Digital filter implementation with the FMAC using STM32CubeG4 MCU Package

Introduction

This document applies to STM32CubeG4 MCU Package, for use with STM32G4 Series microcontrollers.

Digital filters are common in a multitude of application domains, from audio to industrial control and IoT. They are used for a variety of tasks, such as removing unwanted interference from a communications signal, controlling a digital power supply or compressing a sound track. Traditionally implemented with digital signal processors (DSP), digital filters are increasingly present in low-cost applications built around general purpose microcontrollers. Indeed, modern microcontroller cores such as the Arm®️ Cortex®️-M4 include DSP features that speed up the mathematical calculations commonly used for signal processing. Nevertheless, filter tasks can consume a large part of the limited processor budget, to the point that there is not enough processor time available for other essential tasks.

There are three ways to free up processor resources:
- Use a faster, more powerful processor
- Add a DSP to perform the filtering
- Use dedicated hardware to offload the processor.

Of these, the third solution is the most cost- and power-efficient. However in a general purpose microcontroller, such hardware also needs to be general-purpose, to support the vast range of applications for which the MCU may be used.

The FMAC (filter math accelerator) combines the flexibility of a DSP with the cost- and power-efficiency of dedicated hardware. The FMAC is built around a 2x16-bit multiplier and a 26-bit accumulator. It offers the choice between finite impulse response (FIR) and infinite impulse response (IIR) filters. It includes 256 words of 16-bit memory, which can be allocated freely to coefficient and state storage, as well as input and output sample buffers. Capable of operating at the same clock frequency as the processor itself, the FMAC can execute filtering tasks as fast as or even faster than the software. When combined with a DMA unit to load the input and empty the output buffer, the FMAC can free up 100% of processor time that would otherwise be taken up with filtering tasks.

This application note explains how to exploit the FMAC to free up the processor, by means of two examples:
- Noise cancellation using an adaptive FIR filter
- Digital power supply control using a 3p3z IIR compensator.

The examples can be found in the STM32CubeG4 MCU Package, which can be downloaded from www.st.com.

AN5305 - Rev 1 - May 2019
For further information contact your local STMicroelectronics sales office.

www.st.com

1 Example 1: FIR adaptive filter

1.1 Overview

STM32CubeG4 MCU Package runs on STM32G4 Series microcontrollers, based on Arm® Cortex®-M processors.

Note: *Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

arm

An adaptive filter is one that changes its impulse reponse according to the varying characteristics of the input signal. Such filters are often used to eliminate unwanted interference picked up in a communication channel. An adaptive algorithm may, for example, detect the frequency components of an interferer, and adjust the coefficients of a filter to create notches at the appropriate points in the filter frequency response. Then when the received signal is passed though the filter, the unwanted components are removed.

The details of the adaptive algorithm are not discussed here - several such algorithms exist. In this example an auto-regressive algorithm is used. The output of the algorithm is a set of coefficients which make up a FIR filter. The coefficients are updated more or less often, depending on the nature of the signal, the channel and the interference. Calculating the coefficients is therefore best done in software, whenever the processor is not performing higher priority tasks. On the other hand, the filter is applied continuously to the input signal, and is subject to the real time constraints of the signal processing chain. It is therefore an ideal candidate for offloading to the FMAC.

In this example, the input signal is assumed to be sampled, digitised and stored in the microcontroller memory in blocks or frames of 2048 samples. Each time a new frame is ready, the DMA is triggered to transfer it to the FMAC, sample by sample, where it is filtered. The filtered data is then transferred back to memory by the DMA. Between each frame there is an opportunity to update the filter coefficients.

1.1.1 Adaptive filtering example (frame 1)

The spectrum of the input signal in Figure 1 is composed of Gaussian or 'white' noise, with two narrowband tones added. The tones are clearly visible at 8 kHz and 16 kHz. The sample frequency is 48 kHz.

Figure 2. Frequency response of adaptive noise filter (frame 1) shows the frequency response of the FIR filter. The two 'notches' are aligned in frequency and amplitude with the tones. Figure 3. 'Whitened' noise (frame 1) shows the result of filtering the input signal - the tones are attenuated to approximately the same level as the noise.

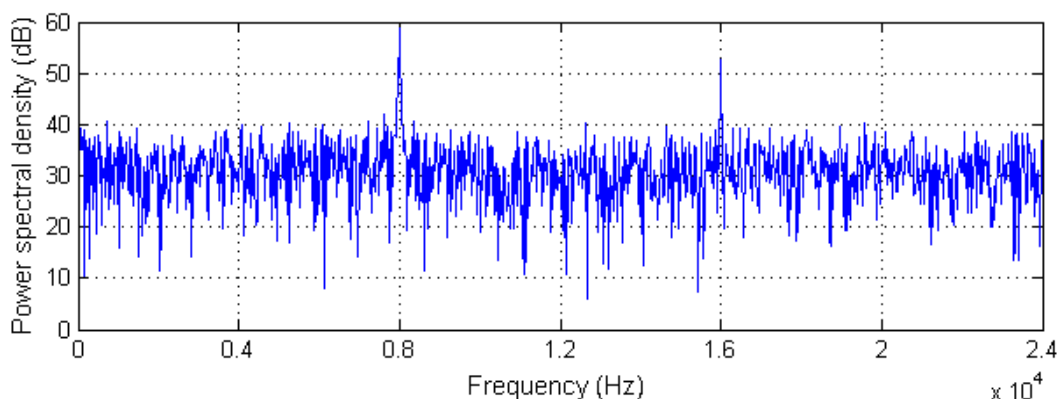**Figure 1. Gaussian noise with two-tone interferer**

Figure 2. Frequency response of adaptive noise filter (frame 1)
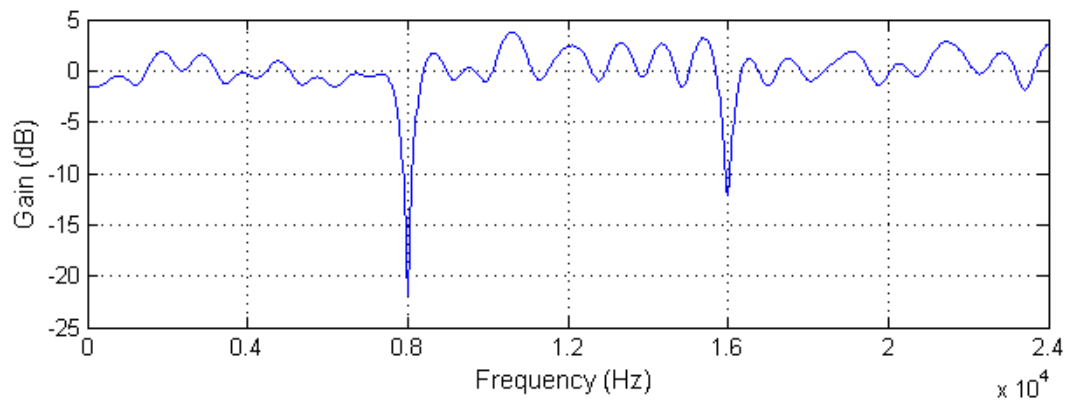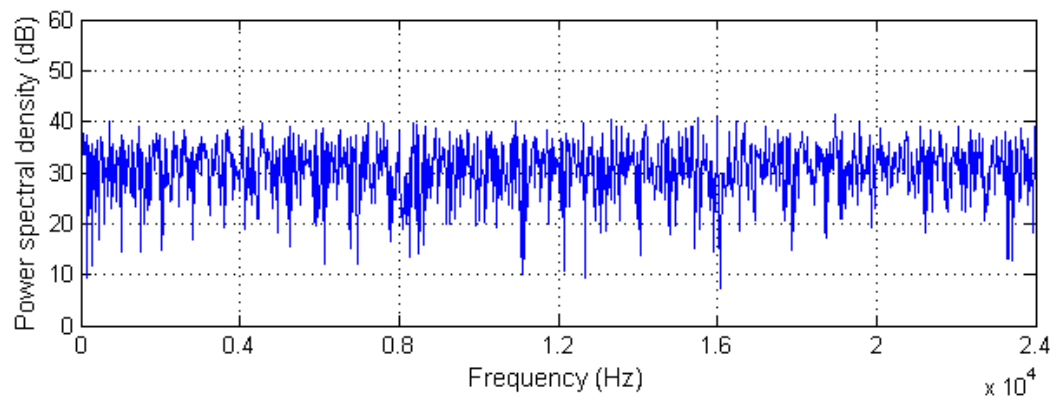

Figure 3. 'Whitened' noise (frame 1)

1.1.2 Adaptive filtering example (frame 2)

In Figure 4, a third interference tone appears at 12 kHz. The adaptive algorithm is run again to generate a new set of FIR coefficients, adding a third notch at the new tone frequency (Figure 5. Frequency response of adaptive noise filter (frame 2)). Once again, the result of filtering the input signal with the new filter coefficients can be seen in Figure 6. 'Whitened' noise (frame 2) .

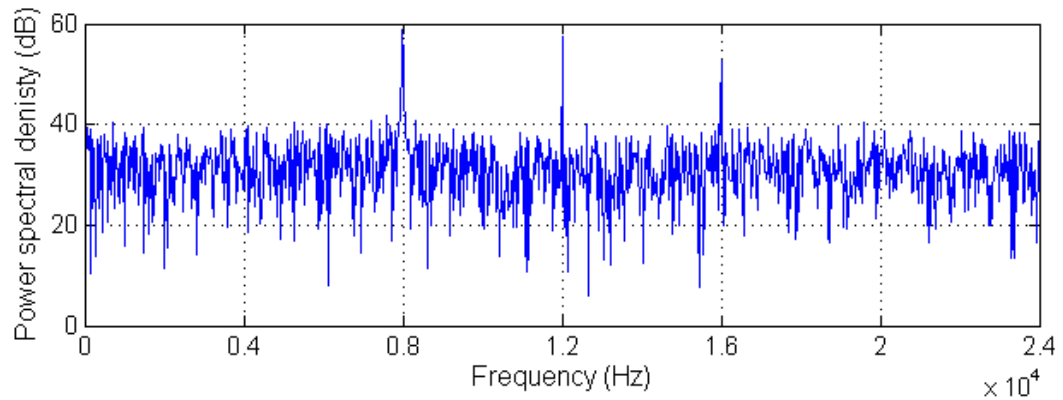**Figure 4. Gaussian noise with three-tone interferer**

