AN921: Configurable Logic Unit

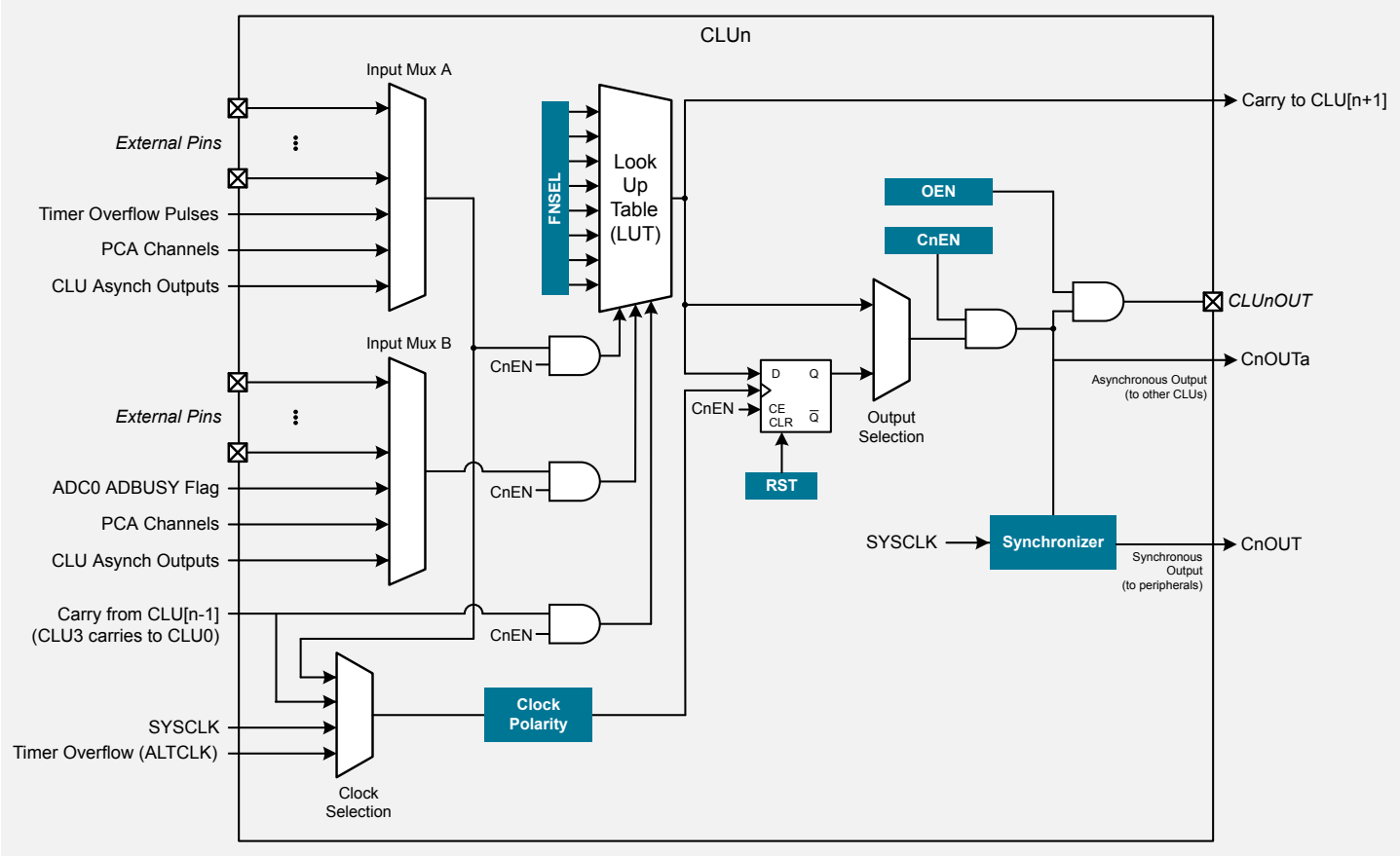
The EFM8LB1 and EFM8BB3 family of MCUs contain Configurable Logic Units (CLUs) that can be applied to applications that require some form of programmable logic.

This document demonstrates how to use CLUs to implement the following functions:

- SR latch
- D latch
- Button debounce
- Manchester encoder/decoder
- Biphase Mark encoder/decoder

KEY POINTS

- Configurable Logic operates without CPU intervention.
- Each unit supports 256 different combinatorial logic functions, such as AND, OR, XOR, and multiplexing.
- Multiple units combined can implement latches, encoders, and decoders.



1. Configurable Logic Overview

The configurable logic module provides multiple blocks of user-programmed digital logic that operate without CPU intervention. In the EFM8LB1 and EFM8BB3 families, the configurable logic (CL) module contains of four independent configurable logic units (CLUs) that support user-programmable asynchronous and synchronous Boolean logic operations. A number of internal and external signals may be used as inputs to each CLU, and the outputs may be routed out to port I/O pins or directly to select peripheral inputs.

Each CLU has a look up table (LUT) logic function that can be used to provide up to 256 different functions for 3 inputs – A and B inputs from 16-input multiplexers and a carry input from the LUT output of the previous CLU. The A and B input multiplexers can select port pins or the output of any CLU. Since there are many possible functions with 3 inputs, it can be quite challenging to determine the appropriate value to write to the LUT register (CLUnFN).

The LUT implements combinatorial Boolean logic, and there is a way to programmatically determine the value to write to CLUnFN using combination Boolean functions. The examples discussed in this document use a header file `SI_LUT.h` that contains macros and definitions to simplify the LUT initialization. For example, if we wish to implement LUT logic such as:

```
(A AND B) OR C
```

The C code would be:

```
CLU0FN = LUT_OR( LUT_AND(SI_LUT_A, SI_LUT_B), SI_LUT_C )
```

Alternatively, the look up table can be initialized using the Simplicity Studio Configurator.

