FRONTGRADE

APPLICATION NOTE

UT32M0R500

GPIO as PWMs

7/6/2021
Version #: 1.0.0

Table 1: Cross Reference of Applicable Products

Product Name	Manufacturer Part Number	SMD#	Device Type
Arm Cortex M0+	UT32M0R500	5962-17212	01

Overview

This Application Note describes how to program the UT32M0R500's General Purpose Input/Output (GPIO) pins to function as Pulse Width Modulator (PWM) signals. A PWM is a digital signal with controllable frequency, duty cycle, and dead band time, and is commonly used to control motors, illuminate LEDs, and more. GPIO pins, when used in combination with an interrupt timer, can replicate this behavior.

Creating a Basic PWM Signal

To create a basic PWM, users should select one of the UT32M0R500's timer peripherals. Such peripherals include the Dual Timers (DTIMER), Real Timer Counter (RTC), System Tick (SysTick), and even the Pulse Width Modulator (PWM). By setting a static period value for the timer, and then XOR-ing the GPIO pin every interrupt, a PWM signal with a 50% duty-cycle is output, as shown in Figure 1.

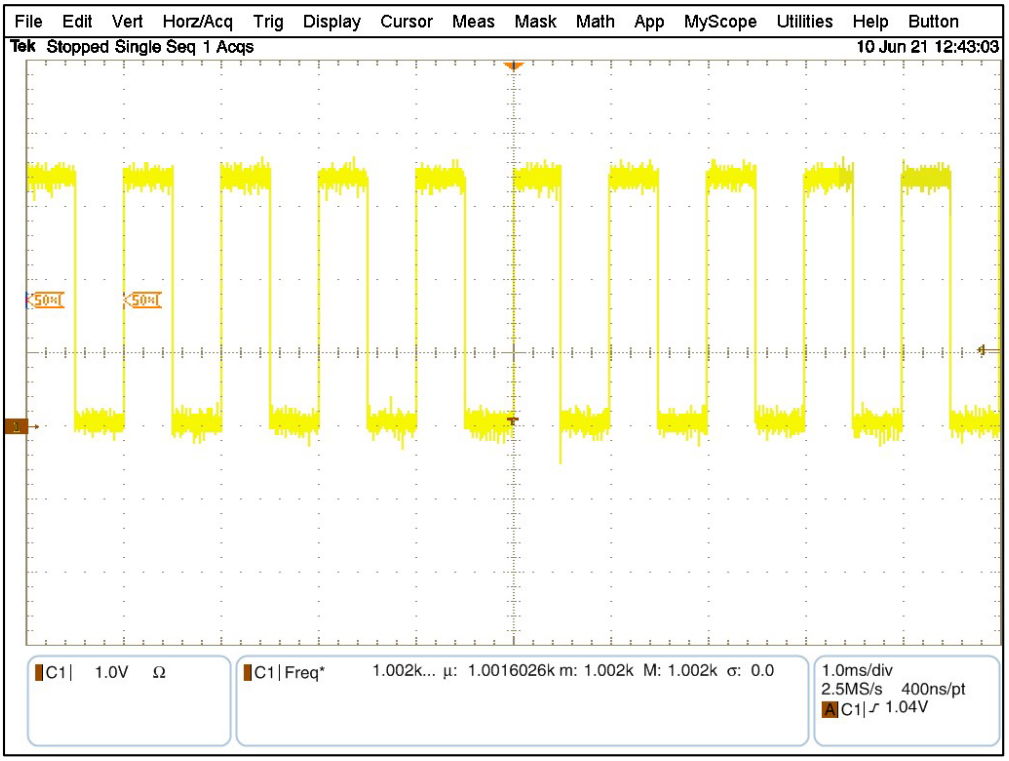

Figure 1: A 1kHz PWM signal with a 50% Duty Cycle

Code 1 shows the DTIMER setup and interrupt function used to generate a 1kHz PWM wave. Users should note that one period requires two interrupts. One interrupt toggles the GPIO pin high, and the next interrupt toggles the GPIO pin low.

```
void DTIMER_Setup(void){
        DTIMER_StructInit(&DTIMER_InitStruct);
        DTIMER_InitStruct.TIMER0.MODE = PERIODIC;
        DTIMER_InitStruct.TIMER0.SIZE = TIMER_32BIT;
        DTIMER_InitStruct.TIMER0.LOAD_VAL = 50000; //50MHz / 1kHz = 50k counts; 50k counts * 20ns = 1ms
        DTIMER_InitStruct.TIMER0.INTRPT_ENABLE = INT_ENABLE;
        DTIMER_InitStruct.TIMER0.ENABLE = TIMER_ENABLE;
        DTIMER_Init(DTIMER0, &DTIMER_InitStruct);
        DTIMER_Cmd(DTIMER0,TIMER0,TIMER_ENABLE);

        NVIC_SetPriority(DUALTIMER0_IRQn, 1); //low priority to ensure accurate PWM waveform