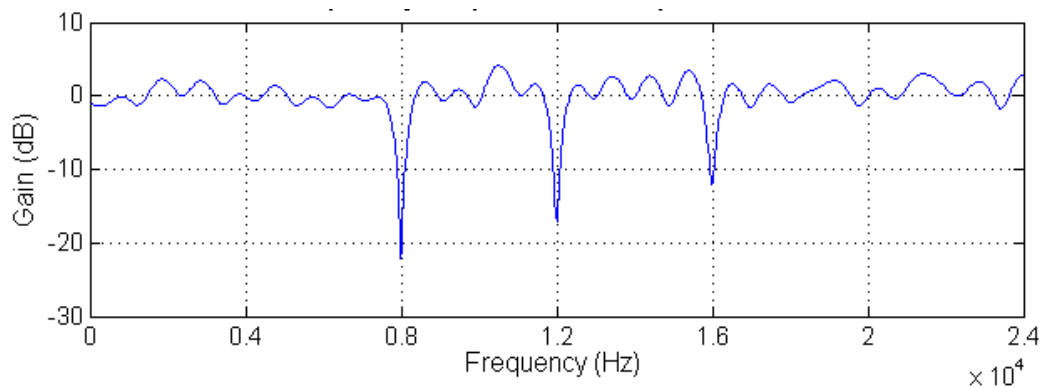**Figure 5. Frequency response of adaptive noise filter (frame 2)**
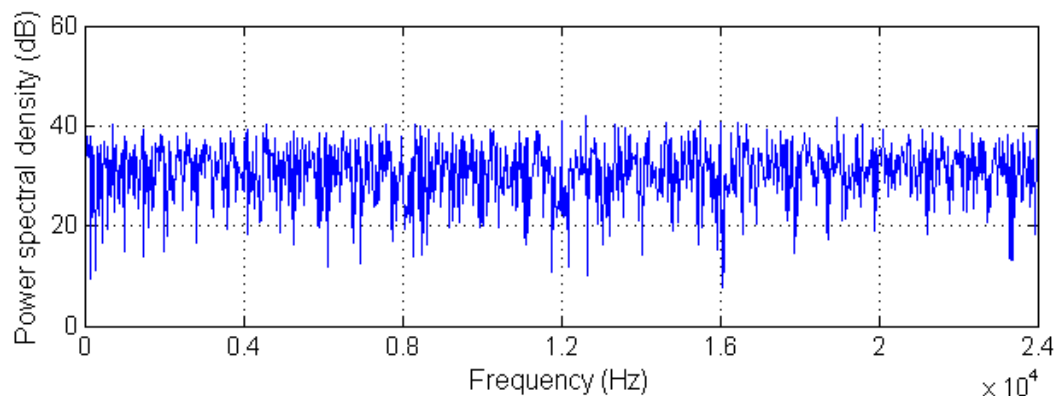

Figure 6. 'Whitened' noise (frame 2)

1.2 FIR filter

An FIR filter is essentially a convolution of the signal with the filter coefficients. The number of coefficients determines the minimum notch size with respect to the signal bandwidth. Having a large number of coefficients allows precise notching of narrow-band noise signals, thus minimising the amount of information lost. However calculating N coefficients typically requires in the order of N^2 operations, so the number of coefficients is limited by the available processing resources. In this example we use a 51-tap FIR filter, as illustrated in Figure 7.

Figure 7. 51-tap FIR filter
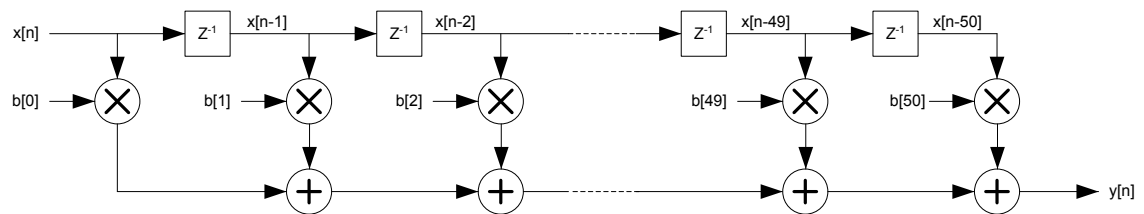
The filter coefficients are most likely calculated in floating point format and must be converted to 16-bit fixed point. The format used by the FMAC is q1.15, which means that bits 0 to 14 represent fractional digits (ie. to the right of the binary point), while bit 15 represents the sign/integer. Q1.15 format can therefore represent numbers in the range -1.0 (0x8000) to +0.999969482421875 (0x7FFF).

If it is properly designed, the adaptive algorithm should only generate coefficient absolute values less than or equal to 1, so all coefficients should be within the q1.15 numeric range. If this is not the case, then all coefficients must be divided by the largest coefficient value, or greater.

To convert from floating point to fixed point, all coefficients must be multiplied by 32768 (0x8000) and cast to int16:

```
value_q15 = (int16_t)(value_f32*0x8000)
```

The above operation on a cortex-M4 uses the processor floating point unit (FPU) and takes 8 clock cycles. So to convert 51 coefficients requires ~408 clock cycles.

The floating point and fixed point filter coefficients generated in our example are listed in Table 1 and Table 2 respectively (for frame 1).

Table 1. FIR coefficients (floating point)

| Coeff.	Value	Coeff.	Value	Coeff.	Value
b0	1.000000000000000	b1	-0.067585539745262	b2	0.006224959010140
b3	0.028277400466664	b4	0.095621684047959	b5	-0.025419309428324
b6	-0.118211581775912	b7	-0.017634551659068	b8	0.007942637819215
b9	0.010412979853874	b10	0.008068146389393	b11	-0.025829002206100
b12	-0.099047067935483	b13	0.013754178201556	b14	0.010326267975033
b15	-0.004410154604102	b16	0.018253517952308	b17	-0.020353758821992
b18	-0.024400399446690	b19	-0.014641218999051	b20	0.056344891227786
b21	0.002202080623664	b22	0.062681980108469	b23	0.010721748758013
b24	-0.076947777702560	b25	-0.013179271499833	b26	0.000291963084803
b27	0.032380282793325	b28	0.055274021304300	b29	-0.028443886878990
b30	-0.073984161617267	b31	0.018549937430136	b32	0.061223395867052
b33	0.005625129616435	b34	-0.012746336662958	b35	-0.014040525509958
b36	-0.006883769006464	b37	-0.033436800912042	b38	0.025923135119918
b39	-0.063274099314035	b40	0.075218785578469	b41	-0.017973648077274
b42	-0.083497943067555	b43	-0.012450048061703	b44	0.051849240421091
b45	-0.036429925307428	b46	0.004933033556656	b47	0.015810917169290
b48	-0.047438072229140	b49	0.006899397242167	b50	0.085755772557646

Table 2. FIR coefficients (q1.15 fixed point)

Coeff.	Value	Coeff.	Value	Coeff.	Value
b0	32767	b1	-2215	b2	203
b3	926	b4	3133	b5	-833
b6	-3874	b7	-578	b8	260
b9	341	b10	264	b11	-847
b12	-3246	b13	450	b14	338
b15	-145	b16	598	b17	-667
b18	-800	b19	-480	b20	1846
b21	72	b22	2053	b23	351
b24	-2522	b25	-432	b26	9
b27	1061	b28	1811	b29	-933
b30	-2425	b31	607	b32	2006
b33	184	b34	-418	b35	-461
b36	-226	b37	-1096	b38	849
b39	-2074	b40	2464	b41	-589
b42	-2737	b43	-408	b44	1698
b45	-1194	b46	161	b47	518
b48	-1555	b49	226	b50	2810

Due to the reduced number of bits available (16 instead of 24 for single precision floating point format), the performance of the filter is affected. This must be taken into account when considering whether to use the FMAC. If the loss of precision results in an unacceptable degradation of performance, a software implementation may be more appropriate (floating point or 32-bit fixed point).

In this example, the fixed point filter spectrum is identical to the floating point, to within 0.007 dB (Figure 8).

Figure 8. Difference between fixed point and floating point coefficients
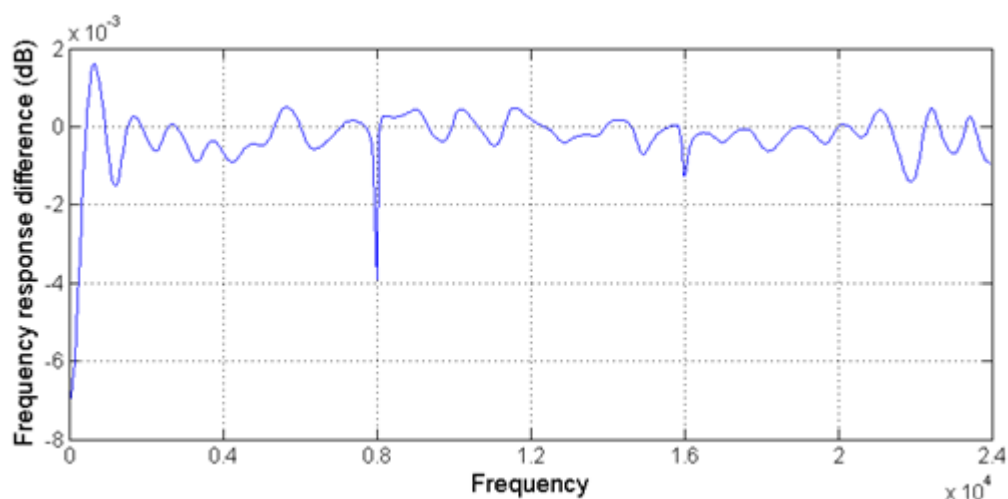
1.3 FMAC configuration

The FMAC can be configured using the HAL driver from the STM32CubeG4 MCU Package. Before accessing any FMAC registers, the FMAC clock must be enabled:

```
__HAL_RCC_FMAC_CLK_ENABLE();
```

An area of system memory must be reserved for the coefficients:

```
/* Declare an array to hold the filter coefficients */
```

```
static int16_t aFilterCoeffB[51];
```

We must also declare a structure to contain the FMAC parameters:

```
FMAC_HandleTypeDef hfmac;
```

Now we can configure the FMAC using the HAL_FMAC_FilterConfig() function:

```
/* declare a filter configuration structure */
FMAC_FilterConfigTypeDef sFmacConfig;
/* Set the coefficient buffer base address */
sFmacConfig.CoeffBaseAddress = 0;
/* Set the coefficient buffer size to the number of coeffs */
sFmacConfig.CoeffBufferSize = 51;
/* Set the Input buffer base address to the next free address */
sFmacConfig.InputBaseAddress = 51;
/* Set the input buffer size greater than the number of coeffs */
sFmacConfig.InputBufferSize = 100;
/* Set the input watermark to zero since we are using DMA */
sFmacConfig.InputThreshold = 0;
/* Set the Output buffer base address to the next free address */
sFmacConfig.OutputBaseAddress = 151;
/* Set the output buffer size */
sFmacConfig.OutputBufferSize = 100;
/* Set the output watermark to zero since we are using DMA */
sFmacConfig.OutputThreshold = 0;
/* No A coefficients since FIR */
sFmacConfig.pCoeffA = NULL;
sFmacConfig.CoeffASize = 0;
/* Pointer to the coefficients in memory */
sFmacConfig.pCoeffB = aFilterCoeffB;
/* Number of coefficients */
sFmacConfig.CoeffBSize = 51;
/* Select FIR filter function */
sFmacConfig.Filter = FMAC_FUNC_CONVO_FIR;
/* Enable DMA input transfer */
sFmacConfig.InputAccess = FMAC_BUFFER_ACCESS_DMA;
/* Enable DMA output transfer */
sFmacConfig.OutputAccess = FMAC_BUFFER_ACCESS_DMA;
/* Enable clipping of the output at 0x7FFF and 0x8000 */
sFmacConfig.Clip = FMAC_CLIP_ENABLED;
/* P parameter contains number of coefficients */
sFmacConfig.P = 51;
/* Q parameter is not used */
sFmacConfig.Q = FILTER_PARAM_Q_NOT_USED;
/* R parameter contains the post-shift value (none) */
sFmacConfig.R = 0;
/* Configure the FMAC */
if (HAL_FMAC_FilterConfig(&hfmac, &sFmacConfig) != HAL_OK)
/* Configuration Error */
Error_Handler();
```

The HAL_FMAC_FilterConfig() function programs the configuration and control registers, and loads the coefficients into the FMAC local memory (X2 buffer).