Each CLU contains a D flip-flop, whose input is the LUT output. The D flip-flop clock is selected using the CLKSEL bitfield in the CLUnCF register. The D flip-flop can be bypassed by using setting the OUTSEL bit high in the CLUnCN register.

2. SR Latch

Configurable Logic Units can be used to build a SR latch. This section shows how this memory circuit can be built without using the D flip-flop portion of the CLU. This is advantageous in applications where we may want the output of the latch to be sent to another CLU via the carry input.

2.1 Background

The truth table for an SR latch is shown below. When the inputs SET = RESET = 0, the next output (Q_{NEXT}) will remain the same as the current output (Q). When SET = 0, RESET = 1, the next output will be reset to 0. When SET = 1, RESET = 0, the next output will be set to 1. When SET = RESET = 1, the next output is not defined.

Table 2.1. Truth Table of SR Latch (Simplified)

SET	RESET	Q	Q_{NEXT}	Comments
0	0	X	Q	Hold state
0	1	X	0	Reset
1	0	X	1	Set
1	1	X	Undefined	Undefined

2.2 SR Latch Implementation

To implement the SR latch, signals SET, RESET and Q must be assigned to CLU inputs and outputs. SET can be assigned to CLU1's MXA input. The current output Q can be assigned to CLU1's MXB input. The last available input to CLU1 is the Carry, which always the output of the previous CLU. Therefore, RESET is assigned to CLU0's MXA, and the CLU0 LUT implements a buffer, as shown below.

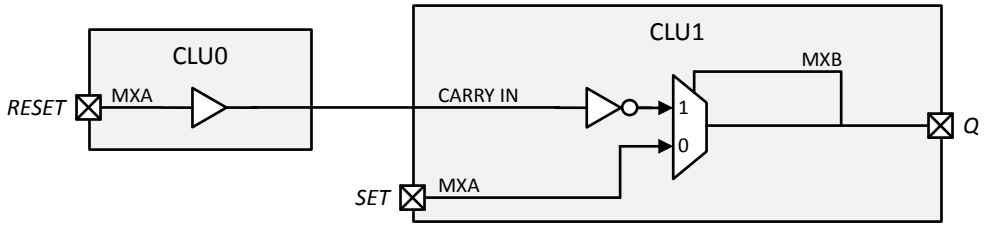


Figure 2.1. Block Diagram of SR Latch

The SR latch truth table can now be expanded as shown below. When the rows are ordered as shown, the Q_{NEXT} column read from top to bottom is the binary value, from most-significant bit to least-significant bit, written to the LUT register to implement the SR latch. Therefore, CLU1FN should be initialized to 0x74.

Table 2.2. Truth Table of SR Latch (Expanded)

SET (MXA)	Q (MXB)	RESET (CARRY IN)	Q_{NEXT}	Comments
1	1	1	0	*User defined
1	1	0	1	Set
1	0	1	1	*User defined
1	0	0	1	Set
0	1	1	0	Reset
0	1	0	1	Hold state
0	0	1	0	Reset
0	0	0	0	Hold state

Note: The CLU output cannot be undefined and must be either high or low. Therefore, when SET = RESET = 1, Q_{NEXT} has been arbitrarily defined. Alternatively, CLU1FN could be set to 0xF4, 0xD4 or 0x54.

2.3 Firmware Example

The SR latch firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Latches**]. The example uses CLU0 and CLU1 to implement the previously-discussed configuration for the SR latch.

3. D Latch

Configurable Logic Units can be used to build a D latch. This section shows how this memory circuit can be built without using the D flip-flop portion of the CLU. This is advantageous in applications where we may want the output of the latch to be sent to another CLU via the carry input.

3.1 Background

The truth table for a D latch is shown below. When the clock input (CLK) is not rising, the next output (Q_{NEXT}) will remain the same as the current output (Q). When the clock is rising and D input is 0, the next output is 0. When the clock is rising and the D input is 1, the next output is 1.

Table 3.1. Truth Table of D Latch (Simplified)

CLK	D	Q	Q_{NEXT}	Comments
Non-rising	X	X	Q	Hold state
Rising	0	X	0	Clock rising, D = 0
Rising	1	X	1	Clock rising, D = 1

3.2 D Latch Implementation

To implement the D latch, signals D, CLK and Q must be assigned to CLU inputs and outputs. CLK can be assigned to CLU3's MXA input. The current output Q can be assigned to CLU3's MXB input. The last available input to CLU3 is the Carry, which always the output of the previous CLU. Therefore, RESET is assigned to CLU2's MXA, and the CLU2 LUT implements a buffer, as shown below.

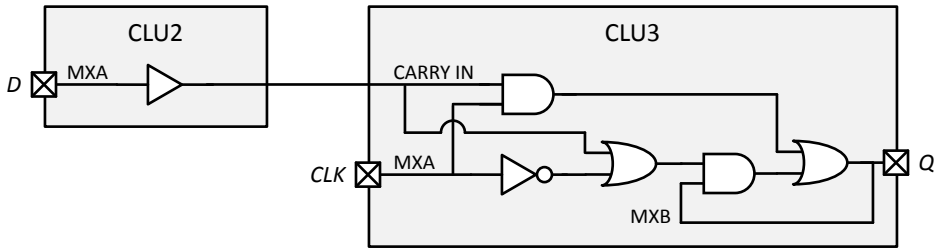


Figure 3.1. Block Diagram of D Latch

The D latch truth table can now be expanded as shown below. When the rows are ordered as shown, the Q_{NEXT} column read from top to bottom is the binary value, from most-significant bit to least-significant bit, written to the LUT register to implement the SR latch. Therefore, CLU1FN should be initialized to 0xAC.

