Embedding it better...

µTasker Document

µTasker – Analog-Digital Converter (ADC) and Digital-Analog Converter (DAC) User's Guide

Copyright © 2023 M.J.Butcher Consulting

Table of Contents

1. Introduction

Many processors contain Analogue to Digital (ADC) converters, where typically 8 multiplexed channels are available. The characteristics and capabilities of the ADC can however vary greatly between different processors, which leads to an interface which can have a degree of processor specific options.

This document details the ADC user interface in the µTasker project and discusses the capabilities and interface of each supported processor. The interface may allow full control of all such features or may support a sub-set of them in order to achieve typical user requirements.

The ADC user interface is based on the general purpose µTasker `fnConfigureInterrupt()` interface. This is used in a generic matter in order to configure peripheral interrupts and perform peripheral configurations, although the interrupt operation is not always used. It is therefore worth discussing this interface in general terms to understand its goals and further methods of optimisation in particular cases:

1. The function `fnConfigureInterrupt()` is intended as a flexible and generic interface for use in configuring peripherals and the interrupt that is often used as interface.
2. By defining such an interface it makes compatibility between different hardware interfaces more easily realisable, although there are still usually different sets of options depending on the exact processor and its capabilities.
3. The interface is designed to be flexible but easy to use. This means that it should be simple to define various configurations without detailed knowledge of the processor's peripheral interface. It should thus be very efficient to start using such supported features by simply selecting the required configuration at a higher abstraction level. This means that enabling and starting using such an interface reliably requires very little work.
4. The interface tries to respect optimum use of power in the processor. Peripherals are only powered when they are required and will be disabled as soon as their functionality is no longer needed - for example, when a sequence of operations has completed. When the peripheral is reused the power management in the interface automatically reconfigures the necessary power control parts too.
5. Such a flexible interface does however mean that the peripheral code requires a high level of diversity. This leads to more code than is probably required to perform just a limited set of such functions, as well as longer configuration times (more instructions) than would be required to very specifically configure just used features. Therefore, in cases where not only a quick start to using such an interface is required but also a subsequent optimisation of a defined configuration is of importance, this can be efficiently achieved by using the interface as a reference – this is especially simple when working with the µTasker simulator by stepping through the configuration code to identify the low level register accesses (and values). These accesses can then be programmed in the same sequence in new dedicated user code to achieve the same results without passing through the reference `fnConfigureInterrupt()` code.

2. Selective ADC Triggering

The are often applications that are only interested in analogue samples when their value is within a certain range. Typical decisions are when the value is higher or lower than a pre-defined value to detect passing a high or low limit, or when it is inside or outside of range limits.

Software can of course perform this function by continuously sampling input values and checking for it with pre-defined limits but this means that the software may be doing unnecessary work while it could be concentrating on other functions or saving power consumption by setting the processor to a low power mode.

ADCs built into processors often support some form of automomous operation that continues without processor intervention and the following example shows use of the comparator feature built into the ADC in the Freescale Kinetis devices, which allows the processor to define limits or ranges and be woken by an interrupt when the ADC (which autonomously samples the value(s)) detects the input value as fulfilling the requirements.

The following shows how such an interrupt is configured (specifically for the Kinetis ADC but generally compatible with devices with this capability) for a low voltage threshold which will cause a defined interrupt to occur only when the input value is detected as being valid.

```
ADC_SETUP adc_setup;                          // interrupt configuration parameters
adc_setup.int_type = ADC_INTERRUPT;           // identifier when configuring
adc_setup.pga_gain = PGA_GAIN_OFF;            // PGA gain can be specified for certain inputs
adc_setup.int_handler = fnRangeInterrupt;     // handling function
adc_setup.int_priority = PRIORITY_ADC;        // ADC interrupt priority
adc_setup.int_adc_controller = 1;             // ADC controller 1
adc_setup.int_adc_offset = 0;                 // no offset
adc_setup.int_adc_mode = (ADC_CALIBRATE | ADC_SELECT_INPUTS_A | ADC_CLOCK_BUS_DIV_2 |
ADC_CLOCK_DIVIDE_8 | ADC_SAMPLE_ACTIVATE_LONG | ADC_CONFIGURE_ADC | ADC_REFERENCE_VREF |
ADC_CONFIGURE_CHANNEL | ADC_SINGLE_ENDED | ADC_SINGLE_SHOT_MODE | ADC_12_BIT_MODE |
ADC_SW_TRIGGERED);          // calibrate the ADC - single shot with interrupt on completion
adc_setup.int_adc_sample = (ADC_SAMPLE_LONG_PLUS_12 | ADC_SAMPLE_AVERAGING_32);
                                              // additional sampling clocks
adc_setup.int_adc_result = 0;                 // no result is requested
adc_setup.int_adc_bit = ADC_DM1_SINGLE;       // ADC DM1 single-ended
adc_setup.int_adc_bit_b = ADC_DISABLED;       // channel B only valid in HW triggered mode
adc_setup.int_adc_int_type = (ADC_LOW_LIMIT_INT);
                    // interrupt type (trigger only when lower than the defined level)
adc_setup.int_low_level_trigger = (unsigned short)(ADC_VOLT * 1.3);
                              // the low level trigger threshold represented as input voltage
adc_setup.int_handler = adc_ready_1;          // handling function
```

Although the mode is set to single shot mode, the internal operation is in fact for continuous sampling at the ADC level, which will then sease once the result is true – that is, it is a single-shot operation as perceived by the application.

A value of 1.3V is defined as low voltage threshold and detection is considered to be true when or a lower value is detected. On detection the user's interrupt callback will be executed (`fnRangeInterrupt()` in this example, although the interrupt code is not shown). `ADC_VOLT` is a system define equivalent to the value that 1VDC reads when the ADC is operated in 16 bit mode. When setting levels the µTasker ADC interface automatically converts to values corresponding to the present ADC operating mode. In the example above the mode is chosen to be 12 bits and so the threshold value actually set will automatically match the one required in that mode.

The internal thresholds in the Kinetis are in fact defined as „greater than or equal" and „less than". The µTasker ADC driver however automaticaly adjusts the limits to include or exclude the threshold in order to keep a consistent and symmetrical interface for the user.

The next example shows the changes required to modify the ADC interrupt to fire only when the input voltage is *higher* than a certain value:

```
adc_setup.int_adc_int_type = (ADC_HIGH_LIMIT_INT);
                  // interrupt type (trigger only when higher than the defined level)
adc_setup.int_high_level_trigger = (unsigned short)(ADC_VOLT * 2.6);
                            // the high level trigger threshold represented as input voltage
```

The next example shows how to configure for an interrupt when the sampled value is outside a range; that is, in case it falls lower than a lower threshold or exceeds a higher threshold:

```
adc_setup.int_adc_int_type = (ADC_LOW_LIMIT_INT  | ADC_HIGH_LIMIT_INT);  // interrupt type
                  (trigger only when higher or lower thresholds exceeded)
adc_setup.int_low_level_trigger = (unsigned short)(ADC_VOLT * 1.3);
                            // the low level trigger threshold represented as input voltage
adc_setup.int_high_level_trigger = (unsigned short)(ADC_VOLT * 2.6);
                            // the high level trigger threshold represented as input voltage
```

In this case the interrupt will trigger in case of either high or low values and the handling routine may want to read the sample to see which of the two cases it was.

The final example shows how the range can be inverted to cause interruption only when the sampled value is *within or equal to* the two limits, controlled by simply inverting the high and low settings:

```
adc_setup.int_adc_int_type = (ADC_LOW_LIMIT_INT  | ADC_HIGH_LIMIT_INT);  // interrupt type
                  (trigger only when range is valid)
adc_setup.int_high_level_trigger = (unsigned short)(ADC_VOLT * 1.3);
                            // the range start represented as input voltage
adc_setup.int_low_level_trigger = (unsigned short)(ADC_VOLT * 2.6);
                            // the range end represented as input voltage
```

This has show how use can be made of automatic level monitoring to allow the processor to only be interrupted when the input value matches a defined range.

3. Delay Line Application (ADC-DAC)

There are many instances where a stream of analogue measurement values are to be sampled and a stream of analogue values are to be generated. A delay line is a good general purpose representation of these two in operation and so is used here as illustration.