1.4 DMA configuration

Here, we configure three DMA channels:

Channel 1 - Preload channel. The preload channel is used to initialise the FMAC input buffer with samples. Each time the coefficients are changed, the FMAC must be stopped and restarted. The FMAC does not begin processing until its input buffer has been filled. So we use the last 51 samples of the previous frame to initialise the filter to the same state as when it was stopped, to avoid losing samples. This could be done by software, but in this example we use the DMA.

Channel 2 - Write channel. The write channel transfers input samples from memory to the FMAC, whenever the FMAC input buffer is not full.

Channel 3 - Read channel. The read channel transfers output samples from the FMAC to memory whenever there are unread samples in the FMAC output buffer.

The following code shows how to configure the DMA channels. We declare three structures to hold the channel parameters:

```
DMA_HandleTypeDef hdma_fmac_preload; /* Preload channel */
DMA_HandleTypeDef hdma_fmac_read; /* Read channel */
DMA_HandleTypeDef hdma_fmac_write; /* Write channel */
```

We then initialise the structures which are passed to the HAL_DMA_Init() function:

```
/* Preload channel initialisation */
hdma_fmac_preload.Instance = DMA1_Channel1;
hdma_fmac_preload.Init.Request = DMA_REQUEST_MEM2MEM;
hdma_fmac_preload.Init.Direction = DMA_MEMORY_TO_MEMORY;
hdma_fmac_preload.Init.PeriphInc = DMA_PINC_ENABLE;
hdma_fmac_preload.Init.MemInc = DMA_MINC_DISABLE;
hdma_fmac_preload.Init.PeriphDataAlignment =
DMA_PDATAALIGN_HALFWORD;
hdma_fmac_preload.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
hdma_fmac_preload.Init.Mode = DMA_NORMAL;
hdma_fmac_preload.Init.Priority = DMA_PRIORITY_HIGH;
if (HAL_DMA_Init(&hdma_fmac_preload) != HAL_OK)
Error_Handler();

/* Connect the DMA channel to the FMAC handle */
__HAL_LINKDMA(hfmac,hdmaPreload,hdma_fmac_preload);

/* Write channel initialisation */
hdma_fmac_write.Instance = DMA1_Channel2;
hdma_fmac_write.Init.Request = DMA_REQUEST_FMAC_WRITE;
hdma_fmac_write.Init.Direction = DMA_MEMORY_TO_PERIPH;
hdma_fmac_write.Init.PeriphInc = DMA_PINC_DISABLE;
hdma_fmac_write.Init.MemInc = DMA_MINC_ENABLE;
hdma_fmac_write.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
hdma_fmac_write.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
hdma_fmac_write.Init.Mode = DMA_NORMAL;
hdma_fmac_write.Init.Priority = DMA_PRIORITY_HIGH;
if (HAL_DMA_Init(&hdma_fmac_write) != HAL_OK)
Error_Handler();

/* Connect the DMA channel to the FMAC handle */
__HAL_LINKDMA(hfmac,hdmaIn,hdma_fmac_write);

/* Read channel initialisation */
hdma_fmac_read.Instance = DMA1_Channel3;
hdma_fmac_read.Init.Request = DMA_REQUEST_FMAC_READ;
hdma_fmac_read.Init.Direction = DMA_PERIPH_TO_MEMORY;
hdma_fmac_read.Init.PeriphInc = DMA_PINC_DISABLE;
hdma_fmac_read.Init.MemInc = DMA_MINC_ENABLE;
hdma_fmac_read.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
hdma_fmac_read.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
hdma_fmac_read.Init.Mode = DMA_NORMAL;
hdma_fmac_read.Init.Priority = DMA_PRIORITY_HIGH;
if (HAL_DMA_Init(&hdma_fmac_read) != HAL_OK)
```

```
Error_Handler();

/* Connect the DMA channel to the FMAC handle */
__HAL_LINKDMA(hfmac,hdmaIn,hdma_fmac_read);
```

The interrupts for the read channel and the preload channel should be enabled, so that the software can be informed when the transfer has completed.

1.5 Running the filter

The filter is triggered by software each time there is a frame's worth of new samples ready for filtering. The new samples are stored in system memory, in a double array:

```
static int16_t aInputValues[2][2048];
int CurrentInputArraySize = 2048;
```

The double array allows a new frame of samples to be received and stored in memory while the previous frame is being processed. We need an index to select which frame is currently in use:

```
int Frame = 1;
```

We also need an array to store the output data:

```
static int16_t aCalculatedFilteredData[2048];
int ExpectedCalculatedFilteredDataSize = 2048;
```

The software triggers the preload of the last 51 samples of the previous frame. This is to restore the filter state after the FMAC has been stopped:

```
if (HAL_FMAC_FilterPreload_DMA(&hfmac,
&aInputValues[CurrentInputArray][1997], 51, NULL, 0) != HAL_OK)
Error_Handler();
/* Switch frames */
Frame ? Frame=0 : Frame=1;
```

When the preload has finished (signalled by a DMA channel 1 terminal count interrupt), the software triggers the DMA write channel to begin transferring the samples from the current frame:

```
if (HAL_FMAC_AppendFilterData(&hfmac, &aInputValues[CurrentInputArray][0],
&CurrentInputArraySize) != HAL_OK)
 Error_Handler();
```

The software starts the FMAC:

```
if (HAL_FMAC_FilterStart(&hfmac, aCalculatedFilteredData,
&ExpectedCalculatedFilteredDataSize) != HAL_OK)
 Error_Handler();
```

The FMAC starts calculating the output samples and writes them in the output buffer. At each new sample the FMAC generates a DMA request on the read channel, and the DMA transfers the new sample to system memory.

This continues without software intervention until the entire frame has been processed, at which point the DMA read channel generates a terminal count interrupt to the processor. The processor stops the FMAC:

```
if (HAL_FMAC_FilterStop(&hfmac) != HAL_OK)
  Error_Handler();
```

The software can now update the coefficients, if necessary, by calling HAL_FMAC_FilterConfig() again. The new coefficients should previously have been stored in aFilterCoeffB[];

To process the next frame, the above procedure is simply repeated, starting from the preload (HAL_FMAC_FilterPreload_DMA()).

2 Example 2: 3p3z compensator

2.1 3p3z overview

DC-DC power supplies require a control loop to maintain output stability and precision under varying load conditions. Traditionally this is performed using analog components, but more and more manufacturers are turning to digital solutions, very often using a general purpose microcontroller which offers the advantage of programmability and low cost. The microcontroller can combine many functions on one device, in addition to the control loop, such as display controller, remote control and status communication, security and safety monitoring, keypad, and so on.

Nevertheless, the control loop task can occupy a significant amount of processor time at high switching frequencies, and is high priority due to the real time constraints - indeed the loop must be executed within one switching period or the controller can go unstable. The FMAC allows a part of the controller task, namely the digital compensator, to be offloaded, thus freeing up processor time for other tasks.

The compensator ensures a sufficiently high phase and gain margin, so that the controller remains stable. They are generally made with IIR filters, which achieve steep phase slopes over frequency even with low order filters, compared to FIR implementations.

This example implements a voltage mode controller for a synchronous buck converter. The compensator is a 3-pole 3-zero (3p3z) IIR filter. The example code runs on a Discovery kit (B-G474E-DPOW1). The code uses the STM32G4xx-HAL-Driver library from the STM32CubeG4 MCU Package firmware.

2.2 Synchronous buck converter

The buck converter schematic is given in Figure 9.

Figure 9. Buck Converter
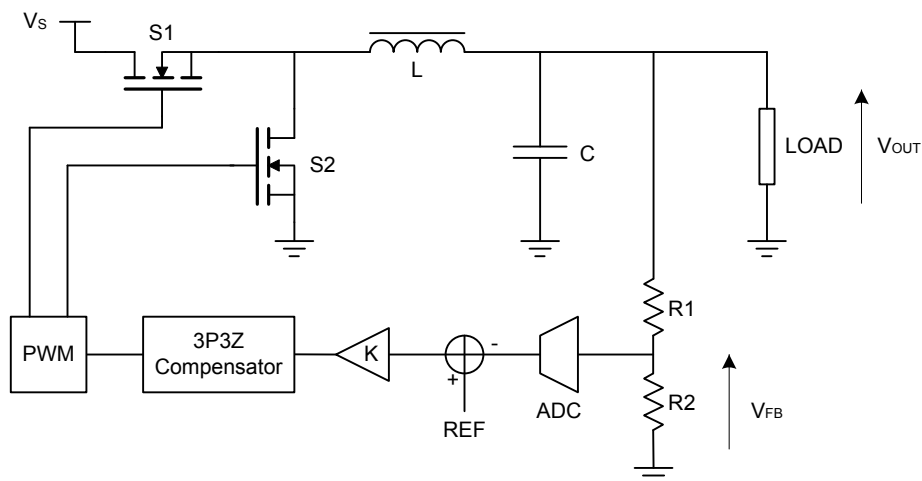
Transistors S1 and S2 act as switches and operate in anti-phase. When S1 is closed, and S2 open, a steadily increasing current flows from the supply, V_S, through S1 and inductor L, to charge capacitor C. After a time T1, switch S1 opens and S2 closes. The current through L continues to flow, but now it is supplied from the ground via switch S2, and steadily decreases. After time T2, S2 is re-opened and S1 closed.

The gates of S1 and S2 are driven by a complementary square wave, with a fixed period T1+T2 and a variable duty cycle T1/(T1+T2) - see Figure 10. Buck converter waveforms. This is called a PWM (Pulse Width Modulation) signal. By increasing T1 and reducing T2, the current through L can be increased, charging the capacitor C to a higher voltage. By reducing T1 and increasing T2, the current though L and hence the voltage on C is decreased. Hence the output voltage of the Buck converter can be regulated by controlling the duty cycle of the PWM signal.