Table 3.2. Truth Table of D Latch (Expanded)

CLK (MXA)	Q (MXB)	D (CARRY IN)	Q_{NEXT}	Comments
1	1	1	1	Clock rising, D = 1
1	1	0	0	Clock rising, D = 0
1	0	1	1	Clock rising, D = 1
1	0	0	0	Clock rising, D = 0
0	1	1	1	Hold state
0	1	0	1	Hold state
0	0	1	0	Hold state
0	0	0	0	Hold state

3.3 Firmware Example

The D latch firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Latches**]. The example uses CLU2, CLU3, and the previously-discussed configuration to implement the D latch.

4. Button Debounce

In this section, we demonstrate how to use the Configurable Logic Units and Timer to build an automatic push button debounce logic to automatically debounce button pushes and releases without any CPU or firmware intervention.

4.1 Background

Mechanical push buttons tend to generate multiple pulses when pressed or released, and some wireless IR signals may also tend to exhibit this behavior when there is a change in state. Hence, it is normally necessary for an embedded system to debounce such signals so that a single press or release does not appear like multiple presses or releases. Typically, this task is performed in firmware and can be prone to bugs if the firmware needs to handle other interrupts responsively. Button detection is normally a small part of the system and should deserve only a bit of attention from firmware. In this section, a debounced button is implemented without firmware resources or external hardware devoted to button debouncing.

In this implementation, a button is considered debounced if the output remains stable for specified amount of time. A timer keeps track of the stabilization time, and the timer resets whenever the button state changes. If the stabilization time is reached, a bit readable by firmware is updated to indicate the new, debounced button state. Firmware only needs to read the bit to determine to debounced button state. Alternatively, changes in the debounced button state can trigger a interrupt to alert firmware that a button has been pressed.

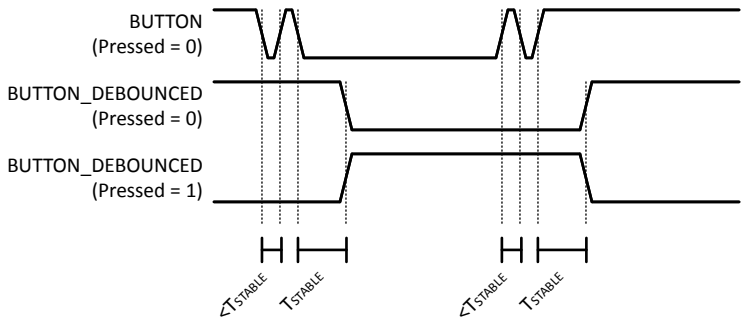


Figure 4.1. Timing Diagram of Debounced Button (Simplified)

4.2 Button Debounce Implementation

To implement the debounced button, two CLUs, one timer, and one GPIO pin is required. Timer 2 is used for a 10 ms stabilization time, and the stabilization time can be adjusted. The timing diagram can now be expanded to include Timer 2.

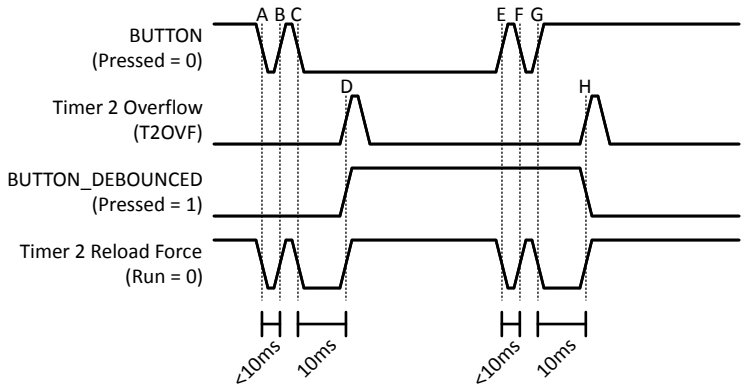


Figure 4.2. Timing Diagram of Debounced Button (Expanded)

The timing diagram shows the following events during a button press:

- A: Button press triggers Timer 2 to run, beginning the 10 ms timer.
- B: Button bounces high, triggering Timer 2 to stop, and DEBOUNCED_BUTTON is unchanged
- C: Button bounces low, triggering Timer 2 to run and beginning the 10 ms timer
- D: Timer 2 overflows indicating BUTTON has stabilized, triggering the update of the BUTTON_DEBOUNCED state, and Timer 2 stops

Similarly, the timing diagram shows the following events during a button release:

- E: Button release triggers Timer 2 to run, beginning the 10 ms timer.
- F: Button bounces low, triggering Timer 2 to stop, and DEBOUNCED_BUTTON is unchanged
- G: Button bounces high, triggering Timer 2 to run and beginning the 10 ms timer
- H: Timer 2 overflows indicating BUTTON has stabilized, triggering the update of BUTTON_DEBOUNCED state, and Timer 2 stops

The following block diagram implements the timing diagram above.