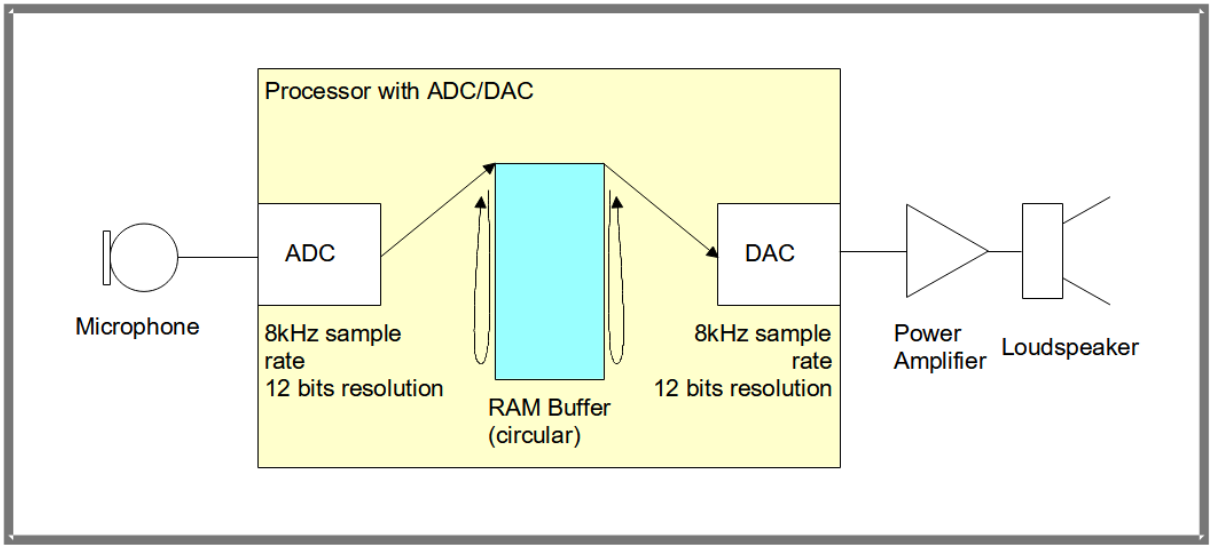


Figure 1: Example of a delay line configuration

Figure 1 shows a processor with ADC and DAC support implementing a delay line between a microphone input and a loudspeaker output. Assuming that each 12 bit sample value is stored in two bytes and the total RAM buffer size of 16k (8k samples) it will be possible to store 1s of input data in the RAM buffer before the circular buffer wraps around to the start again. If the DAC is synchronous to the ADC operation but converting one sample value ahead of the ADC it results in a 1s delay between the microphone input and the output in the loudspeaker.

Assuming that the processor used is capable of performing the copy of the ADC values (using an 8kHz time base) to the RAM buffer using DMA (Direct Memory Access) and also performing the copy from the RAM buffer to the DAC using DMA this delay line can operate without any processor (CPU) involvement. This represents a highly efficient solution which can even continue to operate when the CPU stops, such as when the debugger has hit a breakpoint. The following practical example is based on the Freescale Kinetis with inbuilt ADC/DAC and flexible DMA support. The ADC and DAC timing is controlled by its PDB (Programmable Delay Block) which can be extended if needed to control sampling of A and B channels of its multiple ADC and DACs.

The following program illustrates how such a configuration can be easily setup using the hardware interface.

First the RAM buffer is defined:

```
#define AD_DA_BUFFER_LENGTH        (8 * 1024)               // buffer for 1s at 8k bytes/s
static signed short sADC_buffer[AD_DA_BUFFER_LENGTH] = {0}; // 16 bit samples
```

Then the ADC is configured with suitable parameters:

```
ADC_SETUP adc_setup;                        // interrupt configuration parameters
adc_setup.int_type = ADC_INTERRUPT;         // identifier when configuring
adc_setup.pga_gain = PGA_GAIN_OFF;          // PGA gain can be specified for certain inputs
#if defined KWIKSTIK
adc_setup.int_adc_controller = 0;           // ADC controller 0
adc_setup.int_adc_bit = ADC_SE10_SINGLE;    // microphone on kwikstik board
#else
adc_setup.int_adc_controller = 1;           // ADC controller 1
adc_setup.int_adc_bit = ADC_DM1_SINGLE;     // potentiometer on K60 board
#endif
adc_setup.int_adc_bit_b = ADC_DISABLED;     // channel B only valid in HW triggered mode
adc_setup.dma_int_priority = 7;             // interrupt priority
adc_setup.dma_int_handler = half_buffer_interrupt;
adc_setup.ucDmaTrigger = 0;                 // use default trigger
adc_setup.ptrADC_Buffer = sADC_buffer;      // ADC sample buffer to be used
adc_setup.ulADC_buffer_length = sizeof(sADC_buffer); // physical length of the buffer
adc_setup.ucDmaChannel = 6;                 // DMA channel 6 used

// Hardware triggering (DMA to buffer with interrupt on half-buffer completion)
// - requires PDB set up afterwards
//
adc_setup.int_adc_mode = (ADC_CALIBRATE | ADC_HALF_BUFFER_DMA | ADC_SELECT_INPUTS_A |
       ADC_CLOCK_BUS_DIV_2 | ADC_CLOCK_DIVIDE_8 | ADC_SAMPLE_ACTIVATE_LONG |
       ADC_CONFIGURE_ADC | ADC_REFERENCE_VREF | ADC_CONFIGURE_CHANNEL |
       ADC_SINGLE_ENDED | ADC_SINGLE_SHOT_MODE | ADC_12_BIT_MODE | ADC_HW_TRIGGERED);
adc_setup.int_adc_sample = (ADC_SAMPLE_LONG_PLUS_2 | ADC_SAMPLE_AVERAGING_4);
                                            // additional sampling clocks
adc_setup.int_adc_result = 0;               // no result is requested
fnConfigureInterrupt((void *)&adc_setup); // configure ADC
```

Notice that this configuration is suitable for the Kwikstik module or the K60N512 tower board (depending on the define KWIKSTIK). The Kwikstik has ADC0 SE10 input connected to a microphone and the K60 tower board has ADC1 DM1 connected to a potentiometer. Other configurations are very simple since on the ADC controller and its input need to be specified.

The sample clock and mode (`ADC_12_BIT_MODE` 12 bits) are defined, along with the fact that HW triggering (`ADC_HW_TRIGGERED`) is to be used. Although not required for the basic configuration, an interrupt is defined for each half-buffer completion (`half_buffer_interrupt()`); this operation is discussed later.

Since `ADC_HALF_BUFFER_DMA` is defined (or `ADC_FULL_BUFFER_DMA`) the driver will configure the ADC operation using DMA to the defined buffer. The DMA channel used is channel 6 as specified by the user. Since there may be limits as to the use of DMA channels the user should ensure that the DMA channel to be used is available and doesn't conflict with other uses. The driver then performs all ADC, DMA and interrupt configurations as required by the setup.

`ADC_CALIBRATE` is used on the first configuration of the Kinetis ADC so that it can be appropriately calibrated for use by the driver routine.

In a similar manner the DAC is configured to use the same RAM buffer and perform conversion based on DMA operations:

```
DAC_SETUP dac_setup;
dac_setup.int_type = DAC_INTERRUPT;
dac_setup.int_handler = 0;        // no interrupt used
dac_setup.int_priority = 15;      // lowest priority (not used in this case)

// Configure the DAC to use VDDA as reference voltage in non-buffered mode (using DMA)
//
dac_setup.dac_mode = (DAC_CONFIGURE | DAC_REF_VDDA | DAC_NON_BUFFERED_MODE |
                      DAC_FULL_BUFFER_DMA | DAC_ENABLE);
#if defined KWIKSTIK
dac_setup.int_dac_controller = 1; // DAC 1
#else
dac_setup.int_dac_controller = 0; // DAC 0
#endif
dac_setup.ucDmaChannel = 7;       // use DMA channel 7
dac_setup.ucDmaTriggerSource = DMAMUX0_CHCFG_SOURCE_PDB;

// DAC transmit buffer to be used (use the ADC buffer to create a digital delay line)
//
dac_setup.ptrDAC_Buffer = (unsigned short *)sADC_buffer;
dac_setup.ulDAC_buffer_length = sizeof(sADC_buffer); // physical length of the buffer
fnConfigureInterrupt((void *)&dac_setup); // configure DAC
```

Notice again that the DAC used depends on the hardware (`KWIKSTIK`) and a different DMA controller is selected for the transfer. `DAC_FULL_BUFFER_DMA` is responsible for causing configuration based on DMA and in this case no interrupt is configured.

The same RAM buffer is specified as used by the ADC so that the delay line results. Of course a different buffer could be used by passing details of it instead.