Figure 10. Buck converter waveforms

1. Increasing output voltage
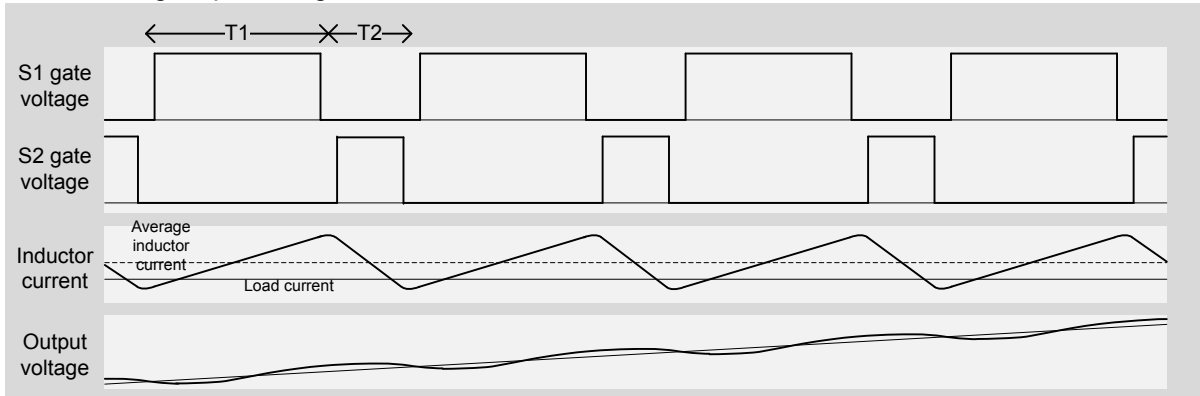

2. Decreasing output voltage
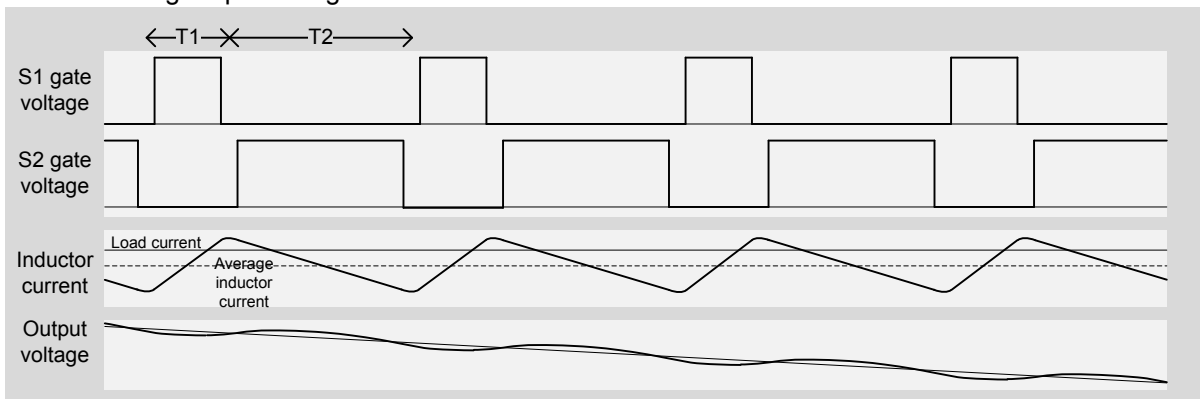
2.3 Digital controller

Under varying supply (line) and load conditions, the PWM duty cycle must be continuously adjusted in order to maintain a constant output voltage. The output voltage, divided by the resistor ladder R1,R2, is fed back via a controller to adjust the duty cycle. This voltage control loop is implemented on the microcontroller. It comprises an analog-to-digital converter (ADC), whose output is subtracted from a reference. The reference is the ADC value corresponding to the target output voltage. So the result of the subtraction represents the error between the actual output voltage and the expected.

This error, after amplification and compensation, sets the duty cycle of the PWM signal used to control the switches S1 and S2. A positive error (REF > ADC output) means that the output voltage is lower than the target and the duty cycle must be increased. A negative error indicates an over-voltage condition on the output and the duty cycle must be reduced.

The A-to-D conversion and PWM stages are implemented in the microcontroller using the ADC and High Resolution Timer hardware blocks. The intermediate stages are generally performed in software. However, in the example presented here, the FMAC is used to implement the compensator, while the reference subtraction and amplification are included in the ADC. The software is only required to condition the compensator output for PWM. The output of the ADC can be transferred directly to the FMAC input by DMA.

The sample flow through the different microcontroller functional units is illustrated in Figure 11. Controller functional diagram.

Figure 11. Controller functional diagram
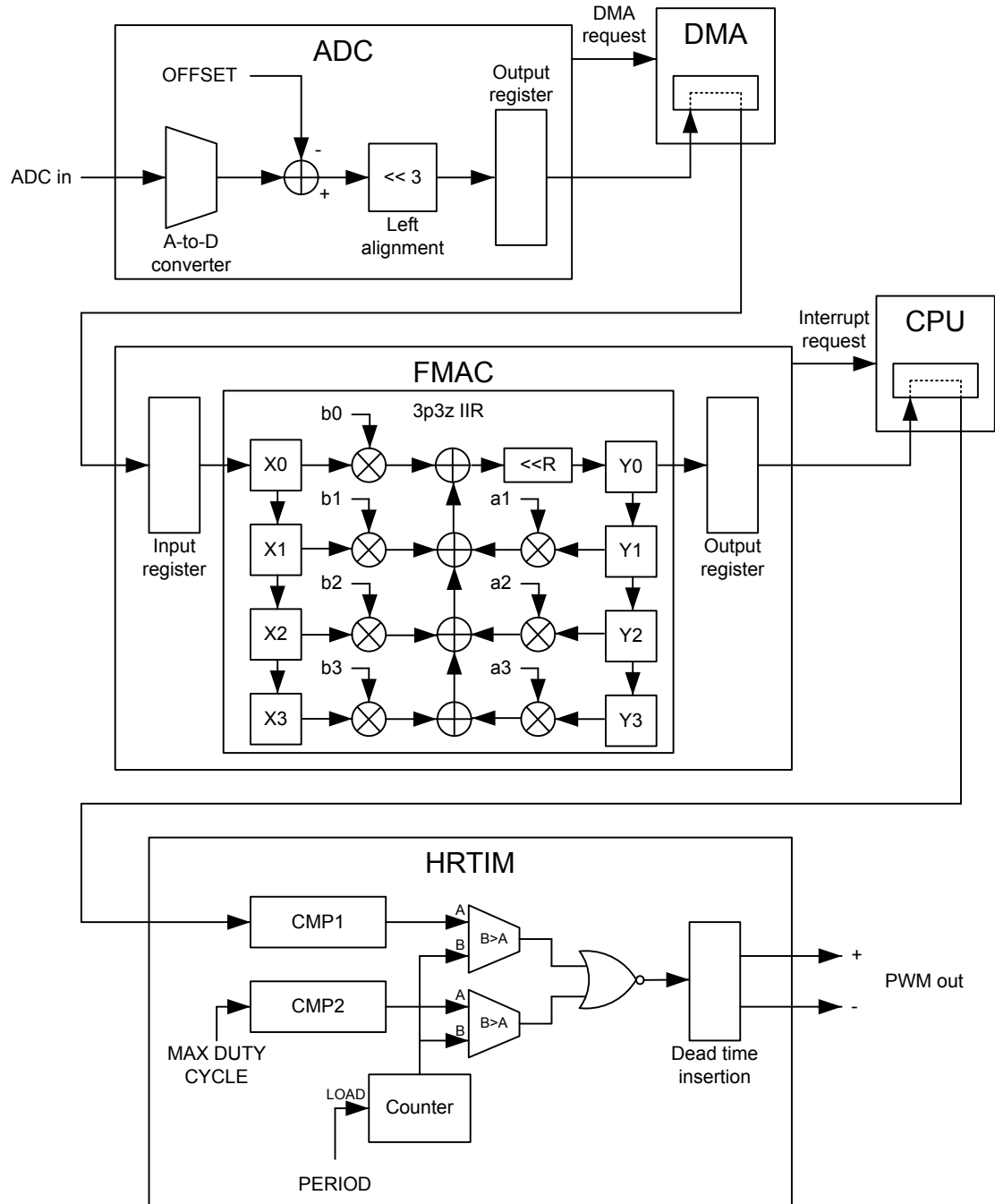
2.4 Buck converter specification

The converter in the example is designed according to the following specification:

- Input supply voltage: 5 V
- Output voltage: 3.3 V
- Maximum current: 0.5 A
- Target ripple: 0.5% (16.5 mV)
- Overshoot (50% load step): 5 mV
- Control mode: Voltage, digital
- Switching frequency: 200 kHz
- Sampling frequency: 200 kHz
- Crossover frequency: 8 kHz
- Phase margin at crossover: 50 degrees
- Duty cycle limit: 90%

The switches on the Discovery kit have the following characteristics:

- Primary switch resistance: 56 mΩ
- Primary rise time: 20 ns
- Primary fall time: 20 ns
- Parasitic capacitance: 79 pF
- Secondary voltage drop: 0.02 V

Output filter characteristics:

- Ripple: 0.125 A pk-pk
- L inductance: 51 µH
- L DCR: 380 mW
- C capacitance: 100 µF
- C ESR: 170 mW

The ADC and PWM characteristics are as follows:

- PWM master clock frequency: 5440 MHz (HR Timer at 170 MHz, x32 resolution)
- PWM period count: 27200 (for 200 kHz frequency)
- ADC resolution: 12 bits
- ADC full scale voltage: 3.3 V
- ADC input voltage divider ratio R2/(R1+R2): 0.19

The ADC full-scale digital code is $2^{12}-1 = 4095$. Hence the reference value (REF) corresponding to the target output voltage of 3.3 V is 0.19 x 4095 = 778.

2.5 Controller design

The voltage controller is based on a type III analog compensator. In the digital domain this is approximated using a 3rd order IIR filter, which has three poles and three zeros, hence the name 3p3z compensator. The placement of the three poles and two of the zeros are done in the analog domain. The transfer function is then converted to the digital domain using a bilinear transform, which inserts the extra zero.
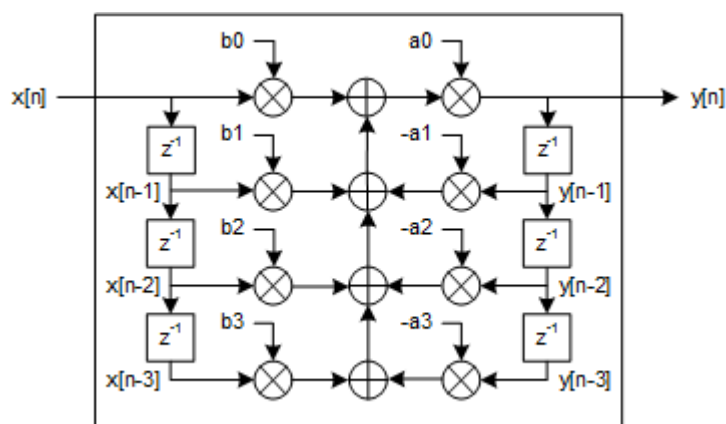
The digital compensator transfer function looks like this:

$$H(z) = \frac{b0 + b1 \cdot z^{-1} + b2 \cdot z^{-2} + b3 \cdot z^{-3}}{a0 + a1 \cdot z^{-1} + a2 \cdot z^{-2} + a3 \cdot z^{-3}}$$

This corresponds to the following difference equation:

$$a0 \cdot y[n] = b0 \cdot x[n] + b1 \cdot x[n-1] + b2 \cdot x[n-2] + b3 \cdot x[n-3] \\ - a1 \cdot y[n-1] - a2 \cdot y[n-2] - a3 \cdot y[n-3]$$

Schematically, the above can be represented as shown in Figure 14. This is known as a direct form 1 implementation of a digital IIR filter.