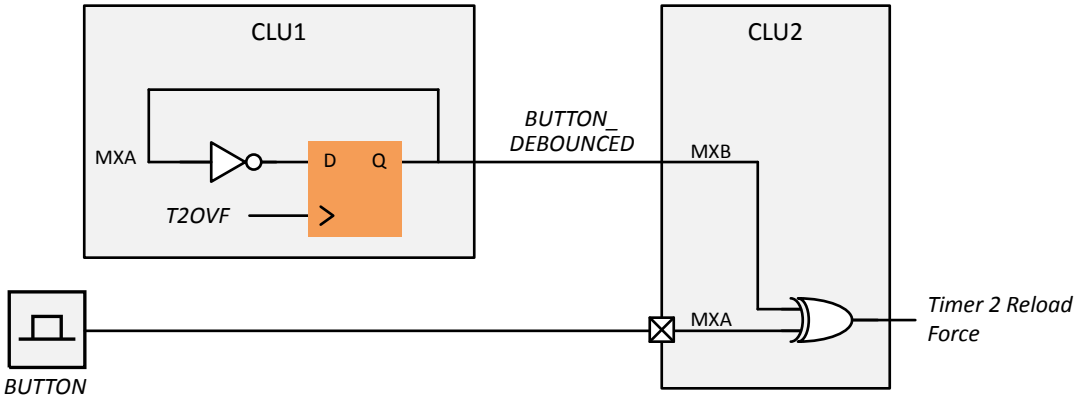


Figure 4.3. Block Diagram of Debounced Button

The BUTTON, Timer 2 overflow, BUTTON_DEBOUNCED, and Timer 2 Reload Force signals must be assigned to CLU inputs and outputs. BUTTON must be a GPIO pin, so it is assigned in CLU2's MXA input. Timer 2 can be reloaded by a CLU output, so it is assigned to the CLU2 output.

A CLU output rising or falling edge can trigger an interrupt, so BUTTON_DEBOUNCE is assigned to the CLU1 output. Since BUTTON_DEBOUNCED updates on the rising the Timer 2 overflow, the CLU1 D flip-flop is used, with the Timer 2 overflow as the clock and the inverted BUTTON_DEBOUNCE as the input. The CLU1 LUT implements the inversion with BUTTON_DEBOUNCE assigned to CLU1's MXA input.

The CLU2 LUT implements an XOR function, which allows Timer 2 to run only when there is a difference in logic between the BUTTON state and the BUTTON_DEBOUNCED state. When the debounce timer expires and BUTTON_DEBOUNCED is updated, Timer 2 Reload Force goes to logic high, continuously resetting Timer 2 and effectively stopping it.

4.3 Firmware Example

The button debounce firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Button Debounce**]. The example uses CLU1, CLU2, Timer 2, and the previously-discussed configuration to implement the button debounce.

The scope capture for the button press and release events are shown below in below, where the button signal (pressed = 0) is shown on CH2 and the debounced button state (pressed = 1) is shown on CH3.

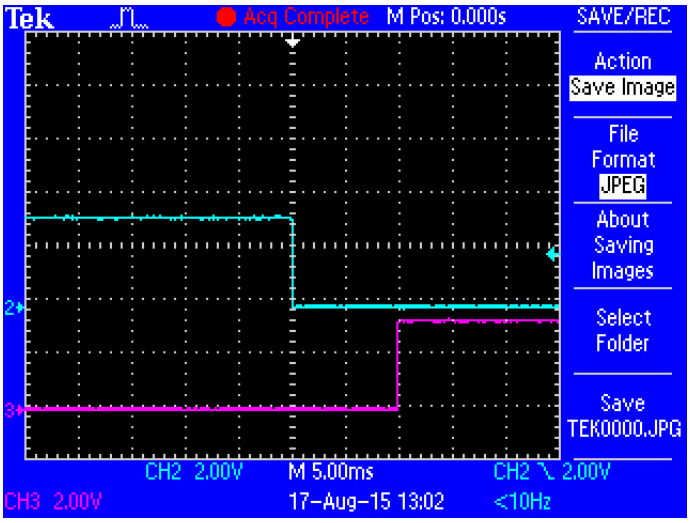


Figure 4.4. Scope Capture of Button Press Event

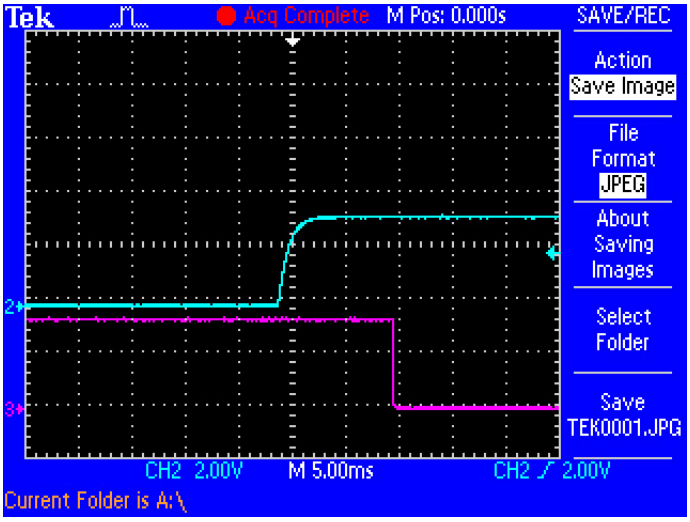


Figure 4.5. Scope Capture of Button Release Event

5. Manchester Encoder/Decoder

In this section, we demonstrate how to use the Configurable Logic Units, Timers, and SPI to build a Manchester encoder and decoder to transmit a Manchester-encoded signal. It requires very few CPU cycles to encode the input signal, as most of the work is performed by the CLUs and the SPI module.

5.1 Background

Manchester code is a type of line code that has constant DC component. There are two versions of Manchester code, with this document referring to the version defined in the IEEE 802.3 standard. Manchester code combines data and clock into a single signal, where one clock cycle is a Manchester bit period. A transition always occurs at in the middle of the bit period. DATA = 0 is represented by a falling edge (high-to-low transition) in the middle of the bit period, and DATA = 1 is represented by a rising edge (low-to-high transition) in the middle of the bit period. An example is shown below:

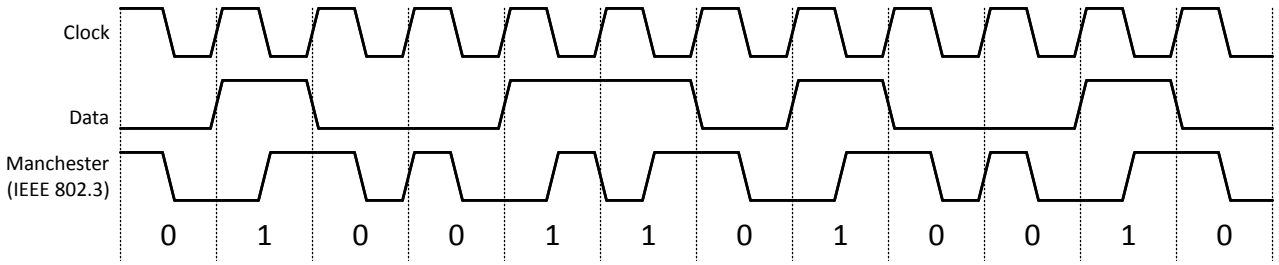


Figure 5.1. Manchester Encoded Data

5.2 Manchester Encoder

In this section, we will demonstrate how to use the CLUs and SPI to combine SPI data and clock signals into a single Manchester-encoded signal with little CPU intervention.

5.2.1 Manchester Encoder Implementation

The implementation of the Manchester encoder will encode the SPI output into a Manchester-encoded signal. The SPI module signals for SCK and MOSI are sent to the GPIOs via the priority crossbar decoder, and the CLUs will take these 2 pins as inputs to encode into Manchester code. The block diagram of the encoder logic is shown below:

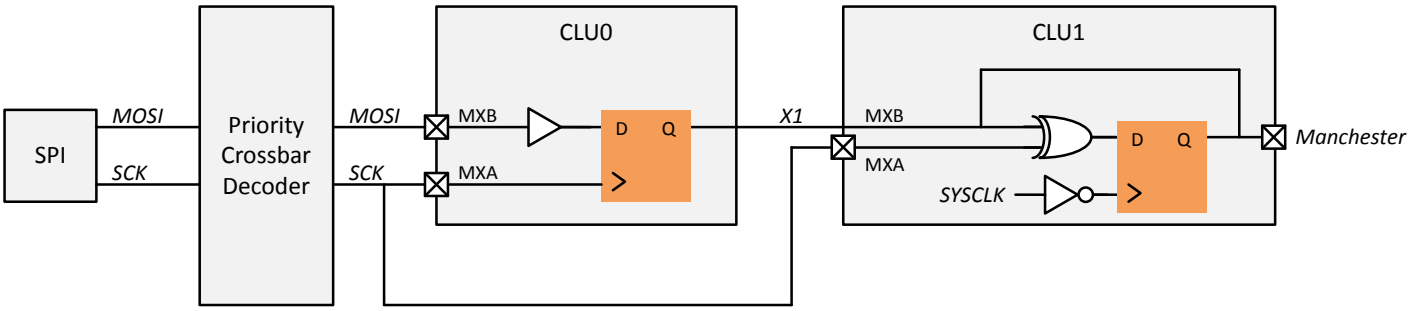


Figure 5.2. Manchester Encoder Block Diagram

The timing diagram of the Manchester encoded signal and the original SPI clock and MOSI data are shown below:

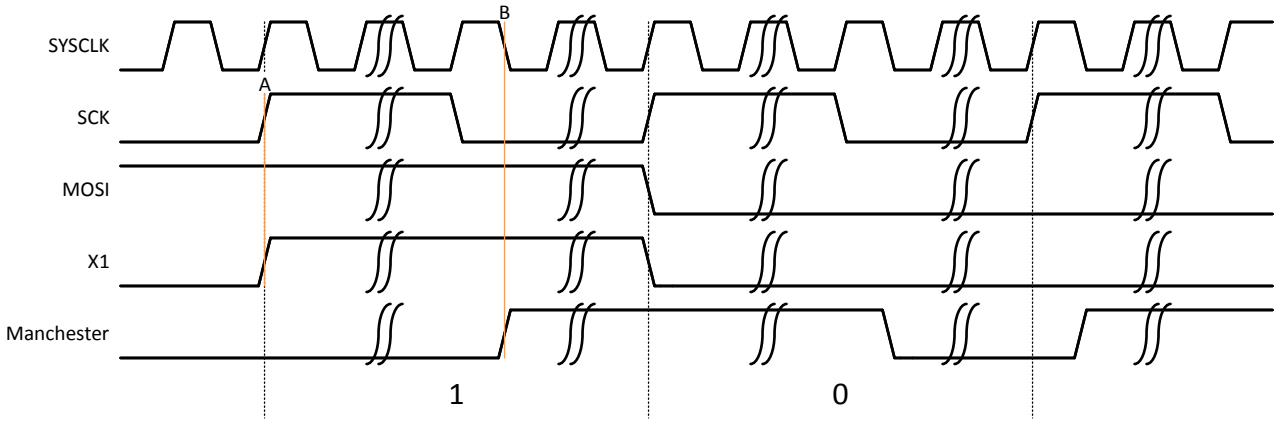


Figure 5.3. Timing Diagram of Manchester Encoder

The implementation is based on SPI configured with low idle state (CKPOL = 0) and data centered on the first edge (CKPHA = 0), the rising edge in the implementation. The implementation generates a Manchester code where the bit boundary is aligned with the rising edge of SCK delayed by a half SYSCLK cycle.

In the implementation, CLU0 is used to generate a data bit representation of MOSI that is valid between the rising edges of SCK (X1, event A). X1 is then XORed with the SCK in CLU0 to generate the Manchester coded signal. If the LUT output of CLU0 was used for the Manchester code, there would be glitches because the inputs of the XOR gate, SCK, and X1 transition on the same edge. Hence, this output is clocked by the inverted SYSCLK to avoid glitches. Due to the synchronous design of the SPI block, we know that the rising edge of SCK will be aligned to the rising edge of SYSCLK. Hence, we can avoid glitches by using the falling edge of SYSCLK to clock the XOR output (event B).