The ADC and DAC have now been configured for DMA operation as required by the delay line but still need a time base for the operation to be able to start. The PDB is therefore configured as final step and the operation starts immediately.

```
PDB_SETUP pdb_setup;                  // interrupt configuration parameters
pdb_setup.int_type = PDB_INTERRUPT;
pdb_setup.int_priority = PRIORITY_PDB;
#if defined KWIKSTIK
// Periodic DMA and trigger ADC0 - channel A
//
pdb_setup.pdb_mode = (PDB_PERIODIC_DMA | PDB_TRIGGER_ADC0_A);
#else
// Periodic DMA and trigger ADC1 - channel A
//
pdb_setup.pdb_mode = (PDB_PERIODIC_DMA | PDB_TRIGGER_ADC1_A);
#endif
pdb_setup.prescaler = (PDB_PRESCALER_4 | PDB_MUL_1); // pre-scaler values of 1, 2, 4, 8,
16, 32, 64 and 128 are possible (with multipliers of 1, 10, 20 or 40)
pdb_setup.period = PDB_FREQUENCY(4, 1, 8000); // frequency of PDB cycle is 8kHz
pdb_setup.int_match = 0; // PDB interrupt/DMA at the start of the period so that it uses
the old ADC value
pdb_setup.ch0_delay_0 = pdb_setup.period; // ADC0 channel A trigger occurs at end of the
PDB period (when used)
pdb_setup.ch0_delay_1 = 0;
pdb_setup.ch1_delay_0 = pdb_setup.period; // ADC1 channel A trigger occurs at end of the
PDB period (when used)
pdb_setup.ch1_delay_1 = 0;
pdb_setup.pdb_trigger = PDB_TRIGGER_SW; // triggered by software (started immediately)
fnConfigureInterrupt((void *)&pdb_setup); // configure PDB and start operation
```

The PDB is one of several methods that could be used to trigger the ADCs and DMA on the Kinetis but represents a flexible method without requiring other timers on the device. By setting preferred PDB pre-scaler and multiplier values the macro `PDB_FREQUENCY()` is used to calculate the 8kHz sampling rate. The DAC conversion is then configured to take place immediately and the ADC sample to be made at the end of the period so that the delay is achieved (the DAC is using the previous buffer sample). `PDB_TRIGGER_SW` causes the PDB operation to start once its configuration has completed.

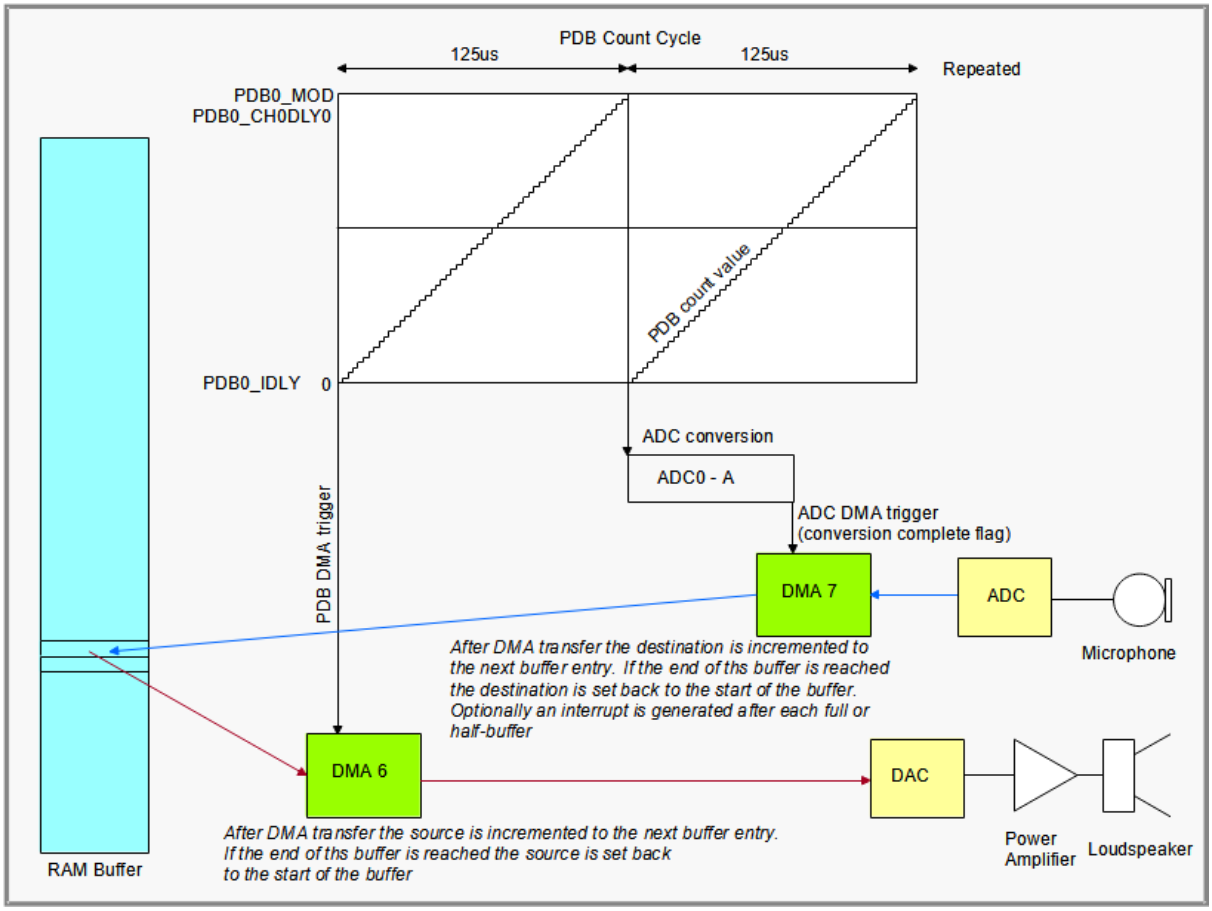


Figure 2: Details concerning PDB and DMA operation

Figure 2 shows more details about the operation. The PDB count cycle is shown whereby the 8kHz (125us) period is shown to be controlled by the PDB0_MOD value. After the count reaches this value it resets to the next cycle.

The value of PDB0_IDLY can control a periodic interrupt or, as in this case, a periodic DMA trigger which is used to trigger a DMA transfer from the RAM buffer to the DAC. Since the PDB0_IDLY value is set to 0 it occurs at the start of the PDB cycle.

The value of PDB0_CH0DLY0 is set to the same value as PDB0_MOD and this causes a hardware trigger to start conversion of ADC0 channel A. This means that the conversion starts at the end of the PDB cycle (after the DAC has used the 'previous' entry in the Ram buffer). The conversion time depends on the ADC (ADC clock, resolution and other) settings and the completion causes a DMA trigger which then copies the value from the ADC to the RAM buffer.

After each DMA transfer the source or destination is incremented (source for the DAC DMA and destination for the ADC DMA) until the end of the RAM buffer is reached (the defined length of the buffer). The source or destination pointer is automatically set back to the start of

the buffer when the end is reached so that operation can continue without and processor intervention.

Optionally the DMA controller can generate an interrupt when the wrap-around occurs as well as optionally when the first half of the buffer has been completed. Although this interrupt is not needed by the delay line's basic operation it does prove to be very useful in many instances.

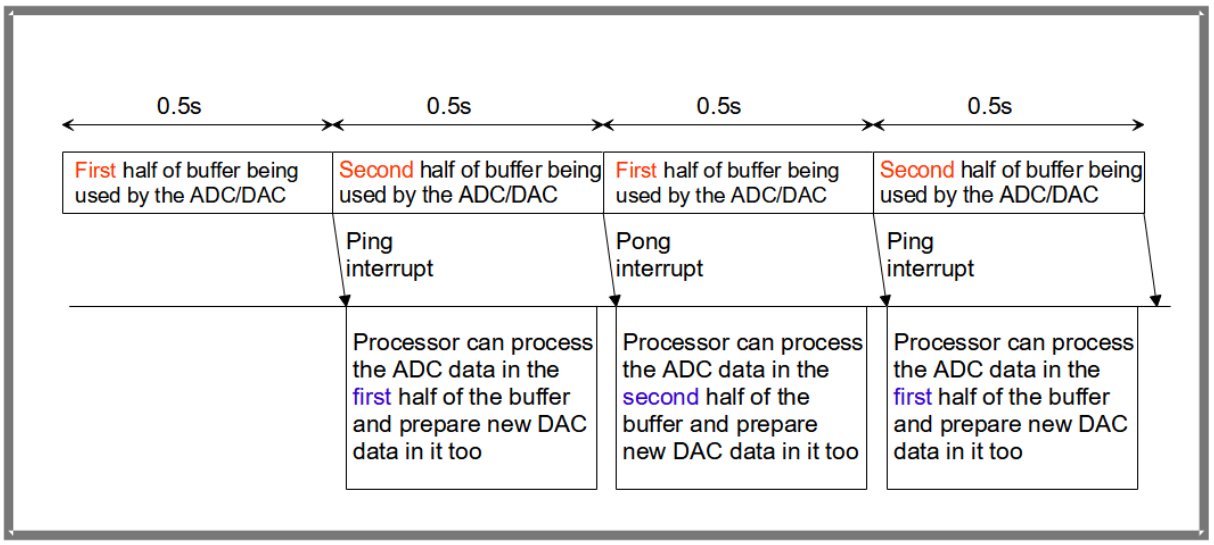


Figure 3: Ping-pong half-buffer interrupts and processing

Figure 3 shows the half-buffer interrupts in operation. These are names ping and pong interrupts since the operation is often referred to as ping-pong operation. The thing that Is seen is that when the interrupt arrives (pong or pong after each half-buffer completion) the half-buffer's content is stable for another half-buffer period (0.5 seconds in the example case). This means that the processor can use the data and also modify the data during this period without any risk of it being used by the DMA (overwritten by new ADC data or converted to a now analogue value by the DAC). Although the 0.5s delay may not be suitable for many applications performing such processing (these may work with delay of 10ms, for example instead) the principle is the same.