Figure 14. 3p3z compensator

The boxes marked z^{-1} represent delays of one sample period. The feed-forward coefficients b0, b1, b2 and b3 are multiplied with the current and delayed input samples, x[n], x[n-1], x[n-2] and x[n-3] respectively. The feed-back coefficients -a1, -a2 and -a3 are multiplied with the delayed output samples y[n-1], y[n-2] and y[n-3] respectively. The results of these multiplications are summed together to obtain the next output sample, y[n]. The coefficient a0 is usually 1, but may be set to another value if necessary - see Section 2.5.3 Normalising the coefficients.

2.5.1 Calculating the coefficients

The coefficients can be calculated by hand, or using an appropriate software tool. In this example, Biricha WDS is used to generate the coefficients starting from the buck converter specifications and the analog ADC and PWM characteristics listed in Section 2.4 Buck converter specification. The coefficients in floating point format can be obtained from the WDS tool as shown in Figure 15.

Figure 15. Floating point coefficients

A1	1.521558814252
A2	-0.356458881462
A3	-0.16509993279
K	115.36533642

B0	1.553498447795
B1	-1.361492224301
B2	-1.5476128749966
B3	1.36737779713

Note: *the a0 coefficient is implicit and has a value of 1, and the coefficients A1, A2 and A3 are already negated by the tool, that is: -a1 = A1, -a2 = A2 and -a3 = A3.*

2.5.2 Gain compensation

Next, we need to compensate the gain of the feedback path. In this example, there are three gain factors which intervene:

1. The ADC input voltage divider ratio R2/(R1+R2): 0.19
2. The ADC conversion gain which is the full-scale digital output divided by the full scale reference voltage: 4095/3.3
3. The PWM gain normalised to 1V full scale ie. 1V divided by the PWM period count in clock ticks: 1/27200

The combined gain is the product of the above three factors. So to obtain unity gain in the feedback path, we need to apply a compensatory gain K as shown in Figure 9. Buck Converter, where K = (1/0.19) x (3.3/4095) x 27200 = 115.37. This is the meaning of the factor K calculated by Biricha WDS in Figure 15. Floating point coefficients.

The gain can be applied in software at the output of the FMAC. However to save a bit more CPU time, it can be applied in hardware as follows. The factor K is divided into two, K_{ADC} and K_{filter}, where K_{ADC} x K_{filter} = K.

In the ADC, the factor K_{ADC} = 8 is applied using the left justification feature. The ADC result (after subtracting the offset) is a 12-bit signed integer. In right-aligned format, the result is sign-extended to 16-bits:

Table 3. Right aligned ADC data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sign	sign	sign	sign	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

In left aligned mode the result is shifted left by 3, retaining only one sign bit:

Table 4. Left aligned ADC data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sign	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	0	0	0

Hence by left aligning the ADC result we have multiplied it by 8.

The remaining gain factor, K_{filter} = K/K_{ADC} = 115.37/8 = 14.42, can be applied in the filter by multiplying the numerator coefficients (see Table 5. Scaling, normalisation and conversion to integer).

It would be possible to apply the entire gain factor, K, in the filter. However for reasons explained in the next section, this would require the denominator coefficients to be scaled down by 128, resulting in a significant loss of precision in the positioning of the poles and potentially rendering the loop unstable.

2.5.3 Normalising the coefficients

As previously mentioned, the FMAC uses fixed point arithmetic, and the coefficients must be 'normalised'. This means that they are scaled to fit the numeric range required by the hardware. In the case of the FMAC, the coefficients are defined in q1.15 format (1 sign bit, 15 fractional bits), which has a numeric range of from -1, represented by the two's complement integer value -32768 (0x8000), to $+1 - 2^{-15}$, represented by the two's complement integer value 32767 (0x7FFF).

Several of the floating point coefficients are greater than 1 or less than -1, and so all the coefficients must be scaled such that the magnitude of the largest coefficient is less than 1. Ideally, the scaling factor, k_{norm}, should be chosen to use the maximum precision available from the fixed point format. In other words, if the magnitude of the largest coefficient is |B0|, then applying a scaling factor k_{norm} of 1/|B0| would result in a new maximum coefficient magnitude of 1. However in the FMAC the scaling factor used for the denominator is restricted to negative powers of two (1, 1/2, 1/4, ...). The reason for this is linked to the specific nature of the a0 coefficient.

Consider the transfer function for the 3p3z compensator again:

$$H(z) = \frac{b0 + b1 \cdot z^{-1} + b2 \cdot z^{-2} + b3 \cdot z^{-3}}{a0 + a1 \cdot z^{-1} + a2 \cdot z^{-2} + a3 \cdot z^{-3}}$$

If a scaling factor, k_{norm}, is applied to the denominator, the transfer function becomes:

$$\frac{Y(z)}{X(z)} = \frac{b0 + b1 \cdot z^{-1} + b2 \cdot z^{-2} + b3 \cdot z^{-3}}{k_{norm} \cdot (a0 + a1 \cdot z^{-1} + a2 \cdot z^{-2} + a3 \cdot z^{-3})}$$

This corresponds to the following difference equation:

$$k_{norm} \cdot a0 \cdot y[n] = b0 \cdot x[n] + b1 \cdot x[n-1] + b2 \cdot x[n-2] + b3 \cdot x[n-3]$$
$$- k_{norm} \cdot a1 \cdot y[n-1] - k_{norm} \cdot a2 \cdot y[n-2] - k_{norm} \cdot a3 \cdot y[n-3]$$

The output of the filter is therefore scaled by k_{norm}, as are the delayed output samples. To compensate the scaling, coefficient a0 must be set to $1/k_{norm}$. The left hand side of the above equation then reduces to y[n].

In the FMAC, coefficient a0 is applied using a left shift, so it can only take the values 1, 2, 4,... Thus only values for k_{norm} of 1, 1/2, 1/4,...etc can be used.

If, as described above, only the denominator coefficients are scaled, the filter gain is multiplied by $1/k_{norm}$. To avoid this, the numerator coefficients (b0, b1, b2, b3) should be scaled by the same factor as the denominator.

In this example, B0 = 1.553 is the largest coefficient. After multiplying by the residual gain compensation, K_{filter}, we get B0' = B0 x 14.42 = 22.40. The lowest value of n for which B0' x $2^{-n} < 1$ is n = 5, giving $k_{norm} = 1/32$. We therefore divide all the coefficients by 32, as shown in Table 5. Scaling, normalisation and conversion to integer. A0 is set to 32, corresponding to a left shift by 5.

2.5.4 Conversion to fixed point integer format

Now that the coefficients are normalised, we have to convert them to 16-bit integer format. Bearing in mind that a value of 1 in q1.15 corresponds to an integer value of 32768, we simply multiply all the coefficients by 32768 and round them to the nearest integer. Table 5 shows the integer values of the coefficients, as well as their intermediate values after each of the steps described above.

Table 5. Scaling, normalisation and conversion to integer

Coeff	Raw values from Biricha WDS	Scale B coeffs to compensate gain (x14.42)	Normalise (x1/32)	Convert to integer (x 32768)
B0	1.553498447795	22.402483882227	0.700077621320	22940 (0x599C)
B1	-1.361492224301	-19.633626061212	-0.613550814413	-20105 (0xB177)
B2	-1.547612874966	-22.317609996047	-0.697425312376	-22853 (0xA6BB)
B3	1.367377797130	19.718499947393	0.616203123356	20192 (0x4EE0)
A1	1.521558814252	1.521558814252	0.047548712945	1558 (0x0616)
A2	-0.356458881462	-0.356458881462	-0.011139340046	-365 (0xFE93)
A3	-0.165099932790	-0.165099932790	-0.005159372900	-169 (0xFF57)

Note that the conversion to integer introduces a rounding error which can have a significant impact on the transfer function of the filter.

The coefficients are defined as constants:

```
#define B0 (0x599C)
#define B1 (0xB177)
#define B2 (0xA6BB)
#define B3 (0x4EE0)
#define A1 (0x0616)
#define A2 (0xFE93)
#define A3 (0xFF57)
```

We also need to define the left shift, $\log_2 A0 = 5$:

```
#define KC_shift (5)
```

2.5.5 Bode plots

The Bode plots of the plant (buck converter), the compensator and the loop response (product of the two) are shown in Figure 19. Bode plots (obtained from the "Frequency Response"tab of Biricha WDS). The compensator performs the following functions:

1. Sets the zero-gain crossing frequency of the loop, Fx, at 8 kHz
2. Sets the phase margin at Fx to 50 dB
3. Sets the gain slope at Fx to -20 dB/decade

These can be verified by inspection of the Bode plot.

Figure 19. Bode plots
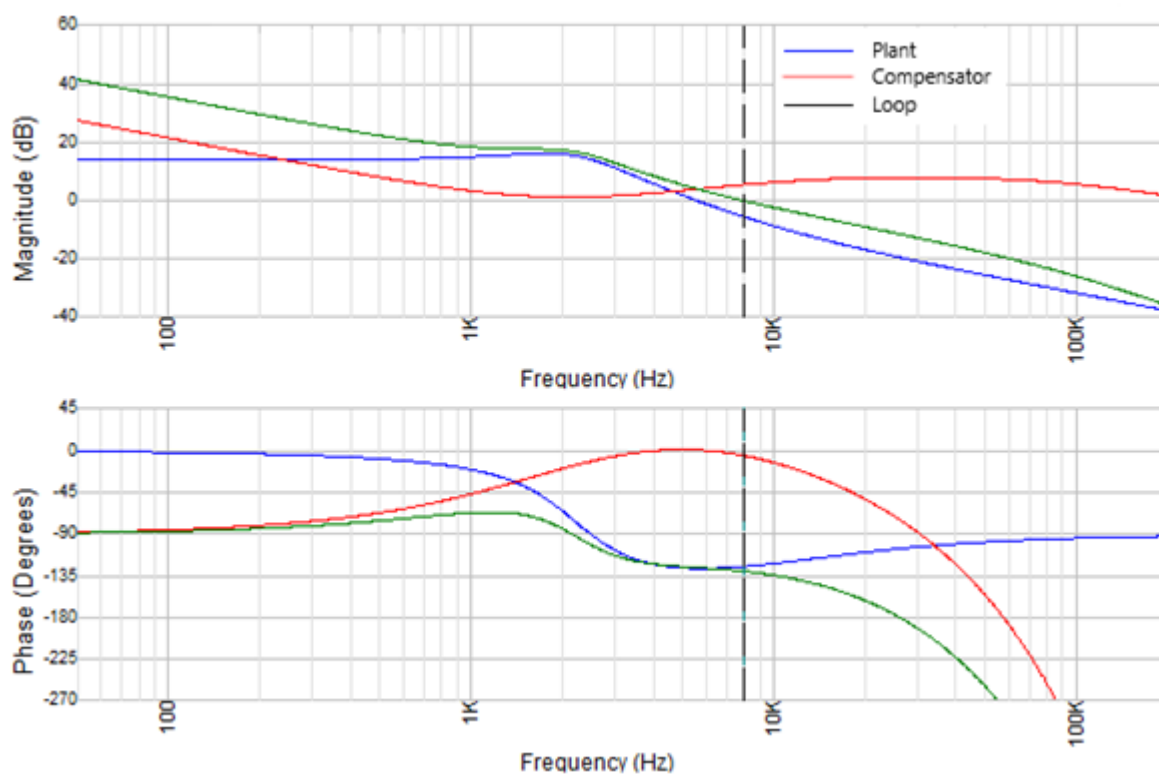
2.5.6 Compensator output and PWM duty cycle