5.2.2 Firmware Example

The Manchester encoder firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Manchester Encoder**]. The example uses the SPI, CLU1, CLU2, and the previously-discussed configuration to implement the Manchester encoder.

The picture below shows the scope capture of the Manchester code, SCK and MOSI.

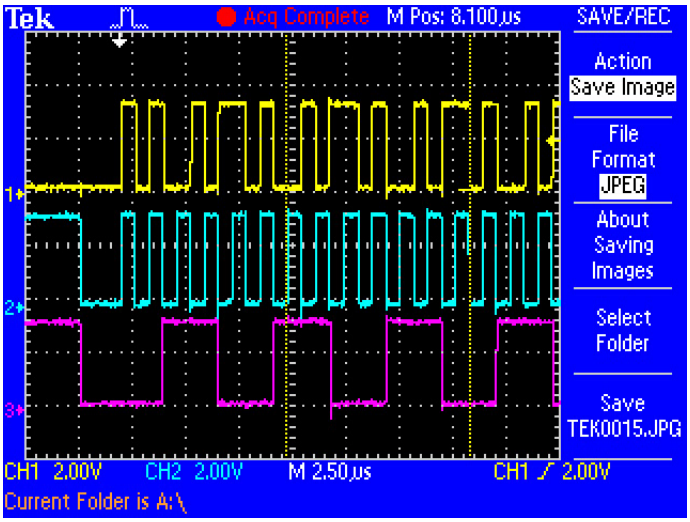


Figure 5.4. Scope Capture of Manchester Code (CH1), SCK (CH2) and MOSI (CH3) Signals

5.3 Manchester Decoder

In this section, we will demonstrate how to use the CLUs, Timers and SPI can be used to decode Manchester data with little CPU intervention.

5.3.1 Manchester Decoder Implementation

The implementation of the Manchester decoder will decode the single-wire signal into its clock and data, which can be fed into the SPI module. The SPI module allows its SCK slave clock and MOSI input to be taken from CLU outputs. In this implementation, the clock is re-synchronized at the end of every bit period. The block diagram of the decoder logic is shown below:

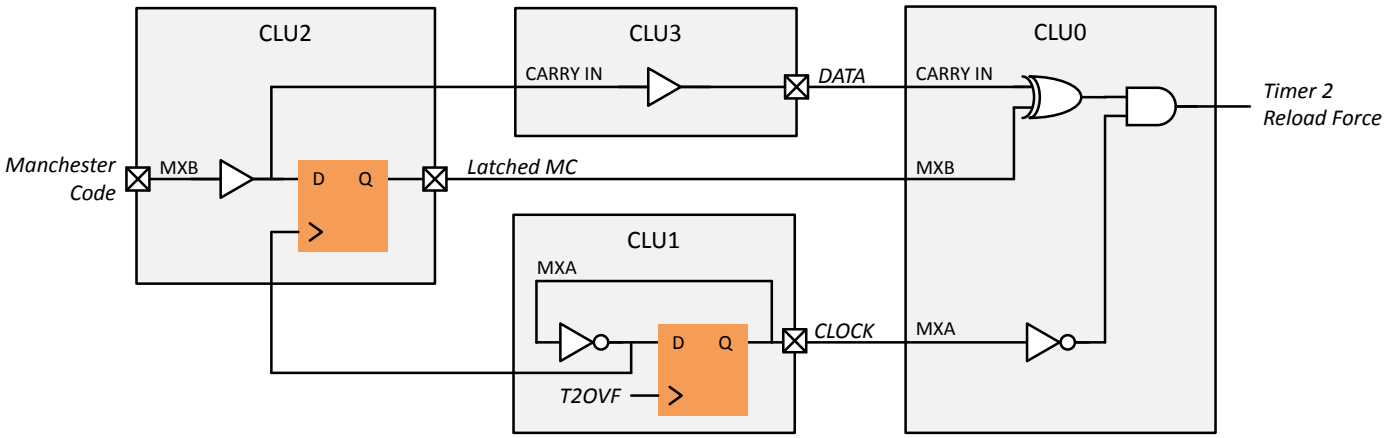


Figure 5.5. Manchester Decoder Block Diagram

The timing diagram of the Manchester encoded signal and the decoded SPI clock and data are shown below:

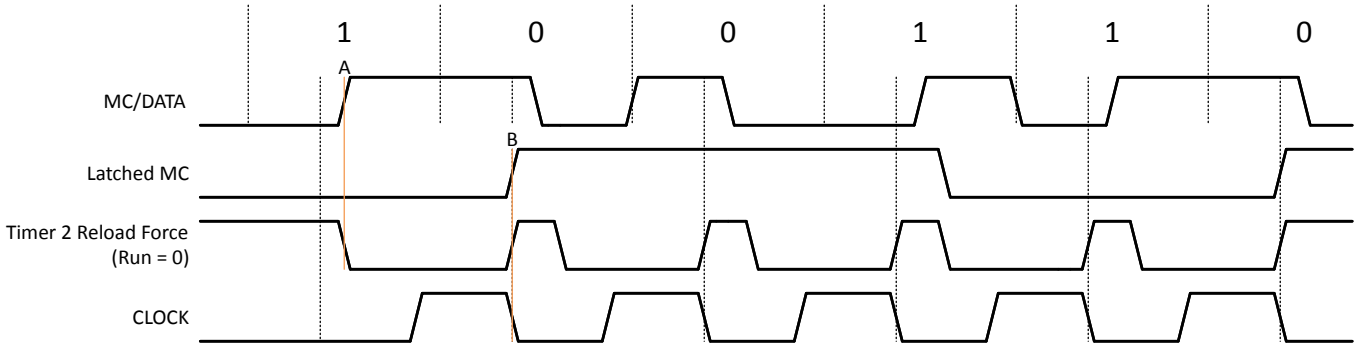


Figure 5.6. Timing Diagram of Manchester Decoder

The implementation is based on the observation that the data bit can be sampled in the second half of the Manchester code. A Manchester code always contains a transition in the middle of the bit period, which can be used to start a timer to generate the sampling edge. The Manchester code itself can be used as the DATA signal as long as we generate an SCK where the clocking edge is somewhere in the second half of the Manchester bit. In the figure above, CLU2 provides the DATA signal. The LUT logic merely passes the Manchester signal through.