        NVIC_EnableIRQ(DUALTIMER0_IRQn);
}

void DUALTIMER0_IRQHandler(void){
        DTIMER_ClearIRQ(DTIMER0, TIMER0); //clear peripheral interrupt
        GPIO0->DATA ^= GPIOasPWM_pin; //Use XNOR to flip bit(s) covered by the GPIOasPWM_pin variable
                //fewer logic steps vs if/then to determine the pin state
        NVIC_ClearPendingIRQ(DUALTIMER0_IRQn); //clear NVIC interrupt
}
```

Code 1: DTIMER Setup and Interrupt functions for a 1kHz PWM wave

Adding a Duty Cycle

Users can change the period of the running PWM signal by updating the DTIMER's Load register (TIMER0BGLOAD). To add duty-cycle functionality to the PWM signal, users need to be able to program alternating interrupt times that add up to the total desired period. Code 2 shows how to use global variables to set the desired period, duty cycle, and compare values. Additionally, Code 2 shows how to update the DTIMER period in the ISR. Figure 2 shows the resulting output.

```
uint32_t GPIOasPWM_Period;
uint32_t GPIOasPWM_Compare;
uint32_t GPIOasPWM_DutyCycle;

GPIOasPWM_Period = 50000;//DTIMER period = d50000 for 1kHz
GPIOasPWM_DutyCycle = 33;
GPIOasPWM_Compare_Hi = GPIOasPWM_Period * GPIOasPWM_DutyCycle / 100;
GPIOasPWM_Compare_Lo = GPIOasPWM_Period – GPIOasPWM_Compare_Hi;
        //Load either Compare value into the DTIMER, then alternate between the two
void DUALTIMER0_IRQHandler(void){
        DTIMER_ClearIRQ(DTIMER0, TIMER0); //clear peripheral interrupt
        GPIO0->DATA ^= GPIOasPWM_pin; //Use XNOR to flip bit(s) covered by the GPIOasPWM_pin variable
                //fewer logic steps vs if/then to determine the pin state
        //Duty Cycle
        if( DTIMER0->Timer0Load == GPIOasPWM_Compare_Hi){
                DTIMER0->Timer0Load = GPIOasPWM_Compare_Lo; }
        else{
                DTIMER0->Timer0Load = GPIOasPWM_Compare_Hi;
        }
NVIC_ClearPendingIRQ(DUALTIMER0_IRQn);//clear NVIC interrupt
}
```