Since the half-buffer interrupt was enabled in our example the following code shows how it could be used in the delay line case to amplify the signal in the delay path.

```
static void half_buffer_interrupt(void)
{
    static unsigned char ucPingPong = 0;
    fnInterruptMessage(OWN_TASK, (unsigned char)(ADC_TRIGGER_1 + ucPingPong));
    ucPingPong ^= 1;
}
```

The interrupt routine can perform processing but it is often more suitable to handle it in a task since it is usually not critical that it be performed immediately. Looking at the timing in figure 3 there is a delay from the interrupt and it is necessary to complete the operation before the next time that the buffer is used. With this requirement in mind, the worst case delay for the completion of the processing (including start delay) needs to be respected. In the case of 0.5s and low overhead processing this can be usually guaranteed without any difficulty.

The interrupt routine simply sends an event indicating whether the ping or the pong buffer is ready and it can then be handled accordingly. The following shows handling two events to amplify the AC content of the signal by 16 (assuming that there is a DC offset 0x0800 with no microphone signal component).

```
if ((ADC_TRIGGER_1 == ucInputMessage[MSG_INTERRUPT_EVENT]) ||
    (ADC_TRIGGER_2 == ucInputMessage[MSG_INTERRUPT_EVENT])) {
    int i;
    signed short *ptrSample;
    if ((ADC_TRIGGER_2 == ucInputMessage[MSG_INTERRUPT_EVENT])) {
        ptrSample = &sADC_buffer[AD_DA_BUFFER_LENGTH/2];
                                    // set sample pointer to second half of the buffer
    }
    else {
        ptrSample = &sADC_buffer[0]; // set sample pointer to first half of the buffer
    }
    for (i = 0; i < (AD_DA_BUFFER_LENGTH/2); i++) {
        *ptrSample -= 0x0800;          // remove DC bias
        if (*ptrSample > (2047/16)) { // limit positive input value if needed
            *ptrSample = (2047/16);   // maximum positive input value
        }
        else if (*ptrSample < (-2048/16)) { // limit negative input value if needed
            *ptrSample = (-2048/16);  // maximum negative input value
        }
        *ptrSample *= 16;              // amplify the AC value
        *ptrSample++ += 0x0800;        // add the DC bias again
    }
}
```

The delay line shows a simple case but still contains the complete framework as required by many diverse applications.

For example

- a filter algorithm could be applied to filter the input before it is passed to the output.
- The ADC input could be saved as a WAV date to an SD card for playback at a later point in time.
- The DAC output could be prepared from a WAV file on an SD card for playback

Binary files of the delay line example are available for the K40 Kwikstik and TWR-K60N512 boards at http://www.utasker.com/SW_Demos.html# KINETIS_APP

4. Repetitive Signal Generation using DAC and DMA

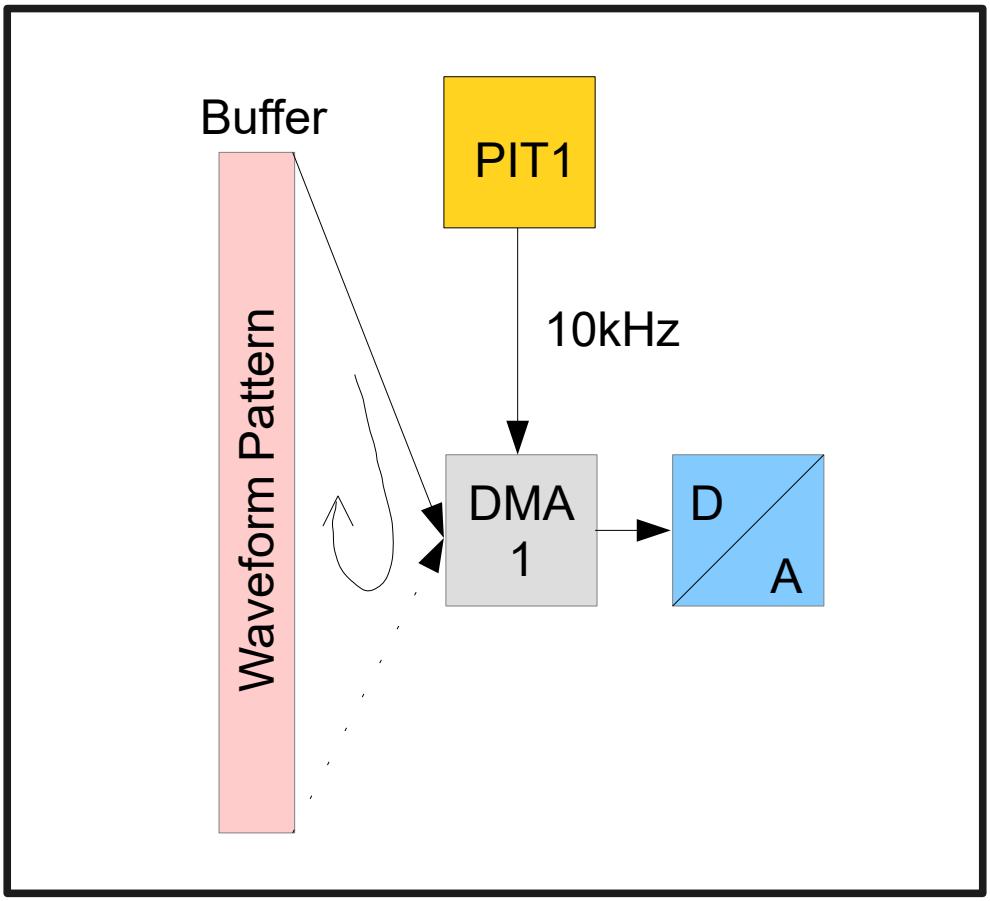
Generating a repetitive signal at a DAC output is very efficient when DMA can be used. The following reference can be enabled with the define `TEST_DMA_DAC` in `ADC_Timers.h` and uses a Kinetis PIT timer as DMA trigger to copy the content of a buffer to the output. The buffer content determines the signal that is generated and the speed of the timer the rate of the output and is related to frequency of the repetitive signal.

```
#define LENGTH_OF_TEST_BUFFER    128
unsigned short *ptrTestBuffer = 0;
DAC_SETUP dac_setup;
PIT_SETUP pit_setup;             // interrupt configuration parameters
pit_setup.int_type = PIT_INTERRUPT;
pit_setup.int_handler = 0;       // no interrupt since the PIT will be used for triggering DMA
pit_setup.int_priority = PIT1_INTERRUPT_PRIORITY;
pit_setup.count_delay = PIT_US_DELAY(100);         // 10kHz
pit_setup.mode = (PIT_PERIODIC);                   // periodic DMA trigger
pit_setup.ucPIT = 1;                               // use PIT1
fnConfigureInterrupt((void *)&pit_setup);         // configure PIT

ptrTestBuffer = uMallocAlign((LENGTH_OF_TEST_BUFFER * sizeof(unsigned short)),
                             (LENGTH_OF_TEST_BUFFER * sizeof(unsigned short)));
                                // obtain buffer memory from heap (aligned for KL devices modulo mode)
// Prepare a ramping signal suitable for 12 bit DAC output
//
for (i = 0; i < (LENGTH_OF_TEST_BUFFER/2); i++) {
    ptrTestBuffer[i] = (i * (LENGTH_OF_TEST_BUFFER/2)); // ramp up
}
ptrTestBuffer[i++] = 0x0fff;                       // max. 12 bit value
for (; i < LENGTH_OF_TEST_BUFFER; i++) {           // ramp down
    ptrTestBuffer[i] = ((LENGTH_OF_TEST_BUFFER - i) * (LENGTH_OF_TEST_BUFFER/2));
}

dac_setup.int_type = DAC_INTERRUPT;
dac_setup.int_dac_controller = 0;                  // DAC 0
dac_setup.int_handler = 0;                         // no interrupt used
dac_setup.int_priority = 15;                       // lowest priority (not used in this case)
dac_setup.dac_mode = (DAC_CONFIGURE | DAC_REF_VDDA | DAC_NON_BUFFERED_MODE | DAC_FULL_BUFFER_DMA |
DAC_ENABLE);  // configure the DAC to use VDDA as reference voltage in non-buffered mode (using DMA)
        #if defined KINETIS_KL
//dac_setup.dac_mode |= DAC_FULL_BUFFER_DMA_AUTO_REPEAT; // optional interrupt controlled DMA reload
        #endif
dac_setup.ptrDAC_Buffer = (unsigned short *)ptrTestBuffer;
dac_setup.ulDAC_buffer_length = (LENGTH_OF_TEST_BUFFER * sizeof(unsigned short));
dac_setup.ucDmaChannel = 1;                        // DMA channel 1 used
dac_setup.ucDmaTriggerSource = DMAMUX0_DMA0_CHCFG_SOURCE_PIT1; // PIT1 triggers the channel mux
dac_setup.dac_mode |= DAC_HW_TRIGGER_MODE;         // use HW trigger mode rather than SW triggered mode
fnConfigureInterrupt((void *)&dac_setup);          // configure DAC
```

This code configures PIT1 as free-running 10kHz timer. This has the capability to trigger DMA channel 1 (*generally Kinetis devices allow PIT0 to trigger DMA channel 0, PIT1 to trigger DMA channel 1, PIT2 to trigger DMA channel 2, etc.*).