Timer 2 is set to reload every 0.375 bit period. Its reload force select signal is configured to use the CLU0 output. CLU0 is designed such that when a mid-bit transition occurs (event A), its output will go low, starting Timer 2. After the first Timer 2 overflow, CLU1 generates the rising edge of SCK, sampling the Manchester signal. When CLOCK is high, it forces the Timer 2 reload force signal to remain low. When the second Timer 2 overflow occurs, CLU1 generates the falling edge of CLOCK. The falling edge of CLOCK causes two events: CLU2 will sample the MOSI input, and Timer 2 reload force signal will go high, resetting Timer 2 and freezing it until the next mid-bit transition (event B).

The SPI SCK and MOSI inputs are connected directly to the CLU outputs CLOCK and DATA respectly. CLOCK and DATA are not routed to GPIO pins to conserve GPIO usage.

5.3.2 Firmware Example

The Manchester decoder firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Manchester Decoder**]. The example uses the CLU0, CLU1, CLU2, CLU3, Timer 2, and the previously-discussed configuration to implement the Manchester decoder.

The picture below shows the scope capture of the CLOCK and the Manchester code/DATA signal.

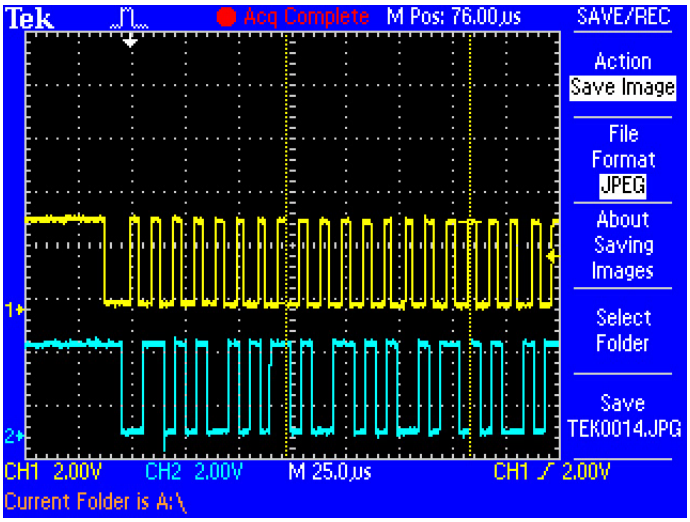


Figure 5.7. Scope Capture of CLOCK (CH1) and Manchester/DATA (CH2)

6. Biphase Mark Encoder/Decoder

6.1 Background

Biphase Mark Code (BMC) combines both data and clock in a single signal. One clock cycle is a BMC bit period. A transition always occurs at the beginning of each bit period. A logic 1 is represented by a transition (rising or falling edge) in the middle of the bit period. A logic 0 is represented by no transition in the middle of the period. An example is shown below:

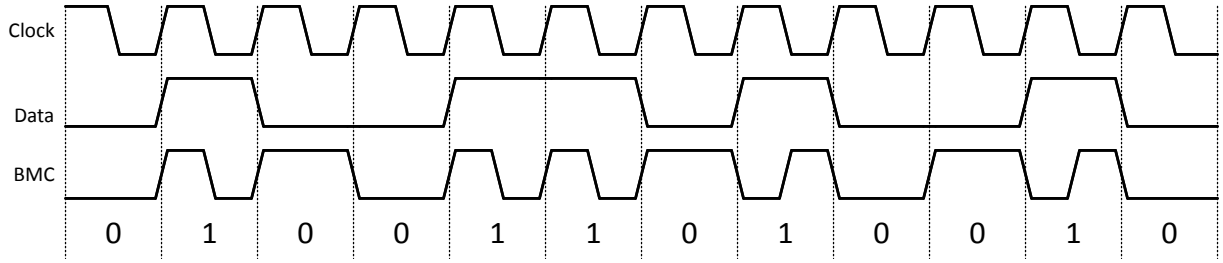


Figure 6.1. Biphase Mark Encoded Data

A BMC encoder accepts a data signal and clock signal as inputs and produces a single BMC encoded output. A BMC decoder accepts a BMC-encoded signal as the input and produces two outputs: data and clock.

BMC is used in standards such as the USB 3.1 Power Delivery Specification CC signaling, AES3 digital audio, or S/PDIF audio.

6.2 Biphase Mark Encoder

In this section, we will demonstrate how to use the CLUs, Timers, and SPI to combine SPI data and clock signals into a single BMC-encoded signal with little CPU intervention. The example will transmit the BMC signal at 300 kbps with the system clock operating at 24.5 MHz. This section also demonstrates how to transmit a non-divisible-by-8 number of bits.

