Application Report
SLUA763A–October 2015–Revised May 2020

Automatic Baud Rate Detection and Configuration in UCD3138

Emily Hirsch *High-Performance Isolated Power*

ABSTRACT

When there are major changes in temperature, the UCD3138 internal oscillator frequency can change. See the data sheet for more specific information. There can be a need to implement an automatic baud rate detection and configuration to improve serial communication between the primary and secondary stages of a power supply. This application report discusses how to configure the UCD3138 to match its baud rate to an incoming serial data stream.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

When using UCD3138s for PFC and LLC applications with serial port communication between the two, there can be a slight difference in the two oscillators due to temperature. To compensate for this variation and prevent errors in primary to secondary UART communication, an automatic baud rate matching solution can be implemented into either PFC or LLC. The PFC (primary side) and LLC (secondary side) constantly transmit information to each other such as fault status, operational modes, voltages, and

currents. The information used in this solution is transmitted through UART0 in the PFC and received through UART0 in the LLC. The solution implements a Timer Capture task inside of the 10-kHz timer interrupt to measure the receive baud rate because we can catch both edges of a 4800 single byte pulse. The solution captures one-, two-, three-, or four-bit long pulses and configures the appropriate UART registers in the LLC to match the baud rate of the PFC.

2 Hardware Configuration

There is only one necessary hardware change for this application. In the UCD3138 device, there is currently no internal connection for *Timer Capture* from UART, it is required to externally connect the receive pin, RX0 of UART0 to the TCAP pin in the LLC, as Figure 1 shows. The UCD3138064 device has the ability to internally connect the UART receive pin to the *Timer Capture* pin.

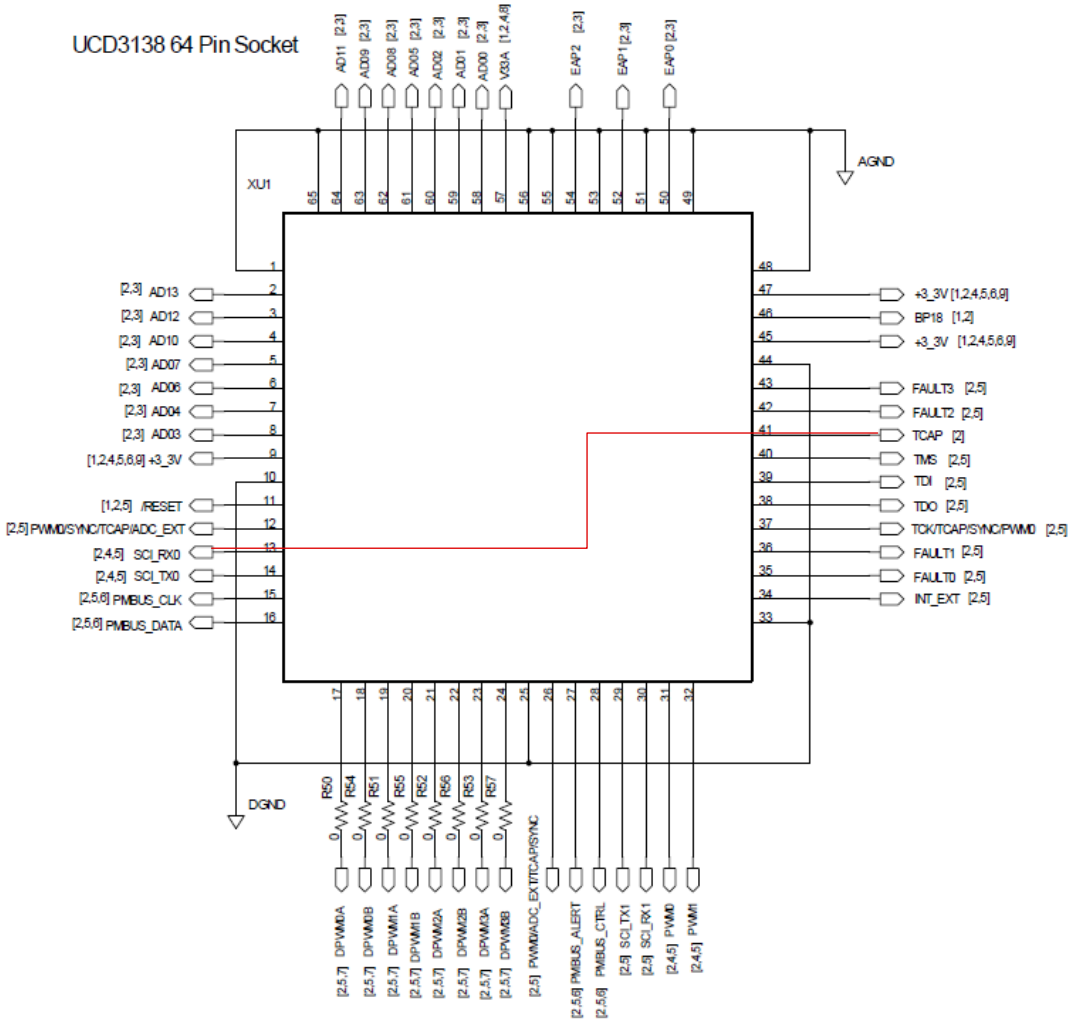


Figure 1. UCD3138 Pin Configuration[1]

On the UCD3138 device, connect pin 13 (SCI_RX0) to pin 41 (TCAP) to capture bit times of received communication.