The compensator output is the requested PWM duty cycle. It could be transferred directly to the High Resolution Timer by DMA. However the output range of the FMAC (with saturation enabled) is -32768 (0x8000) to +32767 (0xFFFF). The HR Timer duty cycle is constrained in the range 0% to 90%. 100% duty cycle corresponds to the period at 200kHz, which is 27200 master clock ticks (see Section 2.4 Buck converter specification), so the max duty cycle at 90% is 24480 ticks, defined as a constant:

```
#define DUTY_TICKS_MAX 24480
```

The gain of the compensator for most frequencies is >0dB. This is useful to ensure rapid convergence, however the disadvantage is that the output can overshoot the max duty cycle, or go negative, leading to unexpected behaviour of the PWM signal. For safety therefore the FMAC is configured to generate an interrupt to the processor, which reads the duty cycle from the FMAC output (RDATA register) and limits it to > 0 before transferring it to the HRTimer comparator 1:

```
void FMAC_IRQHandler(void) /* FMAC interrupt handler */
{
int32_t tmp;
tmp = READ_REG(hfmac.Instance->RDATA);
tmp = (tmp > 0x00007FFF ? 0 : tmp);
 __HAL_HRTIM_SETCOMPARE(&hhrtim1, HRTIM_TIMERINDEX_TIMER_C, HRTIM_COMPAREUNIT_1, tmp);}
```

The intervention of the CPU at this point also allows the software to detect and correct faults.

The maximum duty cycle limit is applied in the HRTimer itself, using a second comparator which is programmed to reset the PWM output if the counter reaches DUTY_TICKS_MAX.

2.6 Configuring the FMAC

Before accessing any FMAC registers, the FMAC clock must be enabled:

```
__HAL_RCC_FMAC_CLK_ENABLE();
```

An area of system memory must be reserved for the A and B coefficients:

```
/* Array of filter coefficients A (feedback taps) in Q1.15 format */
static int16_t aFilterCoeffA[COEFF_VECTOR_A_SIZE] = {A1,A2,A3};
/* Array of filter coefficients B (feed-forward taps) in Q1.15 format */
static int16_t aFilterCoeffB[COEFF_VECTOR_B_SIZE] = {-B0,-B1,-B2,-B3};
```

Note: *The B coefficients are negated. This is to compensate for the fact that the ADC subtracts the offset (reference) from the conversion result, rather than the other way round, meaning that the error signal at the compensator input is inverted.*

We must also declare a structure to contain the FMAC parameters:

```
FMAC_HandleTypeDef hfmac;
```

First we enable the FMAC interrupt in the interrupt controller (NVIC):

```
HAL_NVIC_SetPriority(FMAC_IRQn, 1, 0);
HAL_NVIC_EnableIRQ(FMAC_IRQn);
```

Now we can configure the FMAC using the HAL_FMAC_FilterConfig() function:

```
/* declare a filter configuration structure */
FMAC_FilterConfigTypeDef sFmacConfig;
/* Set the coefficient buffer base address */
sFmacConfig.CoeffBaseAddress = 0;
/* Set the coefficient buffer size to the number of coeffs */
sFmacConfig.CoeffBufferSize = 7;
/* Set the Input buffer base address to the next free address */
sFmacConfig.InputBaseAddress = 7;
/* Set the input buffer size greater than the number of coeffs */
sFmacConfig.InputBufferSize = 5;
/* Set the input watermark to zero since we are using DMA */
sFmacConfig.InputThreshold = 0;
/* Set the Output buffer base address to the next free address */
sFmacConfig.OutputBaseAddress = 12;
/* Set the output buffer size greater than the number of coeffs */
sFmacConfig.OutputBufferSize = 5;
/* Set the output watermark to zero since we are using DMA */
sFmacConfig.OutputThreshold = 0;
/* Pointer to A coefficients in memory*/
sFmacConfig.pCoeffA = aFilterCoeffA;
/* Number of A coefficients */
sFmacConfig.CoeffASize = 3;
/* Pointer to Bcoefficients in memory */
sFmacConfig.pCoeffB = aFilterCoeffB;
/* Number of B coefficients */
sFmacConfig.CoeffBSize = 4;
/* Select IIR filter function */
sFmacConfig.Filter = FMAC_FUNC_IIR_DIRECT_FORM_1;
/* Disable DMA and interrupt requests for input */
sFmacConfig.InputAccess = FMAC_BUFFER_ACCESS_NONE;
/* Enable FMAC read interrupt for output transfer */
sFmacConfig.OutputAccess = FMAC_BUFFER_ACCESS_IT;
/* Enable clipping of the output at 0x7FFF and 0x8000 */
sFmacConfig.Clip = FMAC_CLIP_ENABLED;
/* P parameter contains number of B coefficients */
sFmacConfig.P = 4;
/* Q parameter contains number of A coefficients */
sFmacConfig.Q = 3;
/* R parameter contains the post-shift value */
```

```
sFmacConfig.R = KC_shift;
/* Configure the FMAC */
if (HAL_FMAC_FilterConfig(&hfmac, &sFmacConfig) != HAL_OK)
/* Configuration Error */
Error_Handler();
```

The HAL_FMAC_FilterConfig() function programs the configuration and control registers, and loads the coefficients into the FMAC local memory (X2 buffer).

It is recommended to preload the Y buffer with zeros to avoid unpredictable transient values at the outset.

```
/* Array of output data to preload in Q1.15 format */
static int16_t aOutputDataToPreload[COEFF_VECTOR_A_SIZE] = {0x0000,
0x0000, 0x0000};
```

```
if (HAL_FMAC_FilterPreload(&hfmac, NULL, INPUT_BUFFER_SIZE,
aOutputDataToPreload, COEFF_VECTOR_A_SIZE) != HAL_OK)
/* Configuration Error */
Error_Handler();
```

2.7 Configuring the ADC

The ADC can be configured using the HAL driver from the STM32Cube-FW-G4 Package.

Before accessing any ADC registers, the ADC clock must be enabled:

```
__HAL_RCC_ADC12_CLK_ENABLE();
```

We create a structure for the ADC parameters and initialise it using the HAL_ADC_Init() function:

```
ADC_HandleTypeDef hadc1;
hadc1.Instance = ADC1;
/* ADC max allowed clock frequency is 80MHz */
/* ADC clock configured to use asynchronous clock */
hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
/* 12-bit samples */
hadc1.Init.Resolution = ADC_RESOLUTION_12B;
/* left aligned data equivalent to 8dB gain */
hadc1.Init.DataAlign = ADC_DATAALIGN_LEFT;
hadc1.Init.GainCompensation = 0;
hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.ContinuousConvMode = DISABLE;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DiscontinuousConvMode = DISABLE;
/* Conversion triggered by HRTimer Trig 1 rising edge */
hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG_HRTIM_TRG1;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
/* Allow ADC to generate DMA requests after every conversion */
hadc1.Init.DMAContinuousRequests = ENABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
_Error_Handler();
```

We need to declare a structure for the ADC channel configuration:

```
ADC_ChannelConfTypeDef sConfig;
```

and initialise it using the HAL_ADC_ConfigChannel() function:

```
sConfig.Channel = ADC_CHANNEL_1;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
/* Enable offset feature */
sConfig.OffsetNumber = ADC_OFFSET_1;
/* Set offset to fixed reference */
sConfig.Offset = REF;
/* Offset is subtracted from conversion result */
sConfig.OffsetSign = ADC_OFFSET_SIGN_NEGATIVE;
sConfig.OffsetSaturation = DISABLE;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
_Error_Handler();
```

The offset is the reference. It is defined as a constant in the code: `#define REF (778)`

Next, we declare and initialise a structure for the DMA channel, to be used to transfer the error value from the ADC output to the FMAC input:

```
DMA_HandleTypeDef hdma_adc;
hdma_adc.Instance = DMA1_Channel1;
/* DMA request comes from the ADC */
hdma_adc.Init.Request = DMA_REQUEST_ADC1;
/* Peripheral (source) is ADC; Memory (dest) is FMAC */
hdma_adc.Init.Direction = DMA_PERIPH_TO_MEMORY;
hdma_adc.Init.PeriphInc = DMA_PINC_DISABLE;
hdma_adc.Init.MemInc = DMA_MINC_DISABLE;
hdma_adc.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
hdma_adc.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
hdma_adc.Init.Mode = DMA_CIRCULAR;
hdma_adc.Init.Priority = DMA_PRIORITY_HIGH;
if (HAL_DMA_Init(&hdma_adc) != HAL_OK)
Error_Handler();
/* Connect the DMA channel to the ADC structure */
__HAL_LINKDMA(&hadc1,DMA_Handle,hdma_adc);
```

2.8 Configuring the high-resolution timer

We first create a structure for the HR Timer parameters:

```
HRTIM_HandleTypeDef hhrtim1;
```

The HRTimer is configured to generate a differential PWM signal at 200 kHz. One counter is configured, Timer A, running at 32 x 170 MHz = 5440 MHz.

```
HRTIM_TimeBaseCfgTypeDef pTimeBaseCfg;
pTimeBaseCfg.Period = 27200; /* 200kHz (32 * 170 / 0.2 = 27200) */
pTimeBaseCfg.RepetitionCounter = 0x00;
pTimeBaseCfg.PrescalerRatio = HRTIM_PRESCALERRATIO_MUL32;
pTimeBaseCfg.Mode = HRTIM_MODE_CONTINUOUS;
if (HAL_HRTIM_TimeBaseConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_C,
&pTimeBaseCfg) != HAL_OK)
    _Error_Handler();
```

We need to enable three compare units, used respectively to:

1. Apply the duty cycle

   The duty cycle can be intialised to 0 - it is updated every PWM period with the output of the controller.

   ```
   HRTIM_CompareCfgTypeDef pCompareCfg;
   pCompareCfg.CompareValue = 0; /* Will be updated */
   pCompareCfg.AutoDelayedMode = HRTIM_AUTODELAYEDMODE_REGULAR;
   if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1,
   HRTIM_TIMERINDEX_TIMER_C, HRTIM_COMPAREUNIT_1, &pCompareCfg) != HAL_OK)
       _Error_Handler();
   ```

2. Trigger the ADC conversion

   The maximum duty cycle is defined in Section 2.5.6 Compensator output and PWM duty cycle.