Code 2: The DTIMER Interrupt with Duty Cycle functionality

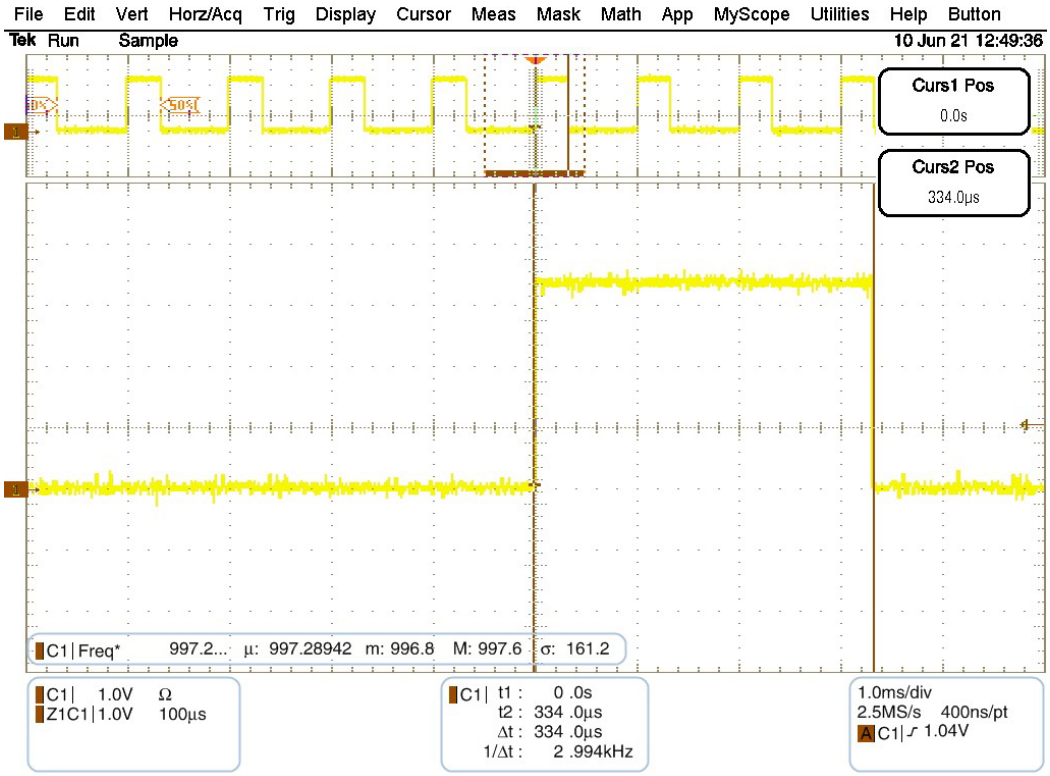

Figure 2: A 1kHz PWM wave with a 33% Duty Cycle

Adding a Dead Band Time

When using multiple GPIO pins to control a motor, users may need to ensure that only one signal is ever asserted at a time. Dead Band time is a delay between a signal de-asserting and the next one asserting, as shown in Figure 3.