3 Timer Capture

3.1 Overview

Timer Capture is implemented using a 24-bit free-running timer in the UCD3138. The 24-bit timer is configured to capture edges on the UART receive pin. The measure_baud() function will monitor the waveform. Whenever an edge is captured, the value of the 24-bit counter is captured and put into the capture register, which is read on every edge of the waveform. On the first edge (falling edge) the UCD3138 stores the value of the 24-bit counter and on the second edge (rising edge) it takes the value from the 24-bit counter and subtracts the value found from the first edge to get the pulse width of the signal. The negative pulse width is capture because the idle state of the UART is high, so a short time of idle could seem like a pulse width and the wrong baud rate could be calculated. Figure 2 represents capturing the positive and negative edges of a waveform.

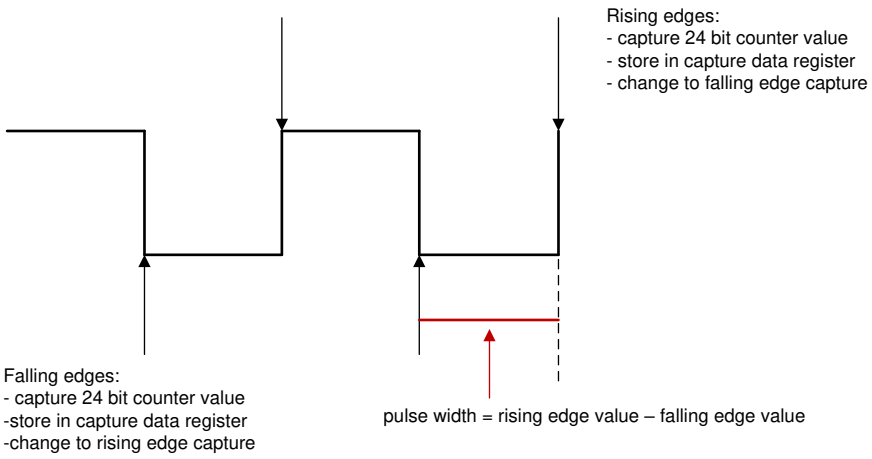


Figure 2. Timer Capture on a Waveform

The clock input for the timer is not prescaled with this application and is solely the ICLK, running at about 15.6 MHz. This method uses polling inside of the interrupt service routine to capture edges, so every 10 kHz (the frequency of the 16-bit timer interrupt) the measure baud rate sequence is executed using timer capture. This solution works for 4800 bps and below because the bit time is less than 5 kHz with the interrupt at 10 KHz, so we can capture each edge with a different interrupt. Faster baud rates than 4800 would require a faster interrupt or the timer capture interrupt. Only the capture block, not the compare block, for the 24-bit timer is used in this application to collect information about the pulse width of the signal received on the LLC. For more information on Timer Capture see the *Timer Module Overview* section of the *UCD3138 Monitoring and Communications Programmer's Manual*.

3.2 Timer Capture Registers

A few registers for Timer Capture must be set for this application. The input signal for Timer Capture must be set to the SCI_RX0 pin. to ensure that the negative pulse width is being calculated, the edge must originally be set to capture on the falling edge of the input signal. This is changed later in the code so it only captures the negative pulse width. The registers that must be set, along with the values they need to be set to are shown in the following:

```
TimerRegs.T24CAPCTRL.bit.CAP_SEL = 1; //input signal comes from SCI_RX0 pin
TimerRegs.T24CAPCTRL.bit.EDGE = 2; //enable capture on falling edge
```