A waveform pattern is programmed in a RAM buffer, which in this case is a ramp up from 0x00 to 0xfff, followed by a ramp down from 0xfff to 0x00. The buffer is created in aligned heap so that Kinetis KL devices can make use of their DMA modulo mode, whereby a buffer of a power of two length on a power of two address boundary allows the DMA controller to repeat the buffer. *The more powerful DMA controller in the Kinetis K parts doesn't need to respect these requirements*.

Finally the DAC is configured with DMA feeding, triggered by the PIT. The result is that the buffer content is continuously fed to the DAC output at the rate defined by the PIT.



Kinetis KL devices only:

When power of two length operation is used, the transfer count is set to 0xffff0, which is the maximum that is accepted. After this total number of transfers (a little more than 1 million) has completed the operation stops. In order to continue operation the user can regularly write the transfer count back to the maximum value using `DMA_DSR_BCRn = 0xffff0;` where n is the DMA channel being used, before the termination takes place.

In some situations it is not practical to use a power of two length buffer. This may be due to the fact that the buffer length is defined based on different requirements or it is not possible to locate it at the required aligned memory location.

In such a situation the ADC and DAC DMA drivers can be instructed to automatically handle the DMA buffer complete interrupt and re-configure the next buffer operation. This can also be used together with a user interrupt call-back, which is called after the next DMA buffer operation has been set up.

```
// Automated DMA restart when not using modulo repetitions
//
dac_setup.dac_mode |= DAC_FULL_BUFFER_DMA_AUTO_REPEAT;      // DAC
```

The disadvantage of the interrupt driven continuous DMA circular buffer operation is that there is an interrupt handled each time the buffer length is reached. If the rate of operation is very high there is a risk of a delayed interrupt not allowing the DMA operation to be reconfigured by the time that the next sample is ready.

5. Further Discussions

Note that there may be further discussions concerning the ADC usage at the µTasker forum. The following are examples from the Coldfire board:

http://www.utasker.com/forum/index.php?topic=280.0
http://www.utasker.com/forum/index.php?topic=437.0

6. Conclusion

The µTasker ADC interface has been presented, including threshold limited triggering.

A delay line example has shown how DMA can be used to efficiently transfer ADC data to a buffer and convert analogue output from the same buffer, resulting in a delay. The same framework can be used for many applications that require a data buffer and ping-pong buffer processing to processor or prepare data without interruptions in the overall data flow.

The µTasker driver interface allows flexibly configuration for the Kinetis based on DMA and PDB (Programmable Delay Block) via a simple user interface. This allows the user to concentrate on the application in hand rather than details of the low level hardware.

Modifications:

V0.01 21.04.2009: Initial draft – work in progress. Not officially released.

V0.02 17.07.2010: Added introduction to use of the `fnConfigureInterrupt()` interface and Luminary ADC discussion.

V0.03 24.12.2013 Added ADC/DAC delay line based on Kinetis using DMA and PDB.

V0.04 14.05.2015 Added threshold triggering based on Kinetis.

V0.05 23.12.2015 Added DAC/DMA repetitive signal generation and Kinetis KL delay-line discussion in appendix.

V0.06 27.12.2015 Added PMW output discussion in appendix.

V0.07 4.1.2016 Added note about `ADC_HALF_BUFFER_DMA` use by Kinetis KL parts.

V0.08 15.1.2017 Added details about Kinetis ADC input multiplexing.

V0.09 4.3.2017 Added Kinetis TPM HW triggering reference for ADC inputs A and B.

V0.10 11.7.2023 Added i.MX RT TEMPMON interface

Appendix A – Hardware Dependencies

a) Luminary Micro ADC

The Luminary Micro parts can be divided into three categories depending on the device class. The original devices had 8 dedicated analogue inputs and a single ADC module.

The Dust-Devil class added an analogue isolation so that the AD inputs could be shared with a GPIO – the isolator being controlled by the GPIOAMSEL_X register.

The Tempest parts use the same analogue isolator together with multiplexed GPIO peripheral functions and added a second ADC controller and up to 16 ADC inputs. A single ADC controller can sample and of the possible 16 inputs but a maximum of 8 samples are possible. By using both of the (independent) ADC controllers all 16 ADC inputs can effectively be sampled in two parallel sequences. The Tempest parts add an additional, optional reference input (VREFA) - optimal performance is achieved with values in the range 2.4V to 3.0V, whereby some internal components start to saturate above 3V – to be avoided.

Beware that the first Tempest parts had a bug in the ADC, meaning that ADC averaging of differential signals should be avoided.

b) Kinetis KL Delay Line

The delay line reference in the main text uses the PDB (Programmable Delay Block) as is available in the Kinetis K parts. Since the Kinetis KL family do not implement PDM the following discusses equivalent operation using a PIT (Programmable Interrupt Timer) instead and also gives some details about the operation of the DMA circular buffer which needs to be considered due to differences in the DMA capabilities.

The following represent the major differences between the KL and K parts with respect to the operation in question:

- *DMA in the KL has less flexibility in its circular buffer operation; such buffers need to be modulo 2 in size (2, 4, 8, 16...1024 etc.) and also be located at equivalent modulo 2 address boundaries*
- *KL family has no PDB for triggering ADC conversion*
- *DMA in the KL family doesn't support half-buffer interrupt*

These differences lead to slight adjustments to the Delay Line configuration as detailed in the following sections.

ADC Configuration Differences:

```
adc_setup.int_adc_mode = (ADC_CALIBRATE | ADC_FULL_BUFFER_DMA | ADC_SELECT_INPUTS_A |
ADC_CLOCK_BUS_DIV_2 | ADC_CLOCK_DIVIDE_8 | ADC_SAMPLE_ACTIVATE_LONG | ADC_CONFIGURE_ADC |
ADC_REFERENCE_VREF | ADC_CONFIGURE_CHANNEL | ADC_SINGLE_ENDED | ADC_SINGLE_SHOT_MODE | ADC_12_BIT_MODE
| ADC_HW_TRIGGERED);                      // hardware triggering example (DMA to buffer)
ptrADC_buffer = uMallocAlign((AD_DA_BUFFER_LENGTH * sizeof(unsigned short)),
                (AD_DA_BUFFER_LENGTH * sizeof(unsigned short))); // create an aligned buffer on heap
adc_setup.ptrADC_Buffer = ptrADC_buffer;
adc_setup.ulADC_buffer_length = (AD_DA_BUFFER_LENGTH * sizeof(unsigned short));
                                        // physical length of the buffer
adc_setup.ucDmaChannel = 1;             // DMA channel 1 used
adc_setup.int_handler = 0;              // no interrupt because free-running circular buffer is used
```

In order for the DMA to continuously save ADC samples to the circular buffer, the buffer must be appropriately aligned in memory and have a modulo 2 length. For example, `AD_DA_BUFFER_LENGTH` of 256 fulfils this requirement and taking its buffer from heap at a 256 byte aligned start address is achieved by making use of `uMallocAlign()` (*although other techniques could also be used to achieve the same*).

DAC Configuration Differences:

```
dac_setup.ucDmaChannel = 0;                          // DMA channel 0 used (highest priority)
dac_setup.ucDmaTriggerSource = DMAMUX_CHCFG_SOURCE_ADC0;
                                                     // trigger DMA to DAC when ADC0 sample completes
dac_setup.dac_mode |= DAC_HW_TRIGGER_MODE;           // use HW trigger mode rather than SW triggered mode
```

Since the DAC uses the same buffer as the ADC it is already compatible with the DMA operation to copy the data from memory to the DAC.

The trigger source cannot use PDB and so specifies instead that the DMA DAC copy is triggered by ADC0 conversion completion.