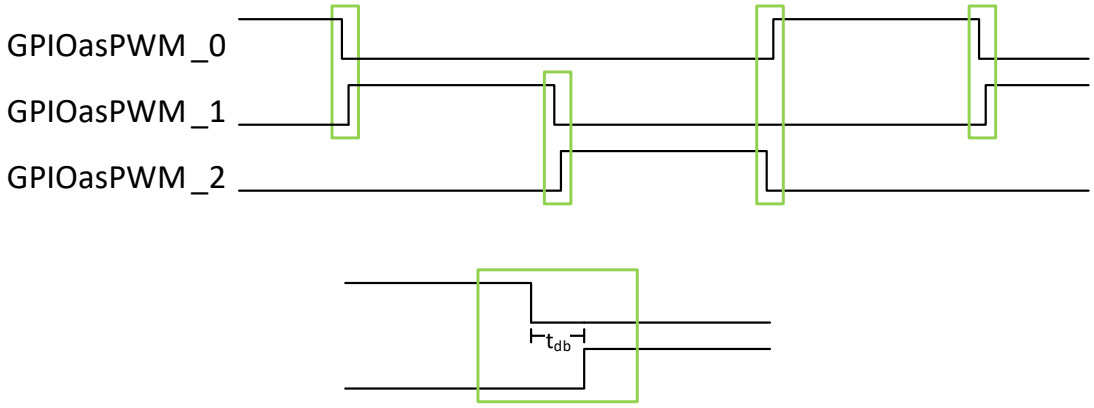

Figure 3: Dead Band time (t_{db}) between alternating PWM signals

When manually using GPIO pins as PWMs, Dead Band is first ensured by de-asserting all of the relevant GPIO pins, and then asserting the next GPIO pin in the sequence. Users can increase the Dead Band time by inserting a delay between the two operations. Dead Band time increases the amount of time required to service the GPIO-as-PWM interrupt, and large dead-band times will impact the processing time available to other code, such that users should carefully consider how much Dead Band time they use. Figure 4 shows the minimum dead-band time (Δt = 618ns) to clear GPIO pins and then write to the next in the sequence (in Code 3).

```
uint8_t GPIOasPWM_DB_PinMask = 0x7; //GPIO pins [2:0]
uint8_t GPIOasPWM_DB_State; //keeps track of the output state
uint8_t GPIOasPWM_DB_Values[3] = {
        0x1, //b001
        0x2, //b010
        0x4 //b100
};

void DUALTIMER0_IRQHandler(void){
        DTIMER_ClearIRQ(DTIMER0, TIMER0); //clear peripheral interrupt

        GPIO0->DATA &= ~GPIOasPWM_DB_PinMask;//turn all GPIOasPWM off
                //change the GPIO pins to enter the next state
        GPIO0->DATA = (GPIO0->DATA & ~GPIOasPWM_DB_PinMask) |
                (GPIOasPWM_DB_Values[GPIOasPWM_DB_State] & GPIOasPWM_DB_PinMask);
        GPIOasPWM_DB_State++;
        if(GPIOasPWM_DB_State >= 3){ //3 total Values in the _DB_Values array, circle back around
                GPIOasPWM_DB_State = 0;
        }
        NVIC_ClearPendingIRQ(DUALTIMER0_IRQn); //clear NVIC interrupt
}
```