3.3 Timer Capture Polling

The timer capture polling can be integrated into the standard interrupt in the LLC code. Every time the timer interrupt is triggered, all global tasks are handled, and the measure baud function has been included in the global tasks along with collecting ADC data, averaging ADC readings, receiving UART data, and calling the pgood handler. The full code is shown in the *Installing Code into Existing Program* section. Whenever the first falling edge has been captured, the capture interrupt flag is set and the value of the 24-bit counter is stored into a data register for the capture module. to get the negative pulse length, the edge must be switched to capture the rising edge. The two edges are captured and the value captured at the falling edge must be subtracted from the value captured at the rising edge. The measure_baud() function is included in the following:

```
void measure_baud(void)
{
     T24SREG = TimerRegs.T24CAPCTRL.bit.CAP_INT_FLAG;
     if((T24SREG == 1) && (edge == 0)) //first edge detected
     {
          result = TimerRegs.T24CAPDAT.bit.CAP_DAT; //read and clear
               //enable capture on rising edge
          TimerRegs.T24CAPCTRL.bit.EDGE = 1;
          edge = 1; //capture second edge
     }
     else if((T24SREG == 1) && (edge == 1)) //second edge detected
     {
               //read and clear register value
          result = TimerRegs.T24CAPDAT.bit.CAP_DAT – result;
               //result contains pulse width
          pulse_width = result; //store result in pulse_width
               //enable capture on falling edge
          TimerRegs.T24CAPCTRL.bit.EDGE = 2;
          edge = 0; //reset edge to capture first edge again
     }
     else
     {
          //do nothing
     }
}
```

By reading the CAP_INT_FLAG register into the variable “T24SREG,” the UCD3138 clears the interrupt flag and stores the flag data into T24SREG. The variable “edge” determines which edge (first or second) is being captured. “Result” stores the value captured and puts it into the data register. This result is then stored into “pulse_width” and used in a different function afterwards to change the baud rate.

3.3.1 Number of Bits Captured

The timer capture interrupt will continually capture pulse widths for every bit sent, so it is important to understand what bits of each byte could be captured and how long of a pulse the timer capture will measure. The message sent is from LSB to MSB, which is explained further in the *UART Byte Format* section, and the number of bits captured is important because the baud rate will only be calculated if one, two, three, or four-bits are captured. Table 1 shows what one-, two-, three-, four-, and five-bit pulses looks like from the UART or Timer Capture. As shown, a one-bit pulse width is captured whenever there is a one, followed by one zero, then a one afterward. A two-bit pulse is the same except there are two zeroes, a three-bit pulse has three zeroes, and a four-bit pulse has four zeroes.

Table 1. Number of Bits Captured for Certain Messages

Value Sent (as Seen in Code)	Message Sent (as Seen by UART)	Entire Message Sent With Start and Stop Bits (as Seen by UART)	Possible Number of Bits Captured
Hexadecimal:0x49 Binary:01001001	Hexadecimal: 0x92 Binary: 10010010	01001001011	1 or 2 match baud possible
Hexadecimal: 0xFD Binary: 11111101	Hexadecimal: 0xBF Binary: 10111111	01011111111	1 match baud possible

Table 1. Number of Bits Captured for Certain Messages (continued)

Value Sent (as Seen in Code)	Message Sent (as Seen by UART)	Entire Message Sent With Start and Stop Bits (as Seen by UART)	Possible Number of Bits Captured
Hexadecimal: 0x35 Binary: 00110101	Hexadecimal: 0xAC Binary: 10101100	01010110011	1 or 2 change baud possible
Hexadecimal: 0xC8 Binary: 11001000	Hexadecimal: 0x13 Binary: 00010011	00001001111	2 or 4 match baud possible
Hexadecimal: 0x1A Binary: 00011010	Hexadecimal: 0x58 Binary: 01011000	00101100011	1, 2, or 3 match baud possible
Hexadecimal: 0xE1 Binary: 11100001	Hexadecimal: 0x87 Binary: 10000111	01000011111	1 or 4 match baud possible
Hexadecimal: 0x83 Binary: 10000011	Hexadecimal: 0xC1 Binary: 11000001	01100000111	1 or 5 match baud possible
Hexadecimal: 0xF0 Binary: 11110000	Hexadecimal: 0x0F Binary: 00001111	00000111111	5 match baud impossible

If there are a variety of bytes being sent with at least a majority of them containing a one to four-bit pulse inside, then there is a robust form of communication between the primary and secondary side. With this solution, however, if it does not send any characters that will give a one to four-bit pulse in the byte, this specific automatic baud change application will not work for it.

4 UART

4.1 Overview

There are two UARTs, or universal asynchronous receiver-transmitters, in UCD3138. These UARTs are used to transmit information between the primary and secondary sides of the power supply. For this solution, UART0 is used on both the primary and secondary sides to send and receive bytes, and the bytes transmitted are used to capture pulse widths of the primary side. Baud rates are set using the baud divide registers and on the secondary side, they are changed after initialization to match the baud rate of the primary side. The received bytes come in to the data register in the UART receive buffer and can be viewed on the UART0 receive pin of UCD3138. The transmitted bytes transmit from the UART transmit buffer and can be viewed on the UART0 transmit pin of UCD3138. More information on UARTs in the UCD3138 is found in the section of the *UCD3138 Monitoring and Communications Programmer's Manual*.