PIT Configuration:

```
PIT_SETUP pit_setup;                          // interrupt configuration parameters
pit_setup.int_type = PIT_INTERRUPT;
pit_setup.int_handler = 0;                    // no interrupt used since the PIT triggers ADC/DAC only
pit_setup.int_priority = PIT0_INTERRUPT_PRIORITY;
pit_setup.count_delay = PIT_US_DELAY(125);    // 8kHz rate
pit_setup.ucPIT = 0;           // use PIT0 since it is the only one that can trigger DAC conversions
pit_setup.mode = (PIT_PERIODIC | PIT_RETRIGGER | PIT_TRIGGER_ADC0_A);
                               // periodically trigger ADC0 channel A (uses retrigger in case the PIT
                                  was running previously)
fnConfigureInterrupt((void *)&pit_setup);     // configure PIT0
```

This shows PIT0 being configured to free-run at 8kHz and trigger HW conversion at ADC0 each time it reloads. *This is used as HW trigger since there is no PDB in the KL parts*.

Since the ADC0 conversion completion is used as a trigger for both involved DMA channels the order or transfer execution is determined by the fixed priority scheme of the DMA, whereby channel 0 has priority over channel 1. This means that the original value in the buffer is first copied to the DAC output and then the new ADC sample is copied into the same location in the buffer, with the result that there is the desired single buffer delay. *In case the DMA channels were to be swapped there would be no delay since the ADC sample would first be copied to the buffer and then the same sample copied by the second DMA channel to the DAC*.

When power of two length operation is used the transfer count is set to 0xffff0, which is the maximum that is accepted. After this total number of transfers (a little more than 1 million) has completed the operation stops. In order to continue operation the user can regularly write the transfer count back to the maximum value using `DMA_DSR_BCRn = 0xffff0;` where n is the DMA channel being used, before the termination takes place.

In some situations it is not practical to use a power of two length buffer. This may be due to the fact that the buffer length is defined based on different requirements or it is not possible to locate it at the required aligned memory location.

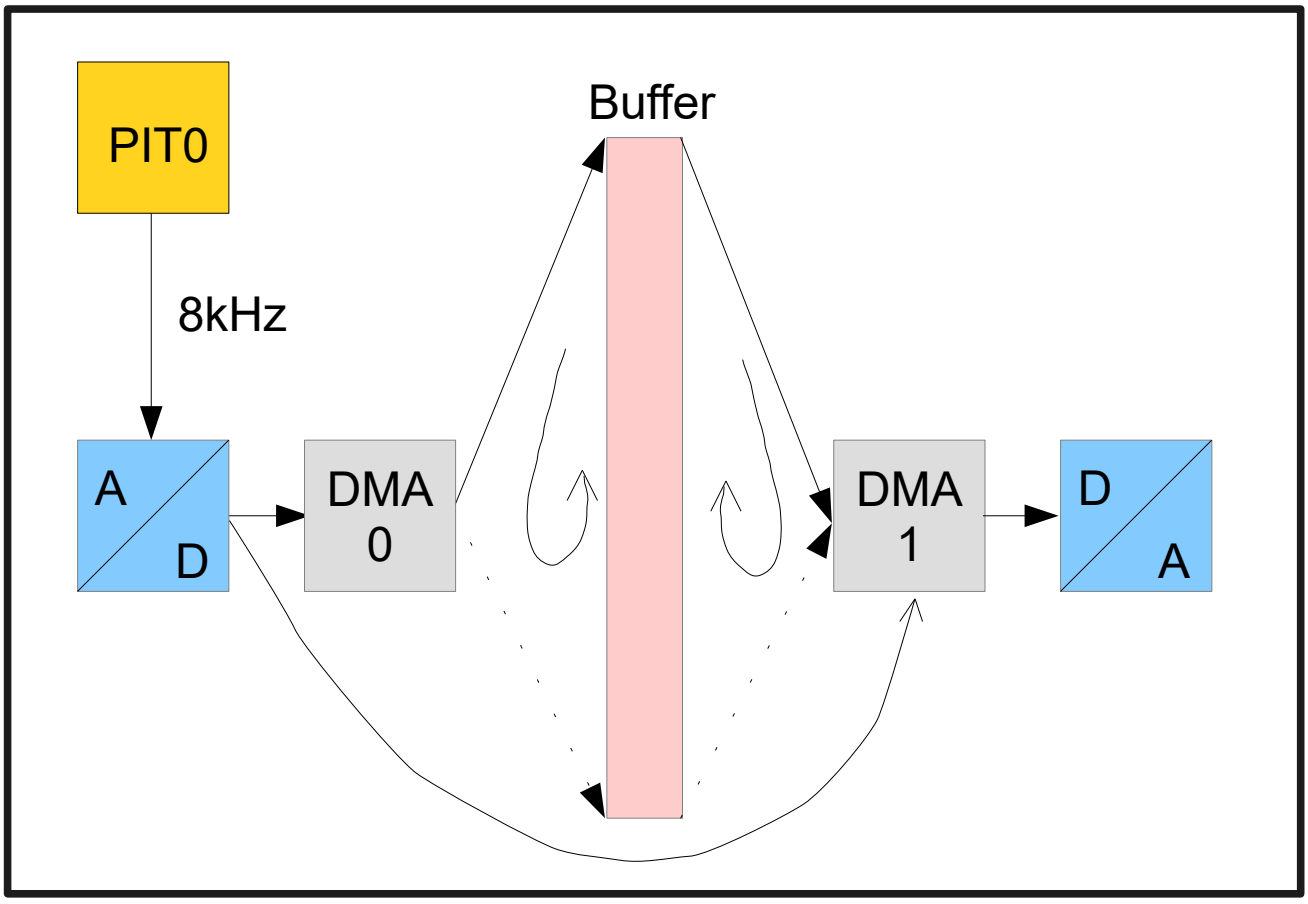
In such a situation the ADC and DAC DMA drivers can be instructed to automatically handle the DMA buffer complete interrupt and re-configure the next buffer operation. This can also be used together with a user interrupt call-back, which is called after the next DMA buffer operation has been set up.

```
// Automated DMA restart when not using modulo repetitions
//
adc_setup.int_adc_mode |= ADC_FULL_BUFFER_DMA_AUTO_REPEAT; // ADC
dac_setup.dac_mode |= DAC_FULL_BUFFER_DMA_AUTO_REPEAT;     // DAC
```

The disadvantage of the interrupt driven continuous DMA circular buffer operation is that there is an interrupt handled each time the buffer length is reached. If the rate of operation is very high there is a risk of a delayed interrupt not allowing the DMA operation to be reconfigured by the time that the next sample is ready.

When user interrupts are used, a further option to emulate a half-buffer mode of operation exists, which automatically divides the operation into two DMA copy processes. This means that the DMA driver handles two interrupt for each cycle and so the two user interrupts can be called back and allow Kinetis KL application layer code to be compatible with Kinetis K code that requires the two interrupts to process ping-pong buffers (to allow processing of buffer content, saving it to a storage medium or actively passing it on to other interfaces).

```
// Automated half-buffer DMA restart when not using modulo repetitions
// (usually used in conjunction with half-buffer user interrupt call-backs)
//
adc_setup.int_adc_mode |= (ADC_FULL_BUFFER_DMA_AUTO_REPEAT | ADC_HALF_BUFFER_DMA); // ADC
dac_setup.dac_mode |= (DAC_FULL_BUFFER_DMA_AUTO_REPEAT | DAC_HALF_BUFFER_DMA);     // DAC
```



ADC/DAC operation controlled by the PIT time-base

c) Kinetis Delay Line with PWM Output

Not all Kinetis parts include a DAC which can be used to output the delayed analogue signal with. PWM (Pulse-Width Modulation) is however another method that can be used to generate an analogue signal when it is filtered appropriately.

For this reason the Kinetis PWM module can also be used as output by configuring it, rather than the DAC, as shown in the alternative code below:

```
PWM_INTERRUPT_SETUP pwm_setup;
pwm_setup.int_type = PWM_INTERRUPT;
pwm_setup.pwm_mode = (PWM_SYS_CLK | PWM_PRESCALER_1 | PWM_FULL_BUFFER_DMA);
                                        // clock PWM timer from the system clock with /16 pre-scaler
#if defined KINETIS_KL
pwm_setup.pwm_mode |= PWM_FULL_BUFFER_DMA_AUTO_REPEAT;
                          // automated DMA (using interrupt) restart when not using modulo repetitions
#endif
pwm_setup.pwm_frequency = PWM_TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(2000), 2);
                                                                    // generate 2000Hz on PWM output
pwm_setup.pwm_value   = _PWM_PERCENT(50, pwm_setup.pwm_frequency);  // 50% PWM (high/low)
pwm_setup.pwm_reference = (_TIMER_2 | 0);                           // timer module 2, channel 0
pwm_setup.int_priority = 0;
pwm_setup.int_handler = 0;
pwm_setup.ucDmaChannel = 0;                                 // DMA channel 0 used (highest priority)
pwm_setup.ucDmaTriggerSource = DMAMUX_CHCFG_SOURCE_ADC0;    // DMA triggered by ADC0 conversion
pwm_setup.ptrPWM_Buffer = (unsigned short *)ptrADC_buffer;
                          // PWM buffer to be used (use the ADC buffer to create a digital delay line)
pwm_setup.ulPWM_buffer_length = (AD_DA_BUFFER_LENGTH * sizeof(unsigned short));
                                                            // physical length of the buffer
fnConfigureInterrupt((void *)&pwm_setup);                   // configure PWM
```

The PWM mode option `PWM_FULL_BUFFER_DMA` informs the PWM driver that it should configure the DMA operation according to the buffer, length, DMA channel and trigger parameters that are passed. In this case a transfer from the buffer is performed to the PWM mark-space ratio control register each time an ADC0 conversion completes.