Code 3: The DTIMER Interrupt with Minimum Dead Band Time

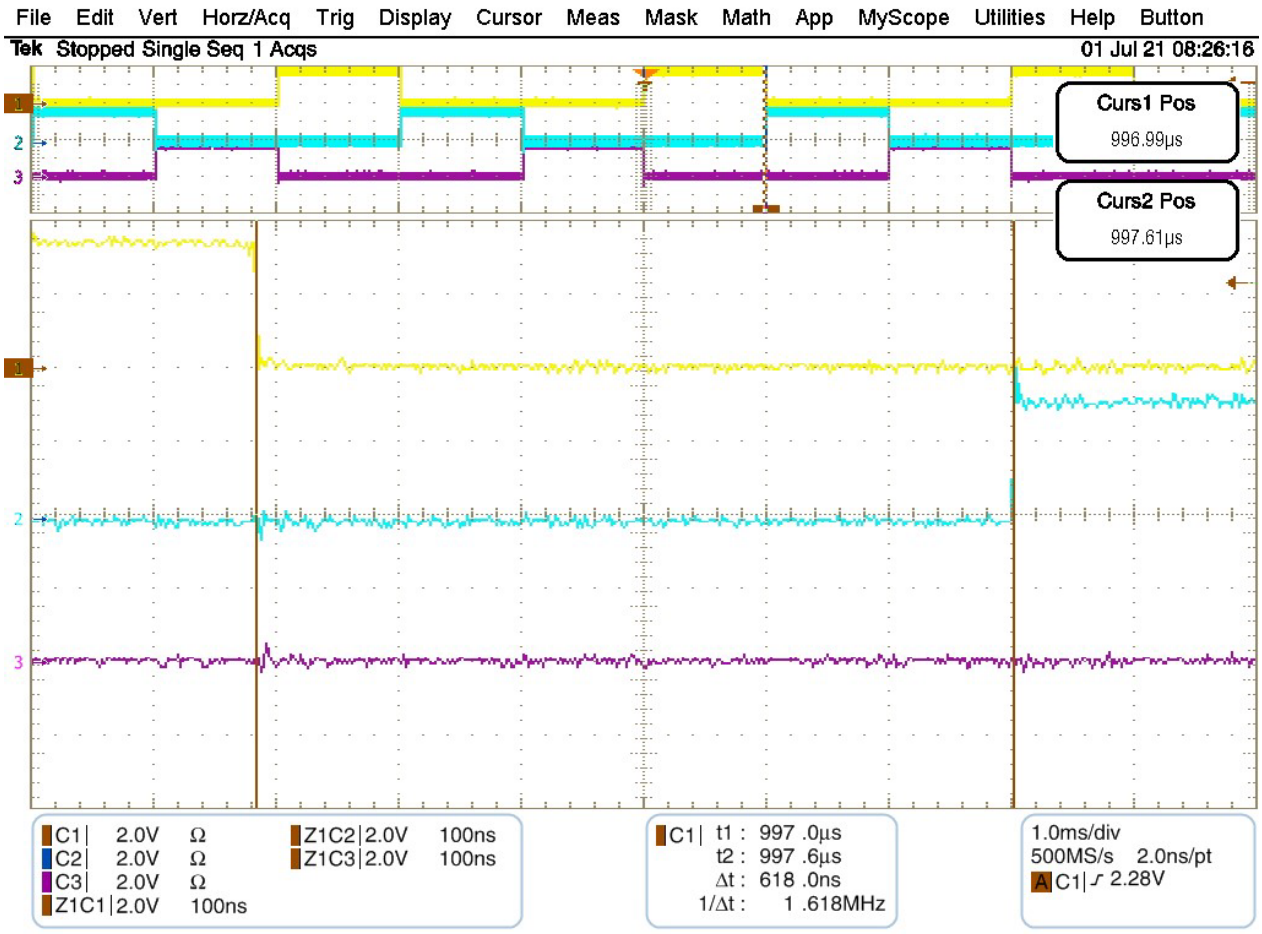

Figure 4: Minimum Dead-Band Delay between two GPIO pins

Code 4 adds additional delay using No-Op instructions. The resulting output is in Figure 5, with the Dead Band time measured as Δt = 10.68µs.

```
        uint8_t GPIOasPWM_DB_PinMask = 0x7;
        uint8_t GPIOasPWM_DB_State;
        uint8_t GPIOasPWM_DB_Values[3] = {0x1, 0x2, 0x4}; //b001, b010, b100 respectively

        void DUALTIMER0_IRQHandler(void){
                uint8_t i;
                DTIMER_ClearIRQ(DTIMER0, TIMER0); //clear peripheral interrupt

                GPIO0->DATA &= ~GPIOasPWM_DB_PinMask; //turn all GPIOasPWM off for(i=0;i<100;i++){
                        //Using __NOPs to increase the Dead Band Time
                __nop();
        }
        //change the GPIO pins to enter the next state
        GPIO0->DATA = (GPIO0->DATA & ~GPIOasPWM_DB_PinMask) |
                (GPIOasPWM_DB_Values[GPIOasPWM_DB_State] & GPIOasPWM_DB_PinMask);
        GPIOasPWM_DB_State++;
        if(GPIOasPWM_DB_State >= 3){ //3 total Values in the _DB_Values array, circle back around
                GPIOasPWM_DB_State = 0;
        }
        NVIC_ClearPendingIRQ(DUALTIMER0_IRQn); //clear NVIC interrupt
}
```