4.2 UART0 Baud Divide Registers

The baud rate is adjusted using the baud divide registers. There are three divide registers for each UART, a low divide, middle divide, and high divide[2].The overall register value is found using Equation 1, where "register value" represents the overall value that will be split into low, middle, and high divide values; "ICLK" represents the internal 15.625-MHz clock; "baud" represents the desired baud rate. After determining the overall register value, the low divide value is found by converting the register value from decimal to hexadecimal, then AND it with 0xFF. The middle divide value is found by shifting the hexadecimal register value to the right 8 bits. The high divide value is always zero for this application.

$$\text{register value} = \frac{\text{ICLK}}{8 \times \text{baud}} - 1 \tag{1}$$

For example, the divide values for a baud rate of 4800 bps are configured using either of these two sets of statements:

Example 1:

```
        //BAUD_RATE set to 4800 bps, BAUD_RATE_VALUE and divide values
        //are calculated in code, rather than by hand

#define BAUD_RATE             4800
#define BAUD_RATE_VALUE        (15625000/ (8* (BAUD_RATE + 1) ) )
#define BAUD_RATE_VALUE_M    (BAUD_RATE_VALUE & 0xFF)
#define BAUD_RATE_VALUE_L    (BAUD_RATE_VALUE >> 8)
```

```
Uart0Regs.UARTHBAUD.bit.BAUD_DIV_H = 0;        //always zero for our purposes
Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = BAUD_RATE_VALUE_M;
Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = BAUD_RATE_VALUE_L;
```

Example 2:

```
      //baud divide values calculated by hand and input directly into code
   Uart0Regs.UARTHBAUD.bit.BAUD_DIV_H = 0;        //always zero for our purposes
   Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = 1;        //overall register value shifted right 8 bits
   Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = 149;    //overall register value AND 0xFF
```

NOTE: When calculated baud values by hand, as in *Example 2* note to truncate the decimal. Implementing the equation in code, as in Example 1, the equation will always round down.

4.3 UART Byte Format

For this application, when a message is sent through UART, it contains a start bit, 8 bits of data and two stop bits, shown in Figure 3. The idle state is high, so the start bit is always low and the two stop bits are always high, with the message in between. The message is sent from Least Significant Bit to Most Significant Bit, so UART0 on the LLC receives the bits from LSB to MSB.

Figure 3. UART Message Format

5 Baud Rate Matching Firmware

5.1 Timer Value to Baud Rate Translation

After the timer capture interrupt has captured both edges of a pulse, it has a pulse width for an unknown number of bits between one and nine bits. This bit time is then sent into a function to change the baud rate for both of the UARTs in the LLC. In this function, even though it changes the baud rate, the baud rate is never actually calculated in order to save time and memory. Instead, the formula for translating the bit time to baud rate (Equation 3) and the baud rate to the overall baud value (Equation 4) is shortened to an equation taking the bit time and finding the overall baud value (Section 5.3.2 or Equation 5) in the UART0 register, shown in Equation 2 to Equation 5. After finding the register value from Equation 5, the middle and low divide values are found using the method explained in the *UART0 Baud Divide Registers* section.

$$\text{pulse} = \frac{\text{ICLK}}{\text{baud}} \tag{2}$$

$$\text{register value} = \frac{\text{ICLK}}{8 \times \text{baud}} - 1 \tag{3}$$

$$\text{register value} = \frac{\text{pulse}}{8} - 1 \tag{4}$$

$$\text{register value} = (\text{pulse} >> 3) - 1 \tag{5}$$

This final equation (Equation 5) can only be used if one bit pulse was captured. If more than one bit pulse was captured, there has to be a compensation to account for the multiple bits acquired.

5.2 Need for UART Reset

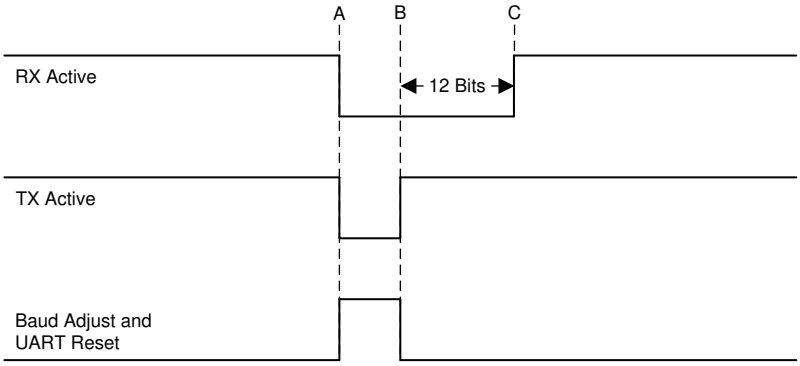
Writing new lower values to the UART BAUD_DIV registers at a narrow time window close to the end of the bit period can cause an 8-second lockup of the UART. To avoid this, resetting the UART right after writing to the BAUD_DIV registers is recommended. This is done with these two lines of code:

```
Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
        Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
```

These two lines of code are seen in the code examples in Section 5.3, *Different Bit Pulse Widths*.

