEL_1,2,3,4 are not identical with the numbers 1,2,3,4. The macro's Doxyegen comments do not cover these details.

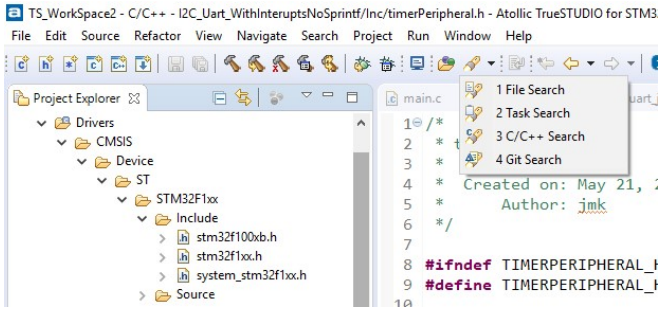
The "htim4" is a very import data structure that is used to communicate with the HAL module for many Timer setup and control functions and Macros in the TIMER HAL module code (stm32f1xx_hal_tim.c, and .h) it is of type **TIM_HandleTypeDef** which is defined as follows:

```
// Located at: /<yourEclipseProjectName>/Drivers/STM32F1xx_HAL_Driver/Inc/stm32f1xx_hal_tim.h

// Line: 289:

typedef struct
{
  TIM_TypeDef                 *Instance;     /*!< Register base address             */
  TIM_Base_InitTypeDef        Init;         /*!< TIM Time Base required parameters */
  HAL_TIM_ActiveChannel       Channel;      /*!< Active channel                    */
  DMA_HandleTypeDef           *hdma[7U];    /*!< DMA Handlers array
                                               This array is accessed by a @ref TIM_DMA_Handle_index */
  HAL_LockTypeDef             Lock;         /*!< Locking object                    */
  __IO HAL_TIM_StateTypeDef   State;        /*!< TIM operation state               */
}TIM_HandleTypeDef;
```

For our macro we only use the structure member "*Instance". But we will soon see that this "Instance" is very important. It is the starting addresses of the "bare metal" control registers for the TIMER peripherals. How do we know this? Notice that "Instance" is of type TIM_TypeDef so a file search in the Atollic / Eclipse IDE lands us here:



We have now entered "the CMSIS zone" - the HAL as the higher layer, uses the lower layer CMSIS.

And here we find that TIM_TypeDef is defined as follows:

```
// Located at /<yourEclipseProjectName>/Drivers/CMSIS/Device/ST/STM32F1xx/Include/stm32f100xb.h

typedef struct
{
  __IO uint32_t CR1;             /*!< TIM control register 1,                    Address offset: 0x00 */
  __IO uint32_t CR2;             /*!< TIM control register 2,                    Address offset: 0x04 */
  __IO uint32_t SMCR;            /*!< TIM slave Mode Control register,           Address offset: 0x08 */
  __IO uint32_t DIER;            /*!< TIM DMA/interrupt enable register,         Address offset: 0x0C */
  __IO uint32_t SR;              /*!< TIM status register,                       Address offset: 0x10 */
  __IO uint32_t EGR;             /*!< TIM event generation register,             Address offset: 0x14 */
  __IO uint32_t CCMR1;           /*!< TIM  capture/compare mode register 1,      Address offset: 0x18 */
  __IO uint32_t CCMR2;           /*!< TIM  capture/compare mode register 2,      Address offset: 0x1C */
  __IO uint32_t CCER;            /*!< TIM capture/compare enable register,       Address offset: 0x20 */
  __IO uint32_t CNT;             /*!< TIM counter register,                      Address offset: 0x24 */
  __IO uint32_t PSC;             /*!< TIM prescaler register,                    Address offset: 0x28 */
  __IO uint32_t ARR;             /*!< TIM auto-reload register,                  Address offset: 0x2C */
  __IO uint32_t RCR;             /*!< TIM  repetition counter register,          Address offset: 0x30 */
  __IO uint32_t CCR1;            /*!< TIM capture/compare register 1,            Address offset: 0x34 */
  __IO uint32_t CCR2;            /*!< TIM capture/compare register 2,            Address offset: 0x38 */
  __IO uint32_t CCR3;            /*!< TIM capture/compare register 3,            Address offset: 0x3C */
  __IO uint32_t CCR4;            /*!< TIM capture/compare register 4,            Address offset: 0x40 */
  __IO uint32_t BDTR;            /*!< TIM break and dead-time register,          Address offset: 0x44 */
  __IO uint32_t DCR;             /*!< TIM DMA control register,                  Address offset: 0x48 */
  __IO uint32_t DMAR;            /*!< TIM DMA address for full transfer register, Address offset: 0x4C */
  __IO uint32_t OR;              /*!< TIM option register,                       Address offset: 0x50 */
}TIM_TypeDef;
```