   ```
   pCompareCfg.CompareValue = DUTY_TICKS_MAX; /* Max duty cycle */
   if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1,
   HRTIM_TIMERINDEX_TIMER_C, HRTIM_COMPAREUNIT_2, &pCompareCfg) !=
   HAL_OK)
       _Error_Handler();
   ```

3. Trigger the ADC conversion

   In this example the ADC is triggered at the start of the PWM cycle. This means that the controller introduces a time lag of one sample period, resulting in a phase erosion of 18 degrees at the 0 dB crossing frequency. To improve the phase margin, the ADC can be triggered later, depending on how long the controller takes to execute.

   ```
   pCompareCfg.CompareValue = 60;
   if (HAL_HRTIM_WaveformCompareConfig(&hhrtim1,
   HRTIM_TIMERINDEX_TIMER_C, HRTIM_COMPAREUNIT_3, &pCompareCfg) !=
   HAL_OK)
       _Error_Handler();
   ```

   Comparator 3 is connected to the ADC trigger:

   ```
   HRTIM_ADCTriggerCfgTypeDef pADCTriggerCfg;
   pADCTriggerCfg.UpdateSource = HRTIM_ADCTRIGGERUPDATE_TIMER_C;
   pADCTriggerCfg.Trigger = HRTIM_ADCTRIGGEREVENT13_TIMERC_CMP3;
   if (HAL_HRTIM_ADCTriggerConfig(&hhrtim1, HRTIM_ADCTRIGGER_1, &pADCTriggerCfg) != HAL_OK)
       _Error_Handler();
   ```

We need complementary outputs to drive the high side and low side switches. A dead time of 75 x 8 PWM clock cycles (110 ns) is inserted between the falling edge of the secondary and the rising edge of the primary, and 300 x 8 cycles (441 ns) between the falling edge of the primary and the rising edge of the secondary. These values are chosen specifically for the Discovery kit.

```
HRTIM_TimerCfgTypeDef pTimerCfg;
HRTIM_DeadTimeCfgTypeDef pDeadTimeCfg;
/* Enable complementary outputs with dead time insertion */
pTimerCfg.DeadTimeInsertion = HRTIM_TIMDEADTIMEINSERTION_ENABLED;
if (HAL_HRTIM_WaveformTimerConfig(&hhrtim1,
HRTIM_TIMERINDEX_TIMER_C, &pTimerCfg) != HAL_OK)
    _Error_Handler();
pDeadTimeCfg.Prescaler = HRTIM_TIMDEADTIME_PRESCALERRATIO_MUL8;
pDeadTimeCfg.RisingValue = 75;
pDeadTimeCfg.RisingSign = HRTIM_TIMDEADTIME_RISINGSIGN_POSITIVE;
pDeadTimeCfg.RisingLock = HRTIM_TIMDEADTIME_RISINGLOCK_WRITE;
pDeadTimeCfg.RisingSignLock =
HRTIM_TIMDEADTIME_RISINGSIGNLOCK_WRITE;
pDeadTimeCfg.FallingValue = 300;
pDeadTimeCfg.FallingSign = HRTIM_TIMDEADTIME_FALLINGSIGN_POSITIVE;
pDeadTimeCfg.FallingLock = HRTIM_TIMDEADTIME_FALLINGLOCK_WRITE;
pDeadTimeCfg.FallingSignLock =
HRTIM_TIMDEADTIME_FALLINGSIGNLOCK_WRITE;
if (HAL_HRTIM_DeadTimeConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_C,
&pDeadTimeCfg) != HAL_OK)
    _Error_Handler();
```

The high side switch output is programmed to be set when the counter reaches the period count and returns to zero. It is reset when the counter reaches the duty cycle count (comparator 1) or the maximum duty cycle (comparator 2).

```
HRTIM_OutputCfgTypeDef pOutputCfg;
pOutputCfg.Polarity = HRTIM_OUTPUTPOLARITY_HIGH;
/* Set the PWM output with the period count */
pOutputCfg.SetSource = HRTIM_OUTPUTSET_TIMPER;
/* Reset the PWM output with comparator 1 or 2 */
pOutputCfg.ResetSource =
HRTIM_OUTPUTRESET_TIMCMP1|HRTIM_OUTPUTRESET_TIMCMP2;
pOutputCfg.IdleMode = HRTIM_OUTPUTIDLEMODE_NONE;
pOutputCfg.IdleLevel = HRTIM_OUTPUTIDLELEVEL_INACTIVE;
pOutputCfg.FaultLevel = HRTIM_OUTPUTFAULTLEVEL_INACTIVE;
pOutputCfg.ChopperModeEnable = HRTIM_OUTPUTCHOPPERMODE_DISABLED;
pOutputCfg.BurstModeEntryDelayed =
HRTIM_OUTPUTBURSTMODEENTRY_REGULAR;
/* Configure the high side output */
if (HAL_HRTIM_WaveformOutputConfig(&hhrtim1,
HRTIM_TIMERINDEX_TIMER_C, HRTIM_OUTPUT_TC1,
&pOutputCfg) != HAL_OK)
    _Error_Handler();
```

The low side switch output is the inverse of the high side, allowing for the dead time. This is selected automatically when dead time is enabled, so no set and reset source is necessary.

```
pOutputCfg.SetSource = HRTIM_OUTPUTSET_NONE;
pOutputCfg.ResetSource = HRTIM_OUTPUTRESET_NONE;
/* configure the low side output */
if (HAL_HRTIM_WaveformOutputConfig(&hhrtim1, HRTIM_TIMERINDEX_TIMER_C,
HRTIM_OUTPUT_TC2, &pOutputCfg) != HAL_OK)
    _Error_Handler();
```

2.9 Starting the controller

Before starting the FMAC, two more HAL driver parameters need to be defined, containing the size and location of the memory space reserved for the FMAC output. In this case, the size is 1 and the location is an 16-bit dummy variable. In fact this is not used since the interrupt handler performs the data transfer directly (see Section 2.5.6 Compensator output and PWM duty cycle).

```
uint16_t ExpectedCalculatedOutputSize = (uint16_t) 1;
int16_t Fmac_output;
```

We can now start the FMAC:

```
if (HAL_FMAC_FilterStart(&hfmac,&Fmac_output,
&ExpectedCalculatedOutputSize) != HAL_OK)
    Error_Handler();
```

Next we calibrate and start the ADC:

```
/* A pointer to the FMAC input data register for the DMA */
uint32_t *Fmac_Wdata;
Fmac_Wdata = (uint32_t *) FMAC_WDATA;
/* Do an auto-calibration */
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);
/* Start the ADC in DMA mode */
HAL_ADC_Start_DMA(&hadc1,Fmac_Wdata,1);
```

Finally, we start the HRTimer:

```
HAL_HRTIM_WaveformOutputStart(&hhrtim1, HRTIM_OUTPUT_TC1 |
HRTIM_OUTPUT_TC2);
HAL_HRTIM_WaveformCounterStart(&hhrtim1, HRTIM_TIMERID_TIMER_C);
```

2.10 Controller performance

Using an oscilloscope, we can check the performance of the buck converter against the specification.

In Figure 20. PWM waveforms, D0 and D1 trace, show the PWM output for the high side (primary) and low side (secondary) switches, when the buck converter is operating with a constant 100 mA load. The PWM period is 5 µs, corresponding to a 200 kHz switching frequency. The output voltage is 3.3 V, matching the target, as can be seen from the channel 1 trace.

In Figure 21. CPU load the D0 trace is a digital IO pin (GPO2) which is toggled while the CPU is idle ie. executing the while (1) loop. The GPO stops toggling when the CPU is executing the FMAC interrupt handler. We can see that the CPU takes the FMAC interrupt 460 ns after the start of the PWM cycle. The interrupt handler, including entry and exit, takes 223ns every 5us PWM cycle. Hence the controller occupies less than 5% of the CPU.

The ADC conversion trigger occurs at the start of the PWM cycle, meaning that the total time required to execute the controller, from sampling the output to updating the PWM cycle, is 685 ns (with the ADC clocked at 56.7 MHz). It would therefore be possible to move the sampling instant to around 700 ns before the start of the PWM cycle, thus increasing the phase margin by nearly 15.5 degrees.

Figure 22. Buck output ripple shows the buck converter output. The ripple due to the switching is around 12 mV pk-pk, within the specification.

Figure 23. Transient response shows the transient response, when the load current is doubled. The transient is approximately -20 mV and recovers in around 150 µs, without any overshoot.