Note that resetting the UART will probably cause any active incoming or outgoing bytes to be corrupted. To avoid this, the match baud function must be called when no messages are active. Generally it is always possible for the firmware to determine when there are outgoing messages, and avoid those. If it is not possible to predict incoming messages, then sufficient error handling, handshaking, and redundancy must be provided to deal with a corrupted incoming message. Especially in the noisy power supply environment, corrupted messages should be protected against in any case.

For receive, there is an additional timing requirement as Figure 4 shows.



(1) Time A is the time at which all bytes on the receive message have been received by the UART and read by the firmware. Also it means that all bytes have finished transmitting.

(2) Time B is the time where the Baud registers have been changed, and the UART reset bit has gone low and then high. At time B, the firmware can write a new byte to the transmit register.

(3) Time C is the time where the receive logic on the UART will be ready to accept a new byte. To ensure that the start bit really indicates the start of a new byte, the UART logic waits for at least 11 bit periods of an idle state on the receive pin before it starts accepting data. The transmit can be started as soon as the reset bit is turned back to a 1. If the RX line goes active during the wait, the bit counter is reset, and it waits for 11 bits from that point. Twelve bits of delay are recommended to provide a bit of a guard band.

Figure 4. Recieve Restart Timing

5.3 Different Bit Pulse Widths

For this application, the baud rate is only changed if one-, two-, or three-bit pulses have been captured and we assume the baud rate is from ±10% from 4800 bps. If the pulse length is longer than expected for 3 bits at the slowest baud rate or shorter than 1 bit at the fastest baud rate, the baud rate isn't changed and the information is treated as an error, or garbage. A block diagram of the changing baud function is shown in Figure 5.

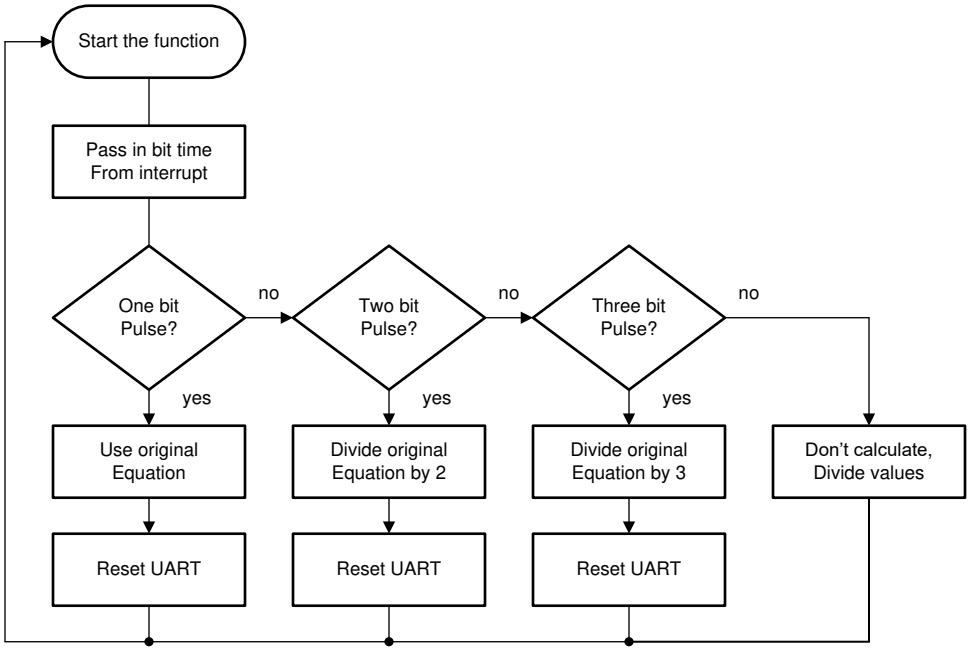


Figure 5. Flowchart of Baud Rate Configuration

5.3.1 One-Bit Pulse

As previously mentioned, if a one-bit pulse is captured, Equation 5 is the function used to calculate the baud divide values from the bit time. Since a one-bit pulse is an exact interpretation of Equation 2 through Equation 5, pulse-width times, called "bit_time" in the code, are calculated using Equation 2. The code used to calculate the baud divide values when a one-bit pulse has been found is included in the following:

```
     //calculated correct pulse width (+/- 10% from 4800 baud)
if ((bit_time >= 2950) && (bit_time <= 3260))
{
     baud_div_value = (bit_time >> 3) - 1;
     Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
}
```

As shown in the code, the bit times for a one-bit pulse must be between 2950 and 3260 to calculate the baud divide values. This was determined by taking a baud rate of 4800 bps and calculating what baud rates would be ±10% from 4800 bps and converting that to the bit time. Table 2 shows the bit time, baud rate, and percent difference from 4800 bps.