At this point to show that we are finally at the bare metal level, our desire target.
Ref The Peripherals Manual "**STM32F100xx.PeriferalsRefManual.en.CD00246267.pdf**", a.k.a. “RM0041”.

<u>Important points of interest in this manual (snippets) for us right now are:</u>

Section 2.3 – Memory Map

For Timer4:

0x4000 0800 - 0x4000 0BFF	TIM4 timer	*Section 13.4.19 on page 338*

Section 13.4.19 – TIMx register map

Table 73. TIMx register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	TIMx_CR1	Reserved																						CKD [1:0]		ARPE	CMS [1:0]		DIR	OPM	URS	UDIS	CEN
	Reset value																							0	0	0	0	0	0	0	0	0	0
0x04	TIMx_CR2	Reserved																								TI1S	MMS [2:0]			CCDS	Reserved		
	Reset value																									0	0	0	0	0			

. . .

0x30	Reserved															
0x34	TIMx_CCR1	Reserved														CCR1[15:0]
	Reset value															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0x38	TIMx_CCR2	Reserved														CCR2[15:0]
	Reset value															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0x3C	TIMx_CCR3	Reserved														CCR3[15:0]
	Reset value															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0x40	TIMx_CCR4	Reserved														CCR4[15:0]
	Reset value															0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

One wrinkle, is that this table says offset 0x30 is reserved, but the our CMSIS structure "TIM_TypeDef" has the address at offset 0x30 defined as "RCR" – our bet is that the code is more update than the documentation.

Section 13.4.13 to 13.4.16 TIMx CCR1..4 registers: (Short explanation for PWM usage)

"The active capture/compare register contains the value to be compared to the counter TIMx_CNT and signalled on OCx output"

Explanation for PWM use - *This means that the pwm signal level on the OCx output pin is configured to change once the counter TIMx_CNT reaches the value written to the CCRx register. Note there is no restriction on this CCRx value being only 0 to 100, but if the TIMx_CNT register was previously configured to have a maximum of 100 in the initialization code then setting CCRx to be 0 to 100 will be directly equivalent to setting the duty cycle as 0..100.*
(If your wanted duty cycle resolution down to 0.1% you could set Max TIMx_CNT, and Max CCRx to both be 1000.)

Ok, now let's see where in the code does our TIMER4 "do everything" structure " **htim4.Instance**" get setup:
This is at function "MX_TIM4_Init(void)" auto generated by STM32CubeMX, line 522 as illustrated below.

main.c | serialCmdPa... | timerPeriph... | timerPeriph... | ledIntensit

```
513
514 /** TIM4 init function */
515 static void MX_TIM4_Init(void)
516 {
517
518   TIM_ClockConfigTypeDef sClockSourceConfig;
519   TIM_MasterConfigTypeDef sMasterConfig;
520   TIM_OC_InitTypeDef sConfigOC;
521
522   htim4.Instance = TIM4;
523   // 8 Mhz internal RC clock / 400 = 20,000 Hz.
524   htim4.Init.Prescaler = 399; // To get /400, use 399 f
525   htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
526   // Further division of 20,000 /100 is 200 Hz = pwm re
527   htim4.Init.Period = 99; // to get /100 use 99 since c
```

It turns out the TIM4 is a macro that expands to “0x40000000 + 0x00000800” – **bingo !**
This points the bare metal registers as specified in the Peripherals Manual - TIM4 section of section 2.3.

Sure there is more HAL code we could dissect and explore, but now that we know that our “htim4.Instance”: refers to the correct start of the TIMER4 register set, so we can complete the promised dissection of the __HAL_TIM_SET_COMPARE (..) macro.