Code 4: The DTIMER Interrupt with a 100-NOP Dead Band

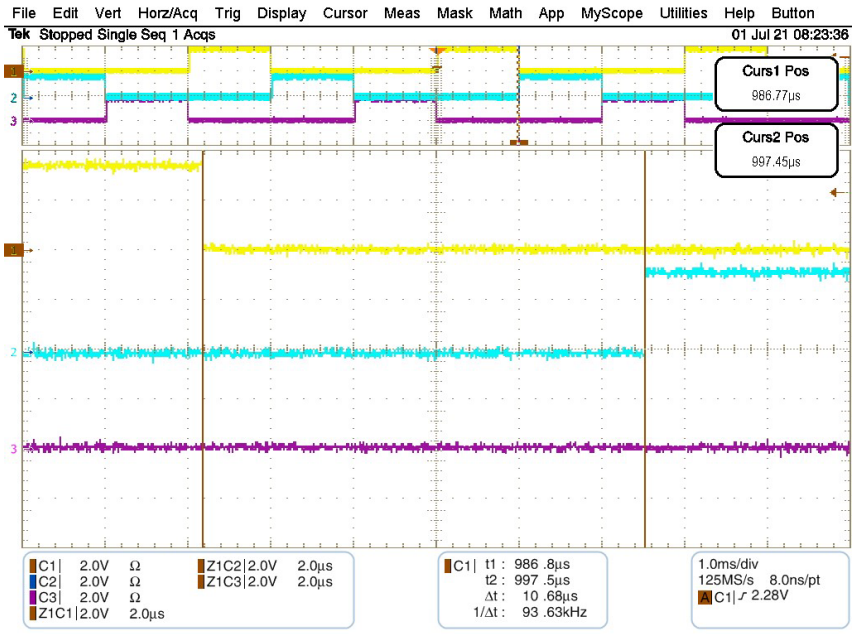

Figure 5: Dead Band Delay with 100 NOP instructions between two GPIO pin writes

For the minimum dead-band time between an external trigger (PWM or GPIO input) and the GPIOasPWM pin, consult the measurements in section 3.

Interrupt Service Routine Timing

Software that relies heavily on periodic interrupts limits the time available to execute other code. This section looks at the amount of time the GPIOasPWM example spends processing an Interrupt Service Routine (ISR).

To understand the time spent servicing an ISR, four different signals were measured:

- **GPIOasPWM** is the GPIO pin acting as a PWM output
- **Start/End ISR** is a GPIO pin set to toggle at the very start and very end of the ISR
  - *Note:* These lines of code provide visual reference for time spent in the ISR, but increase the duration of the interrupt, and do not need to be included in user code. See Table 3 for interrupt timing without these lines of code
- **Main Toggle** is a GPIO pin programmed to toggle as long as the processor is running code in the main while loop
- **EXT Trigger** is an external signal routed to a GPIO input with falling edge interrupt detection, providing the delay between a trigger signal and the GPIOasPWM output changing
  - *Note:* External signals used to trigger a GPIO interrupt can be triggered by either edges or levels (See GPIO INTTYPEx and INTPOLx registers). For this example, use edge detection. By flipping the INTPOLx bit in the ISR, users can trigger a signal on both edges instead of just one.

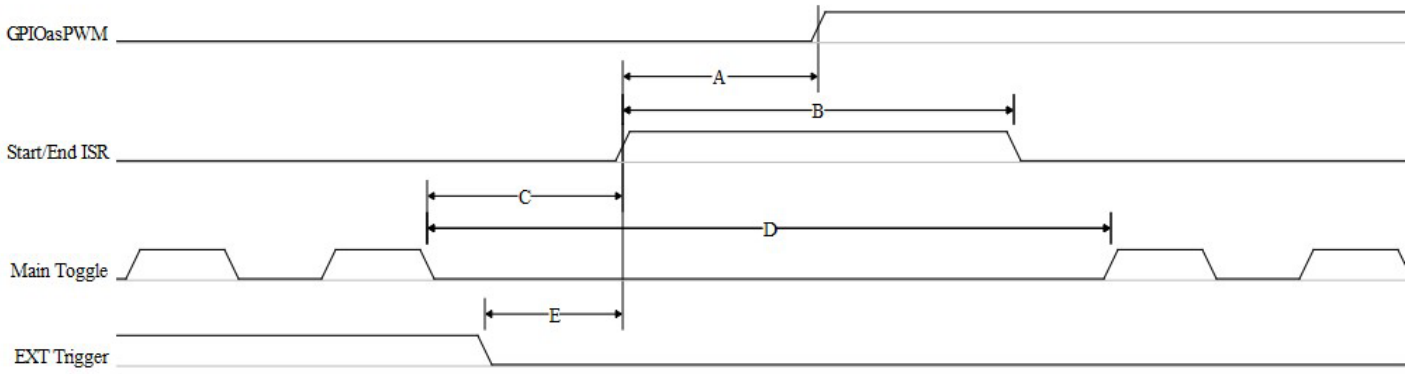

Figure 6: Delays Within various Interrupt Service Routines

Five different peripherals were used to record the data in Table 2. Each ISR had the same bare-bones set of instructions: Toggle the Start/End ISR bit, clear the peripheral interrupt flag, toggle the GPIOasPWM bit, clear the NVIC flag, toggle the Start/End ISR again, and exit. DTIMER0, RTC, PWM, and SysTick are all timer interrupts, while the GPIO Input uses an external signal (such as an external clock or the output of the UT32M0R00's PWM) to trigger the ISR. Comparing the five different peripherals, the context switching and time spent within an ISR is similar. The SysTick interrupt, as it does not require a peripheral flag clear, takes less time to write to the GPIOasPWM bit and spends less time in the ISR.