Table 2. One-Bit Pulse Bit Time and Baud Rate

Bit Time	Corresponding Baud	Baud With Adjustment	± From 4800
2950	5296	5296	+10.33%
3255	4800	4800	0.00%
3620	4316	4316	–10.08%

5.3.2 Two-Bit Pulse

If a two-bit pulse is captured, Equation 5 needs to be modified to account for the longer pulse width captured. Since the time captured is twice the actual time for one bit, the equation must be divided by two. The equation and code used to calculate the baud divide values when a two-bit pulse has been found are included in the following:

Equation 4 modified:

$$\text{register value} = \frac{\text{pulse}}{16} - 1 \tag{6}$$

Equation 5 modified:

$$\text{register value} = \left(\left(\text{pulse} >> 4\right) \,\&\, 0\text{xffff}\right) - 1 \tag{7}$$

```
     //calculated 2 times pulse width (+/- 10% from 4800 baud)
if ((bit_time >= 5900) && (bit_time <= 7240))
{
     baud_div_value = ((bit_time >> 4)&0xffff) - 1;
     Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
}
```

As shown in the code, the bit times for a two-bit pulse must be between 5900 and 7240 to calculate the baud divide values. This was determined by taking a baud rate of 4800 bps and determining what baud rates would be ±10% from 4800 bps. Those rates must then be divided by 2 to get the “baud rate” that would be calculated for a two-bit pulse and converting that to the bit time. Table 3 shows the bit time, calculated baud rate, baud rate adjusted to be centered around 4800, and percent difference from 4800 bps.

Table 3. Two-Bit Pulse Bit Time and Baud Rate

Bit Time	Corresponding Baud	Baud With Adjustment	± From 4800
5900	2648	5296	+10.33%
6510	2400	4800	0.00%
7240	2158	4316	–10.08%

5.3.3 Three-Bit Pulse

If a three-bit pulse is captured, Equation D is divided by three to account for capturing three times the pulse width centered at 4800 bps. The modified equation and code used to calculate the baud divide values when a three-bit pulse has been found are included in this section:

Equation 4 modified:

$$\text{register value} = \frac{\text{pulse}}{24} - 1 \tag{8}$$

Equation 5 modified:

$$\text{register value} = \left((\text{pulse} \times 2730) >> 16\right) - 1 \tag{9}$$

Code:

```
     //calculated 3 times pulse width (+/- 10% from 4800 baud)
if ((bit_time >= 8850) && (bit_time <= 10860))
{
     baud_div_value = ((bit_time * 2730) >> 16) - 1;
     Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
}
```

As shown in the code, the bit times for a three-bit pulse must be between 8850 and 10860 to calculate the baud divide values. This was determined by taking a baud rate of 4800 bps and calculating what baud rates would be ±10% from 4800 bps. Those rates must then be divided by 3 to get the “baud rate” that would be calculated for a three-bit pulse and converting that to the bit time. Table 4 shows the bit time, calculated baud rate, baud rate adjusted to be centered around 4800, and percent difference from 4800 bps.

Table 4. Three-Bit Pulse Bit Time and Baud Rate

Bit Time	Corresponding Baud	Baud With Adjustment	± From 4800
8850	1765	5295	+10.31%
9765	1600	4800	0.00%
10860	1438	4314	–10.13%

5.3.4 Four-Bit Pulse

If a four-bit pulse is captured, Equation 5 is divided by four to account for capturing four times the pulse width centered at 4800 bps. The modified equation and code used to calculate the baud divide values when a four-bit pulse has been found are included in this section:

Equation 4 modified:

$$\text{register value} = \frac{\text{pulse}}{32} - 1 \tag{10}$$

Equation 5 modified:

$$\text{register value} = \left(\text{pulse} >> 5\right) - 1 \tag{11}$$

Code:

```
     //calculated 4 times pulse width (+/- 10% from 4800 baud)
if ((bit_time >= 11801) && (bit_time <= 14481))
{
     baud_div_value = (bit_time >> 5) - 1;
     Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
```

```
     Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
     Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
     Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
}
```

As shown in the code, the bit times for a four-bit pulse must be between 11801 and 14481 to calculate the baud divide values. This was determined by taking a baud rate of 4800 bps and calculating what baud rates would be ±10% from 4800 bps. Those rates must then be divided by 4 to get the “baud rate” that would be calculated for a four-bit pulse and converting that to the bit time. Table 5 shows the bit time, calculated baud rate, baud rate adjusted to be centered around 4800, and percent difference from 4800 bps.