Repeating the usage illustration here for convenience:

```
// Which as utilized in my wrapper code file of "ledIntensity.c" as follows:

extern TIM_HandleTypeDef htim4; // inside timerPeripheral.h

void setRedPercent(int32_t percent)
{
    percent = percent <= 100  ? percent : 100U; // Limit to <=100 ########
    percent = percent > 0     ? percent : 0;    // Limit to >=0

    // htim4 is in timerPeripheral module.
    // ##### Using the "super convenient" (but obscuring) HAL Macro:
    __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, percent);
    redLEDIntensity = percent;
}
```

The macro expands like this

```
#define __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__) \
(*(__IO uint32_t *)(&((__HANDLE__)->Instance->CCR1) + ((__CHANNEL__) >> 2U)) = (__COMPARE__))
```

__HANDLE__ is a pointer to a our “htim4” structure, so in order of C language precedence:

The first part to be expanded is:
Htim4->Instance->CCR1

The next part is:
&(Htim4->Instance->CCR1), This is the address of CCR1 for TIMER4.

The next part is
(&((__HANDLE__)->Instance->CCR1) + ((__CHANNEL__) >> 2U))

This is adding an integer to a pointer. Note the change in the address of the pointer achieved by doing this is affected by the type of the pointer, in our case the type for CCR1 is “uint32_t”, since our processor has byte addressing and integers are 4 bytes, we note that adding 1 will increment the address by 4. This fact is very important in understanding the result.

Note that __CHANNEL__ is filled with one of the follow constants defined at:

/I2C_Uart_WithInteruptsNoSprintf/Drivers/STM32F1xx_HAL_Driver/Inc/stm32f1xx_hal_tim.h

```
451 #define TIM_CHANNEL_1                      0x00000000U
452 #define TIM_CHANNEL_2                      0x00000004U
453 #define TIM_CHANNEL_3                      0x00000008U
454 #define TIM_CHANNEL_4                      0x0000000CU
455 #define TIM_CHANNEL_ALL                    0x00000018U
```

If they had defined the channels 1,2,3,4 as 0,1,2,3 rather than 0,4,8,C the macro could eliminate the ">>2U" but either way it works !

&((__HANDLE__)->Instance->CCR1) + ((__CHANNEL__) >> 2U)) is the address of CCR1, CCR1, 3 and 4 depending on what x is set to on TIM_CHANNEL_x.

Let's call the result of this "newAddres".

The last two part, is:
*(__IO uint32_t *) newAddress = (__COMPARE__)

This first typecasts newAddress to be pointer to "__IO uint32_" *I have a hunch this is not needed since ,The original address was &((__HANDLE__)->Instance->CCR1) and CCR1 was already defined as type "__IO uint32_t"*, finally the pointer is dereference with the outermost * and then assigned the value of __COMPARE__

This macro is technically correct, super convenient, I suppose, but, it is not easily readable. But, yes, via careful analysis we concluded that it is correct !

If the bare metal guy was to do it his way, he might trade "the ultimate in code portability", for a drastic improvement in readability, by eliminating the macro.

Now we see directly see what the code is doing:

```
24 void setRedPercent(int32_t percent)
25 {
26     percent = percent <= 100 ? percent : 100U;  // Limit to <=100
27     percent = percent > 0     ? percent : 0;    // Limit to >=0
28
29     // htim4 is in timerPeripheral module.
30
31     // Using the official  (but hard to follow) HAL Macro ?????
32     /* Forget about the Macro !
33     __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, percent);
34     */
35
36     // Much more readable and direct:
37     // Set timer4 ch1 "Capture Compare Register":
38     htim4.Instance->CCR1 = percent;
39
40     redLEDIntensity = percent;
41 }
```

Conclusion:

“Bare metal” firmware engineer’s can benefit from available code generation tools such as ST Microelectronic’s “STM32CubeMX”, but they also may desire to occasionally enhance and modify code generated by such tools. Continued improvement in the documentation of the HAL code will also benefit all developers. One area of future discussion is how to use code auto-generation tools to upgrade and enhance existing firmware, but the area interest will be how to ensure sure that enhanced and modified coded written earlier by the end user is not overwritten by the code generation tool when new feature are added.