Table 2: Peripheral vs ISR Duration

Interrupt Source	A (µs)	B (µs)	C (µs)	D (µs)	E (µs)
DTIMER0	0.696	1.296	0.816	2.496	-
RTC	0.752	1.352	0.880	2.576	-
GPIO Input	0.696	1.172	0.800	2.400	0.696
PWM	0.696	1.296	0.656	2.496	0.640
SysTick	0.340	0.820	0.720	2.040	-
Average	0.636	1.187	0.774	2.402	0.669

Users should ensure they set the priority of the NVIC interrupt as low (aka urgent) as possible, to avoid other interrupt signals from context switching away from the GPIOasPWM interrupt. This would delay the GPIOasPWM signal from changing at the correct time. Of the peripherals listed in Table 2, all five are usable sources, but the SysTick has the fastest response times.

Impact of Duty Cycle and Dead Band Time

Table 3 uses the DTIMER0 interrupt to examine the time spent in the ISR (B) as duty cycle and dead band time operations are added to the ISR. Additionally, the time taken to context switch to the ISR, process it, and switch back to the main() program is captured by the D and D (no B toggle) measurements. By knowing the time taken by B and D with the B toggle code included, we can use the D (no B toggle) measurements to calculate the duration spent inside an ISR (without the "Start/End Toggle" code).

Time in the ISR (no B toggle) = D (no B toggle) – [D – B]

Table 3: ISR Time vs Added Functionalities

ISR Features	B (µs)	D (µs)	D (no B toggle) (µs)	Time in the ISR (no B toggle) (µs)
GPIO Toggle Only	1.296	2.496	1.936	0.736
With Duty Cycle	1.840	3.056	2.600	1.384
With a 100 NOP Dead Band	12.28	13.48	12.88	11.68

Processing Time Versus Frequency

As the GPIOasPWM frequency increases, the amount of time available for all other code to run per period decreases. Using the timing measurements from Section 3.1, we can look at the frequency versus the amount of time the processor can dedicate to non-GPIOasPWM code. Note the time spent servicing the ISR per period in table 4 is equal to two times the "D (no B toggle)" from the "With Duty Cycle" row from Section 3.1, to account for context switching and two interrupts per period.

Table 4: Frequency vs ISR Processing Time

Frequency (Hz)	Period (µs)	Time spent servicing the ISR per period (µs)	Time spent in other code (µs)	Percentage of time per period servicing the ISR (%)
1,000	1000	5.200	994.8	0.52
5,000	200	5.200	194.8	2.6
10,000	100	5.200	94.8	5.2
20,000	50	5.200	44.8	10.4
40,000	25	5.200	19.8	20.8
80,000	12.5	5.200	7.3	41.6
100,000	10	5.200	4.8	52
200,000	5	5.200	n/a	n/a

When designing software with multiple timing reliant operations, users can use the above table to make informed decisions about the maximum GPIOasPWM frequency their system can accommodate.

Conclusion

Using a GPIO pin as a PWM output requires a programmable periodic interrupt signal, and the UT32M0R500 has plenty to choose from. Doing so is an easy way to bolster the amount of PWM outputs possible, but requires processing overhead that the actual PWM peripheral does not. Using this appnote, designers can determine if using GPIO pins as PWM signals makes sense for their systems.

Revision History

Date	Revision #	Author	Change Description	Page #
07/06/2021	1.0.0	OW	Initial Release	

Datasheet Definitions

	Definition
Advanced Datasheet	Frontgrade reserves the right to make changes to any products and services described herein at any time without notice. The product is still in the development stage and the **datasheet is subject to change**. Specifications can be **TBD** and the part package and pinout are **not final.**
Preliminary Datasheet	Frontgrade reserves the right to make changes to any products and services described herein at any time without notice. The product is in the characterization stage and prototypes are available.
Datasheet	Product is in production and any changes to the product and services described herein will follow a formal customer notification process for form, fit or function changes.