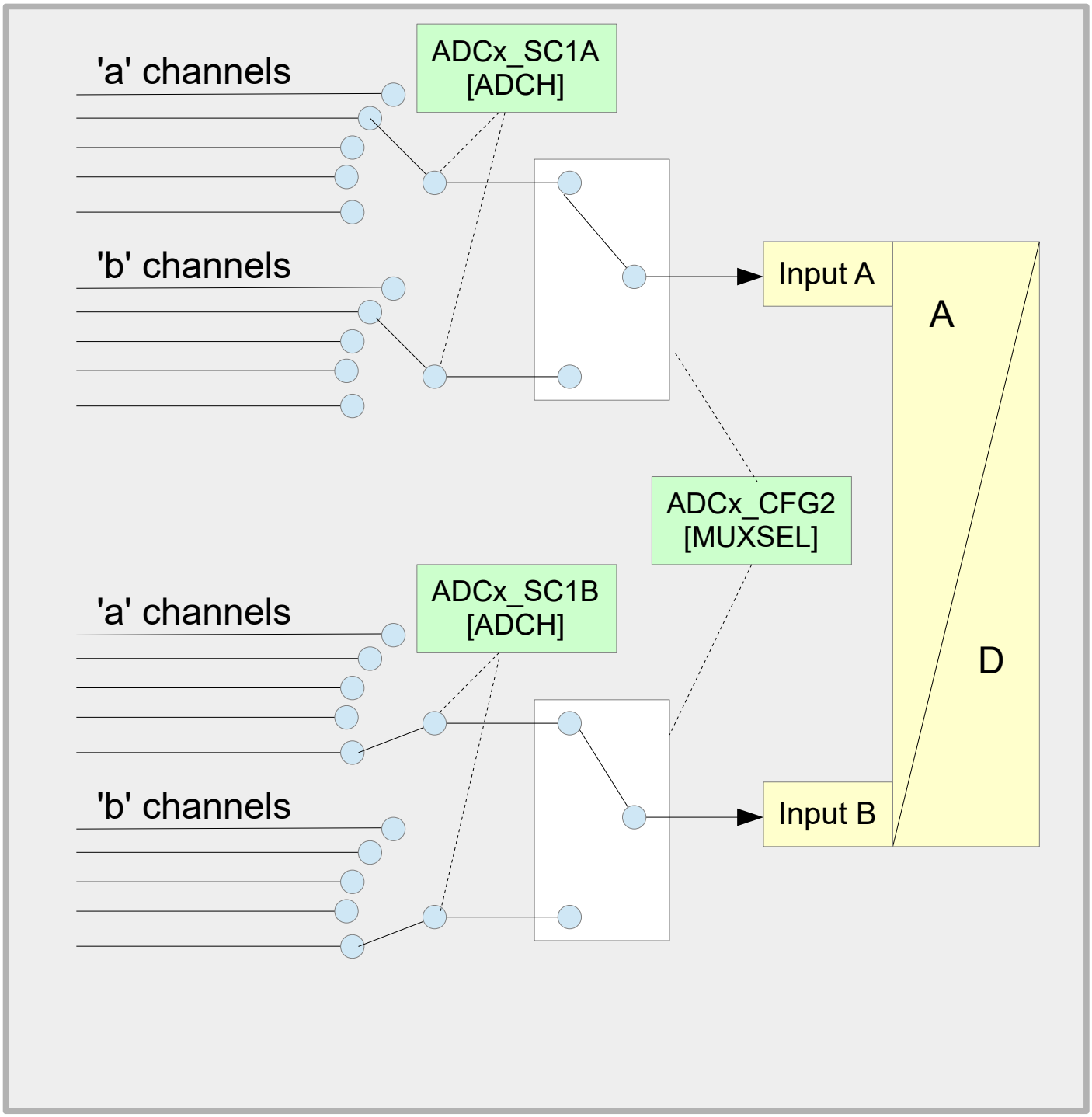
The flag `PWM_FULL_BUFFER_DMA_AUTO_REPEAT` can be used together with the Kinetis KL parts if the module buffer operation is not used so that the driver interrupt automatically reconfigures the next buffer transfer after each one completes.

In order for PWM to be used the values in the ADC input buffer must be appropriate for the PWM output control (when not processed by software, thus allowing pure DMA based operation). In this reference the ADC input range is considered to be 12 bits (0x0000..0x07fff) and so the PWM mark-space ratio register should expect a similar range. By choosing the output frequency value (`pwm_setup.pwm_frequency`) to be approximately 0x07ff, although not smaller, already gives a good result as long as the PWM frequency itself is an order higher that the sample frequency.

What if possible and whether it allows the desired operation to be achieved with the expected quality will depend on the frequencies involved and available clocking options. The 2000Hz used in the reference code was suitable for a certain processor and clocking frequency but would not give good results unless the sampling frequency (8kHz in the references) were not also reduced to compensate. Even so, it demonstrates well the basic operation involved.

d) Kinetis ADC Input Multiplexing

There is often confusion about Kinetis ADC input multiplexing and for this reason the following illustration intends to simplify understanding of the ADC input path selection.



The first important detail is that the **Input B** of the ADC is only used in hardware triggered mode, which means that the ADC conversions are being triggered by HW timers (PIT or PDB); in simple software triggered or free-running mode only **Input A** is used.

The ADC input connected is defined by two registers:
ADC0_SC1A [ADCH] determines which of the various inputs (channel number) is selected, noting that only one of the possible inputs is actually connected to the ADC at any time;
ADC0_CFG2 [MUXSEL] selects optional input channel multiplexing in parts where there is an option available on the particular channel.

For example, a Kinetis part may have three inputs of interest to an application called ADC0_DP0, ADC0_SE22 and the internal temperature circuit. These may be selected in the µTasker interface by the defines `ADC_DP0_SINGLE`, `ADC_SE22_SINGLE` and `ADC_TEMP_SENSOR` respectively. These selections cause the ADCx_SC1A switch to be set accordingly.

Some Kinetis parts however have inputs called, for example, ADC1_SE4**a** and ADC1_SE4**b**. That is, there are two input pins with the name ADC1_SE4; one on **channel 'a'** and the other on **channel 'b'** of it. Defining which of the two possible inputs that have been selected by using `ADC_SE4_SINGLE` is actually connected is then controlled by ADC1_CFG2 [MUXSEL] .

Often there is no channel b pins available and there is nothing specific in the pin's name. In this case the channel input is inherently „channel a“ and one can imagine that the full name of the pin is in fact with a small 'a' at the end.

When selecting channel b inputs in the µTasker interface the flag `ADC_SELECT_INPUTS_B` is set when configuring the channel in question. For other channels either `ADC_SELECT_INPUTS_A` can be specifically specified or else no flag needs to be set (default is for channel a since these channels are much more common).

It is interesting to note that if channel b is selected for input A, this setting is also valid for Input B; *it is not possible to select these differently when using two inputs in HW mode!*

Note finally that the illustration is a representation of the path selection. It is equally possible that each channel has its own MUXSEL switch (for a and b inputs) rather than a general one before the input; the overall results is the same.

e) Kinetis ADC Input A/B HW Triggering with TPM

The Kinetis ADC usually includes 2 inputs called input A and B (see previous section), although conceptionally the number could be more based on the ADC design. Only the A input can be read using software triggered operation but the other input (or inputs) can be used when hardware triggering is used.

This section is particularly relevant for devices that do not have a PDB (Programmable Delay Block) since it discusses how both ADC inputs can be used when triggering from the TPM (Timer/PWM Module). In fact, although there are various sources for triggering a single input (either input A or input B), such as PIT, RTC, only the TPM is capable of allowing both inputs to be *continuously sampled without software intervention between sample*s. The TPM operation is thus highly useful in systems using ADC/DMA methods to achieve high speed with minimal CPU overhead.