Table 5. Four-Bit Pulse Bit Time and Baud Rate

Bit Time	Corresponding Baud	Baud With Adjustment	± From 4800
11801	1324	5296	+10.33%
13020	1200	4800	0.00%
14481	1079	4316	–10.08%

5.3.5 Five or More Bit Pulse

If a five-bit pulse or longer has been captured, the baud rate will not match to the PFC because going higher than four bits creates an overlap in the bounds between –10% from 4800 with a 5-bit pulse and +10% from 4800 with a 6-bit pulse so a five and six-bit pulse could be difficult to distinguish from each other.

6 Installing Code into Existing Program

6.1 Overview

Integrating this code into an existing program is relatively simple. The only added code is initializing timer capture, integrating timer capture and the measure_baud function into the standard interrupt service routine, and creating the match_baud function as well as calling it from the main program.

6.2 Initialize Timer Capture

As mentioned in the *Timer Capture Registers* section, there are a few lines of code that must be implemented to capture edges of the signal. The following code sample is the main LLC code with an integrated initialization of the timer capture. Added code is in red and code already inside the LLC code is in black.

```
void init_TCAP(void)
{
      TimerRegs.T24CAPCTRL.bit.CAP_SEL = 0; //input signal comes from TCAP pin
     TimerRegs.T24CAPCTRL.bit.EDGE = 2; //enable capture on falling edge
}
void main()
{
...
     //Initialize global variables.
     init_variables();
     //Initialize GPIO pins and set initial states.
     init_gpio();
     //Initialize UARTs
     init_uart0();
```

```
        init_uart1();
        //Initialize TCAP
        init_TCAP();
        //Initialize TCAP
        init_TCAP();

        //Initialize DPWMs
        init_dpwms();
...
}
```

6.3 Timer Capture and Measure Baud Function

The timer capture inside of the measure baud function is implemented into the standard interrupt service routine. The sample code in this section is the standard interrupt LLC code with an integrated measure baud function into the existing code. Everything changed from the original is in red and the original code is in black.

The measure baud function is a new function to integrate into existing code. It is further explained in the *Baud Rate Matching Firmware* section but the entire function has been integrated into the standard_interrupt.c file with the other functions used in the interrupt service routine.

```
/* standard_interrupt.c
* measure_baud is called inside of the handle_standard_interrupt_global_tasks functions
* it measures the pulse width of the incoming signal from the UART on the PFC
*/
void measure_baud(void)
{
    T24SREG = TimerRegs.T24CAPCTRL.bit.CAP_INT_FLAG;
    if ((T24SREG == 1) && (edge == 0)) //first edge detected
    {
        result = TimerRegs.T24CAPDAT.bit.CAP_DAT; //read and clear
        edge = 1;
            //increase edge to capture second edge
    }
    else if ((T24SREG == 1) && (edge == 1)) //second edge detected
    {
        result = TimerRegs.T24CAPDAT.bit.CAP_DAT - result;
            //result now contains the difference between edges captured
        pulse_width = result;
            //store result into pulse_width - will be passed into match_baud();
        edge = 0;
            //reset edge to detect first edge again
    }
    else
    {
    }
}
```

The measure baud function is called inside of the global tasks function inside of the standard interrupt.

```
/* standard_interrupt.c
* handle_standard_interrupt_global_tasks is called in the standard interrupt
* it handles tasks such as polling the ADC and collecting the data,
* receiving UART messages and using Timer Capture
*/
void handle_standard_interrupt_global_tasks(void)
{
    //Collect ADC data
    poll_adc();
    //Average ADC readings;
    average_adc_readings();
    //Receive UART data
    uart_receive_data();
    //call pgood handler
    handle_pgood();
      //measure baud rate with timer capture
      measure_baud();
}
```