Figure 20. PWM waveforms
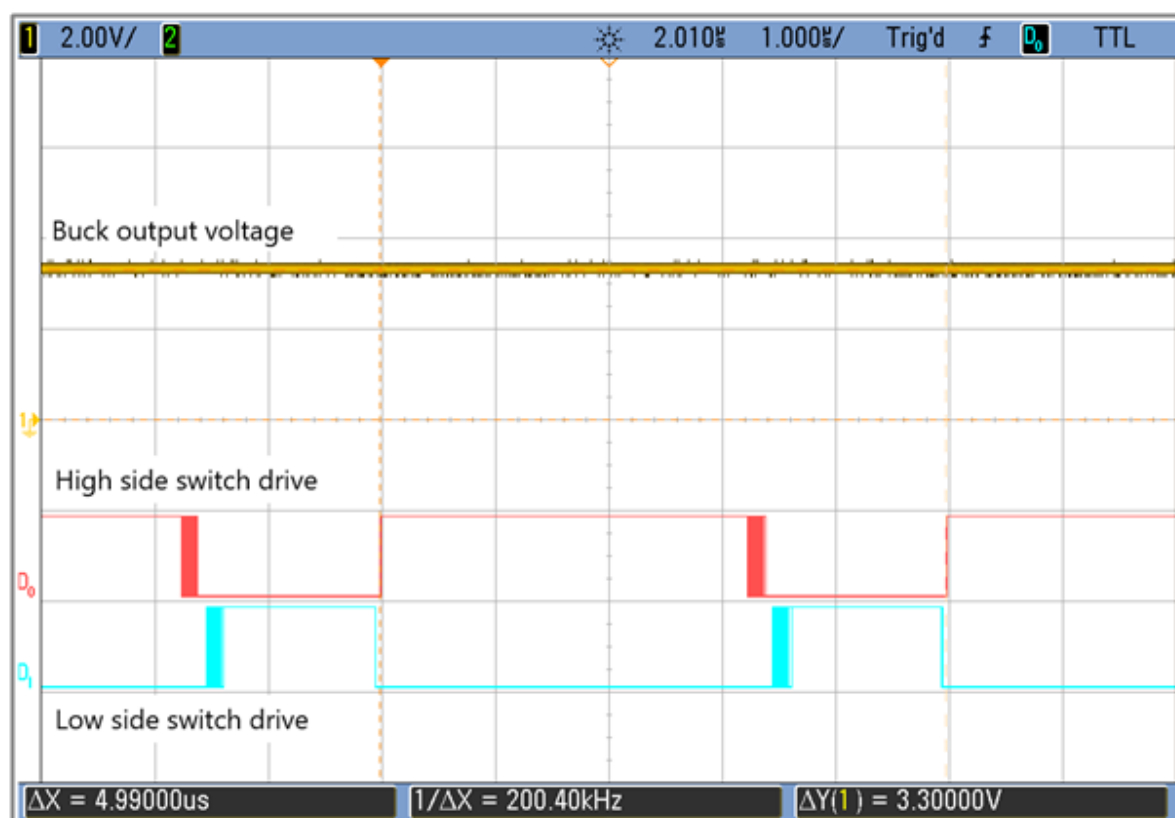
Figure 21. CPU load
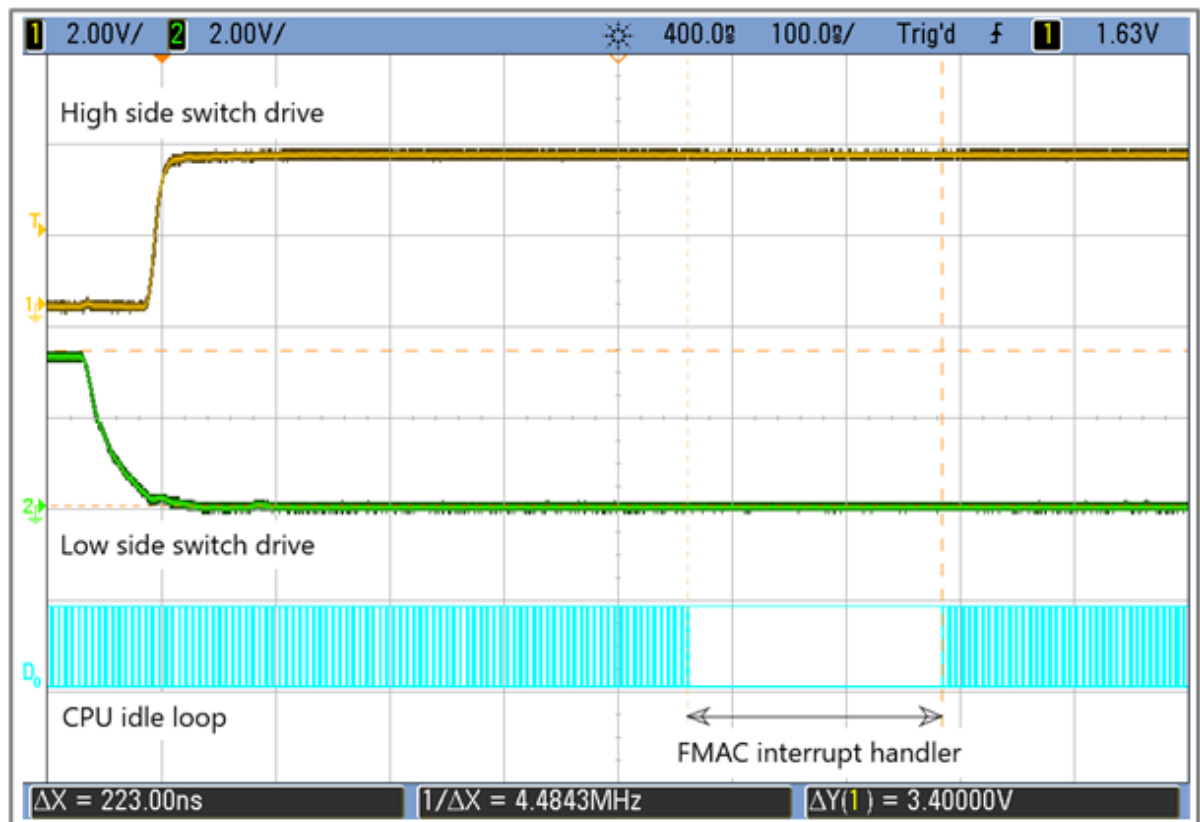
Figure 22. Buck output ripple
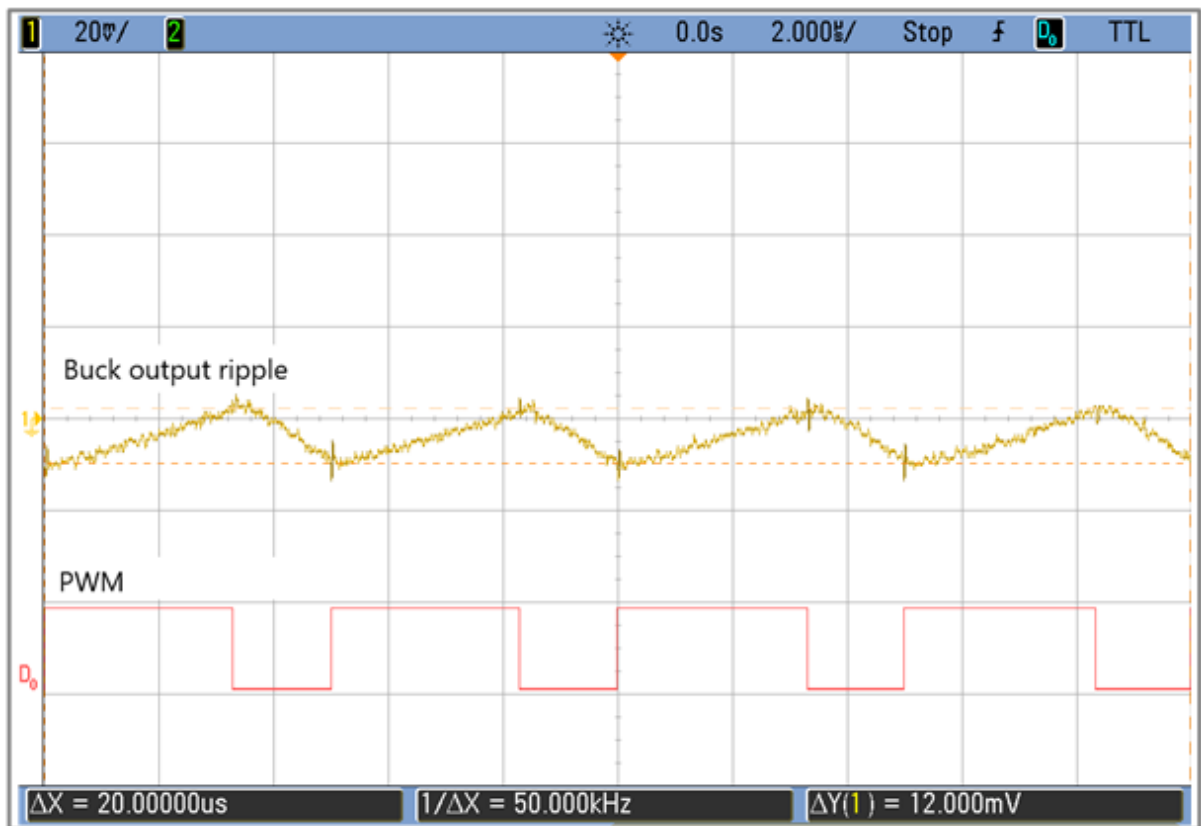
Figure 23. Transient response
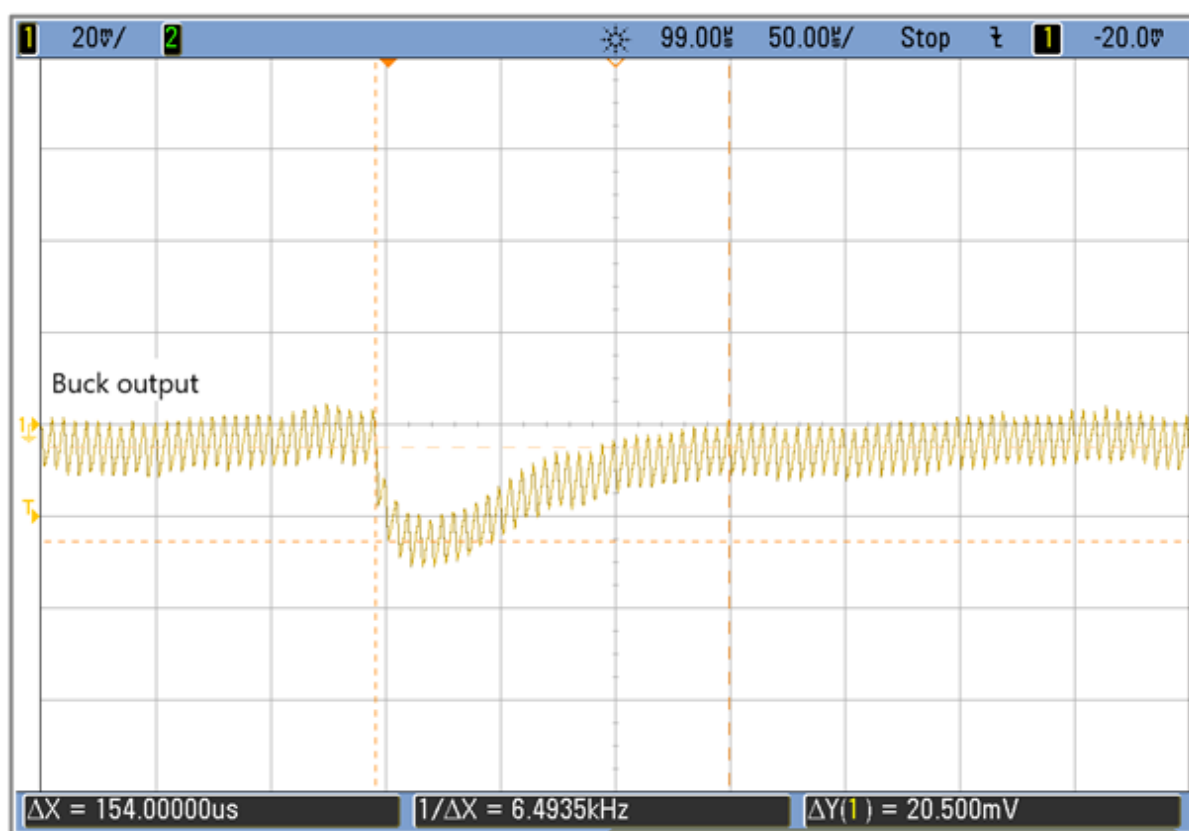
3 Conclusion

The two examples presented in this application note show how the FMAC can free up significant processing resources, by performing the repetitive number crunching associated with filtering tasks.

Filters may be designed with different tools, for different applications. But any filter that can be converted to FIR or IIR direct form 1 can potentially be handled by the FMAC, releasing precious CPU resources.

Clearly the gain is more significant in the case of a large filter operating on large buffers of data, such as the 51-tap adaptive FIR of example 1, since the processor is only required to intervene occasionally (in this case each frame to update the coefficients). The processor can be occupied more usefully performing the adaptation algorithm, for example.

In a control loop, such as the buck converter in example 2, where the processor has to intervene every sample, the gain is less significant. However in such cases real time constraints often mean that other tasks have to be pre-empted by the control loop. Reducing the length of this pre-emption even by a small amount can avoid the need for a faster, more powerful processor.

Revision history

Table 6. Document revision history

Date	Version	Changes
24-May-2019	1	Initial release.

Contents

Contents

List of tables

List of figures