This discussion uses the KL26, where TPM 1 channels 0 and 1 are defined in its hardware as the trigger sources for ADC0 input A and B respectively. Although other processors follow this layout it is advisable to check the details of the particular processor used in case of any deviations in its actual configuration.

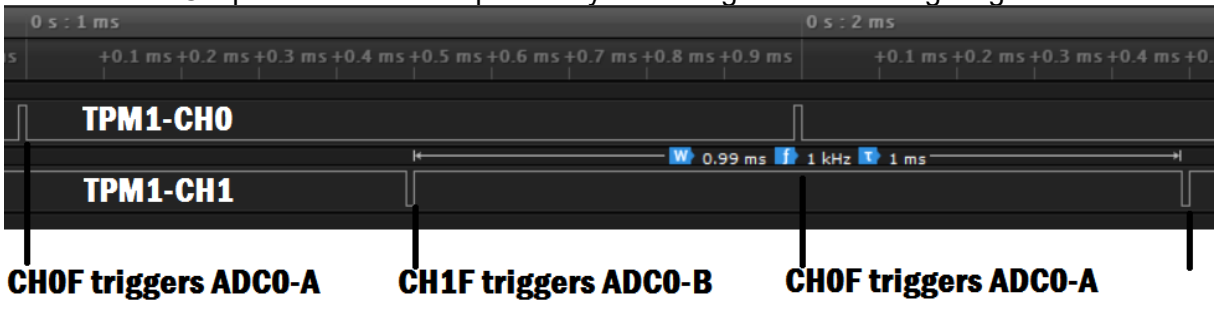
In order to illustrate the setup and operation the KL delay line reference (see Appendix A – b for a general description based on a single input and PIT triggering) is used. The only change made is to perform triggering by the appropriate TPM channels so that two input samples are available after each sampling period. ADC input A is used as input to the delay line and ADC input B is used by software, whereby the software shows how this input's source is simply changed as required to allow a number of multiplexed inputs to be sampled in parallel to the main delay line operation.

TPM Configuration:

```
PWM_INTERRUPT_SETUP pwm_setup;
pwm_setup.int_type = PWM_INTERRUPT;
pwm_setup.pwm_mode = (PWM_SYS_CLK | PWM_PRESCALER_16 | PWM_CENTER_ALIGNED | PWM_DMA_PERIOD_ENABLE |
PWM_NO_OUTPUT);
                     // clock PWM timer from the system clock with /16 pre-scaler (don't use an output)
pwm_setup.int_handler = 0;                           // no user interrupt call-back on PWM cycle
pwm_setup.pwm_frequency = PWM_FREQUENCY(1000, 16); // generate 1kHz on PWM output
pwm_setup.pwm_value = _PWM_PERCENT(1, pwm_setup.pwm_frequency); // 1% PWM (high/low)
pwm_setup.pwm_reference = (_TIMER_1 | 0);        // timer module 1, channel 0 (triggers ADC0 input A)
fnConfigureInterrupt((void *)&pwm_setup);
pwm_setup.pwm_value = _PWM_PERCENT(99, pwm_setup.pwm_frequency); // 99% PWM (high/low)
pwm_setup.pwm_reference = (_TIMER_1 | 1);        // timer module 1, channel 1 (triggers ADC0 input B)
fnConfigureInterrupt((void *)&pwm_setup);
```

The following details are to be noted:

1. Two PWM channels are configured to operate a 1kHz, which determines the base sampling rate.
2. By configuring different PWM mark space ratios for the two PWM channels the triggering of the two ADC inputs is controlled specifically according to the following diagram:



This shows that the PWM setting has been chosen so that both ADC inputs are triggered at equally spaced points in the cycle so that each has the maximum conversion time available in the cycle. Various PWM settings can be used to control this, meaning that this is just one possible setup.
3. The TPM channel's flag CH0F or CH1F is set when the PWM mark-space-ratio point is reached and used internally to trigger the respective ADC input channel conversion.
4. PWM_NO_OUTPUT is specified if the PWM output is not needed to appear on a pin. The previous recording was made without this flag so that the timing could be checked.
5. PWM_DMA_PERIOD_ENABLE is required by KL devices to enable the PWM cycle to be used as DMA trigger for the ADC.
6. The ADC and DAC DMA trigger sources can be set to the PWM cycle trigger in case DMA is not streamed on both ADC inputs. The reason for this is that the ADC0 input B will not be read and so the ADC DMA trigger will remain pending, causes the DMA to fire multiple times. Using the PWM overflow DMA trigger instead ensures that there is only one DMA trigger per cycle and this can be set using:

```
adc_setup.ucDmaTriggerSource = DMAMUX0_CHCFG_SOURCE_TPM1_OVERFLOW; // trigger DMA TPM overflow
dac_setup.ucDmaTriggerSource = DMAMUX0_CHCFG_SOURCE_TPM1_OVERFLOW; // trigger DMA/DAC on TPM overflow
```

7. When setting up the ADC in hardware triggered mode the initial ADC input B multiplexing is controlled by the configuration (example):

```
adc_setup.int_adc_mode |= ADC_DIFFERENTIAL_B;
adc_setup.int_adc_bit_b = ADC_D0_DIFF;  // channel B only valid in HW triggered mode
```

Once the operation has started the ADC0 Input B result can be read directly by reading the ADC0_RB register. A new ADC0 input B multiplexer setting can be defined by writing ADC0_SC1B. Between setting a new multiplexer input and a new sample being ready it is necessary to wait until a further sample has been triggered and the ADC conversion is ready, whereby if the sampling of this input is being performed at a periodic rate slower than the ADC sampling frequency the following code would be suitable to sequentially sample a number of inputs:

```
#define MUX_INPUT_COUNT    4
static const unsigned long ulNextMuxInput[MUX_INPUT_COUNT] = {
    ADC_SE4_SINGLE, ADC_SE12_SINGLE, (ADC_D3_DIFF | ADC_SC1A_DIFF), ADC_TEMP_SENSOR
};
static unsigned short fnSampleInput(void) // called periodically to collect result and set next mux input
{
    static int InputMuxSelect = 0;
    unsigned short usResult = ADC0_RB;          // read the result of the previous B input sample
    if (++InputMuxSelect >= MUX_INPUT_COUNT) {  // cycle though inputs
        InputMuxSelect = 0;                     // set back to the first input
    }
    ADC0_SC1B = ulNextMuxInput[InputMuxSelect]; // set next B input to be sampled
    return usResult;
}
```

f) i.MX RT Temperature Monitor

The i.MX RT 10xx includes a temperature monitor module (TEMPMON) that allows interrupts to be generated if the core temperature exceeds two programmable high temperature levels or falls below a low programmable lower limit level.

Software can also read the present die temperature at any time.

In order for it to work correctly it needs the bandgap reference to be enabled, plus the 480MHz PLL and the 32kHz RTC modules to be operating.

The temperature monitor is factory calibrated and the calibration values can be read from the HW_OCOTP_ANA1 registers. These are used by software to extrapolate the temperature and also to correctly set temperature limits.

The support in the µTasker project is enabled with the define SUPPORT_TEMPMON which adds an interface to read the core temperature via the ADC API in a compatible manner for projects that also run on processors that use ADC based temperature reading.

An example of reading the temperature periodically can be activated in ADC_Timers.h by activating the define ADC_INTERNAL_TEMPERATURE when using the ADC reference. For compatibility the ADC API is used with the input set to

```
adc_setup.int_adc_bit = ADC_TEMP_SENSOR; // ADC internal temperature
```

which is how ADC based temperature reading is performed.

Although there is no interrupt generated when the measurement has completed such an interrupt is emulated so that applications remain compatible. The only difference is that the result returned when collecting the value is in °C x 100 (allowing hundredth of degree resolution) and not the raw ADC value itself. Therefore the only modification at the application level is to remove any HW specific conversion that may originally have been performed and use the result directly (or after modification to the desired form). The following shows retrieving and rounding to 1°C resolution:

```
ADC_SETUP adc_setup; // interrupt configuration parameters
ADC_RESULTS results;
adc_setup.int_type = ADC_INTERRUPT; // identifier
adc_setup.int_adc_mode = (ADC_READ_ONLY | ADC_GET_RESULT);
adc_setup.int_adc_controller = 0;
adc_setup.int_adc_result = &results;
fnConfigureInterrupt((void *)&adc_setup);
results.sADC_value[0] += 50;
results.sADC_value[0] /= 100; // the approximate temperature
rounded up/down to 1°C
fnDebugDec(results.sADC_value[0], DISPLAY_NEGATIVE);
fnDebugMsg(" degC\r\n");
```

In this particular case the conversion was started previously and it shows just the subsequent retrieval. The conversion was started using the standard API for the ADC with the ADC_TEMP_SENSOR defined as input. When multiple ADC controllers are implemented in the i.MX RT the same one should be referenced for the conversion and retrieval, although there is only one TEMPMON module shared by both and the ADC controller is not actually used.