The global tasks function is called inside of the standard interrupt service routine. This is already implemented; it is shown to further explain where everything has been called.

```
/*
 * standard_interrupt.c
 * includes measure baud timer capture interrupt function and uart receive interrupt function
 * as well as standard interrupt
 * the only function changed / used for this application is
handle_standard_interrupt_global_tasks();
 * this function includes the timer capture interrupt and receive data interrupt
*/
#pragma INTERRUPT(standard_interrupt,IRQ)
void standard_interrupt(void)
{
    //Set a GPIO to signify the beginning of the interrupt
//    MiscAnalogRegs.GLBIOVAL.bit.DPWM2A_IO_VALUE = 1;
    //Perform general tasks.
    handle_standard_interrupt_global_tasks();

    switch (supply_state)
    {
    case STATE_IDLE:
        handle_idle_state();
        break;
    case STATE_RAMP_UP:
        handle_ramp_up_state();
        break;
    case STATE_RAMP_DOWN:
        handle_ramp_down_state();
        break;
    case STATE_REGULATED:
        handle_regulation_state();
        break;
    case STATE_VOUT_TRANSITION:
        handle_vout_transition_state();
        break;
    case STATE_LIGHT_LOAD:
        handle_light_load_state();
        break;
    case STATE_CPCC:
        handle_cpcc_state();
        break;
    case STATE_HICCUP:
        handle_hiccup_state();
        break;
```

```
    case STATE_FAULT:
        handle_fault_state();
        break;
    }

    //clear interrupt flag by a read/write to register.
    TimerRegs.T16PWM2CMPCTRL.all = 3;
    //Clear the GPIO to signify the end of the interrupt
//    MiscAnalogRegs.GLBIOVAL.bit.DPWM2A_IO_VALUE = 0;
}
```

6.4 Match Baud Function

The match baud function must be implemented in the LLC code and called in the main function in order for the baud matching to occur.

```
/*
 * match_baud.c
 *
 *      Equations used to find bit times:
 *      baud_rate = ICLK / bit_time
 *      baud_div_value = ICLK/(8*baud_rate) - 1
 *      baud_div_value = bit_time/8 - 1
 *      baud_div_value = (bit_time >> 3) - 1
 */
#include "system_defines.h"
#include "cyclone_device.h"
#include "pmbus_commands.h"
#include "pmbus.h"
#include "variables.h"
#include "function_definitions.h"
#include "software_interrupts.h"
#include "cyclone_defines.h"

#define ICLK    (15625000)

void match_baud(Uint32 bit_time)
{
    if ((bit_time >= 2950) && (bit_time <= 3620))
//calculated correct pulse width (+/- 10% from 4800 baud)
    {
        baud_div_value = (bit_time >> 3)-1;
        Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
        Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
        Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
        Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
        Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
        Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
        Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
        Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
    }
    else if ((bit_time >= 5900) && (bit_time <= 7240))        //calculated 2x pulse width
    {
        baud_div_value = ((bit_time >> 4) & 0xffff)-1;
        Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
        Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
        Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
        Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
        Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
        Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
        Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
        Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;

    }
    else if ((bit_time >= 8850) && (bit_time <= 10860))        //calculated 3x pulse width
    {
```

```
            baud_div_value = (((bit_time >> 4) & 0xffff)*2/3)-1;
            Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
            Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
            Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
            Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
            Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
            Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
            Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
            Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;

        }
        else if ((bit_time >= 11801)&&(bit_time <= 14481))       //calculated 4x pulse width
        {
            baud_div_value = ((bit_time >> 5) - 1);
            Uart0Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
            Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
            Uart1Regs.UARTMBAUD.bit.BAUD_DIV_M = (baud_div_value >> 8);
            Uart1Regs.UARTLBAUD.bit.BAUD_DIV_L = (baud_div_value & 0xff);
            Uart0Regs.UARTCTRL3.bit.SW_RESET = 0;
            Uart0Regs.UARTCTRL3.bit.SW_RESET = 1;
            Uart1Regs.UARTCTRL3.bit.SW_RESET = 0;
            Uart1Regs.UARTCTRL3.bit.SW_RESET = 1;
        }
        else
        {
            //don't calculate baud rate
        }

}
```

The match baud function is called in the main function inside of the “forever” or “always” loop, so that it continually matches the baud rate. Ideally it should only be called when UART messaging is temporarily inactive as described in Section ???. This depends on the system, so there is a generic term in an if statement to represent this.

```
void main()
{
for(;;)
{
        if (erase_segment_counter > 0)
        {
            //Handle the DFlash segment erases
            erase_task();
        }
            //Run the PMBus handler
            pmbus_handler();
            //Transmit data out the UART
            uart_transmit_data();
            //Process data received from the UART
            uart_process_rx_data();
                //Match baud
                If(no_message_active)
                {
                    match_baud(pulse_width);
                }
            //output the baud rate from UART1 - used for debug purposes
            char_out_1(Uart0Regs.UARTLBAUD.bit.BAUD_DIV_L);


}
}
```

7 References

1. B. McDonald, H. Huang. “UCD3138 64 Pin Open Loop Board”. Texas Instruments. August 2012.
2. UCD3138 Monitoring and Communications Programmer’s Manual. (Literature Number: SLU996A) Link: http://www.ti.com/lit/pdf/sluu996 3.
3. S. Rajagopalan. Example Code.

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (October 2015) to A Revision	Page
• Revision A of this application report is changed to show the need for a UART reset after changing the baud rate.	7

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265