6.2.1 Biphase Mark Encoder Implementation

This implementation also demonstrates how to transmit only a non-divisible-by-8 number of bits using a Timer and CLU without CPU intervention. The block diagram of the encoder logic is shown below:

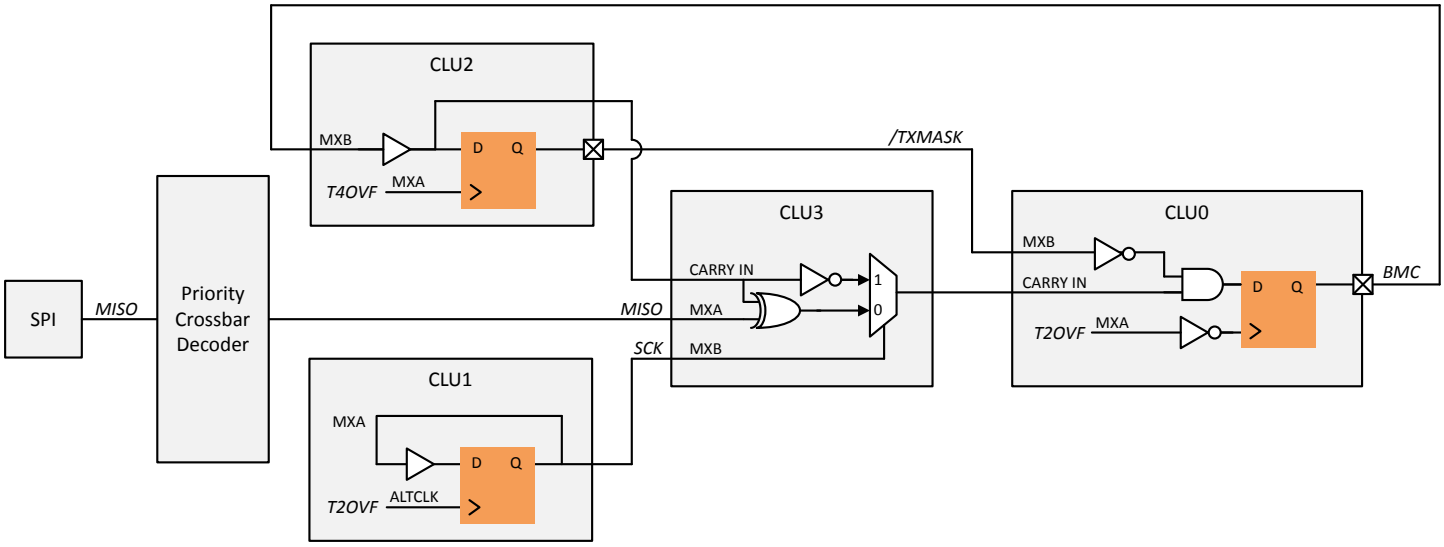


Figure 6.2. BMC Encoder Block Diagram

The design ensures that the last BMC transition is a falling edge. The timing diagram of the BMC encoded signal and the MISO data are shown below:

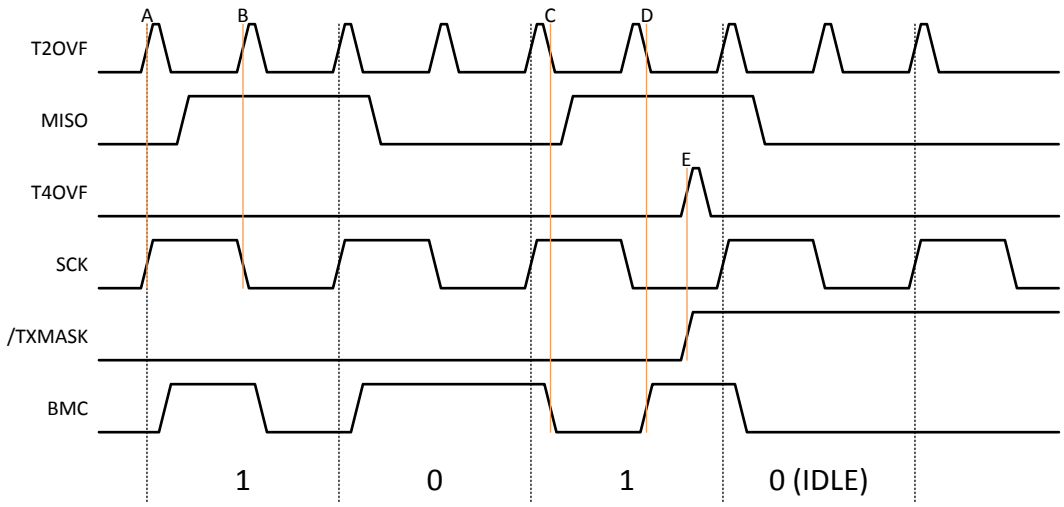


Figure 6.3. Timing Diagram of BMC Encoder - Last BMC Transition is a Falling Edge

CLU1 generates the SPI slave clock, and the MISO output is sent to the CLU3 via the priority crossbar. The SPI is configured with low idle state (CKPOL = 0) and data centered on the second edge (CKPHA = 1), the falling edge. The rising edge of SCK corresponds the BMC bit boundary (event A and event B in the Timing Diagram). A BMC signal always has a transition at every bit boundary (event C). This is achieved via CLU3 selecting the inverse of the original BMC output. When the falling edge of SCK occurs, CLU3 selects the XOR of the MISO and the current BMC output; this inverts the BMC output if the MISO bit is 1. When the next falling edge of Timer 2 overflow occurs in the middle of the bit period, this is clocked into the BMC output (event D).

The CLU2 output generates the mask used to force the BMC output to low (event E). This design ensures that BMC transmission, with a non-divisible-by 8 number of bits, stops automatically without CPU intervention. Firmware detects the end of transmission by sensing for the CLU2 output rising edge. CLU0 clocks in the BMC output one clock cycle later because it uses the inverted Timer 2 overflow, which is also used to generate the SPI SCK slave clock. The rising edge of the Timer 2 overflow clocks the transitions of many signals, so the inverted clock is used in CLU0 to prevent glitches in the BMC output.

The timing diagram for the case where last BMC transition is a rising edge is shown below. If the last data bit results in a BMC rising edge, the termination event is delayed to the next bit (event F). As long as the user ensures the subsequent MISO bit is a zero bit, the final falling edge transition will be generated.

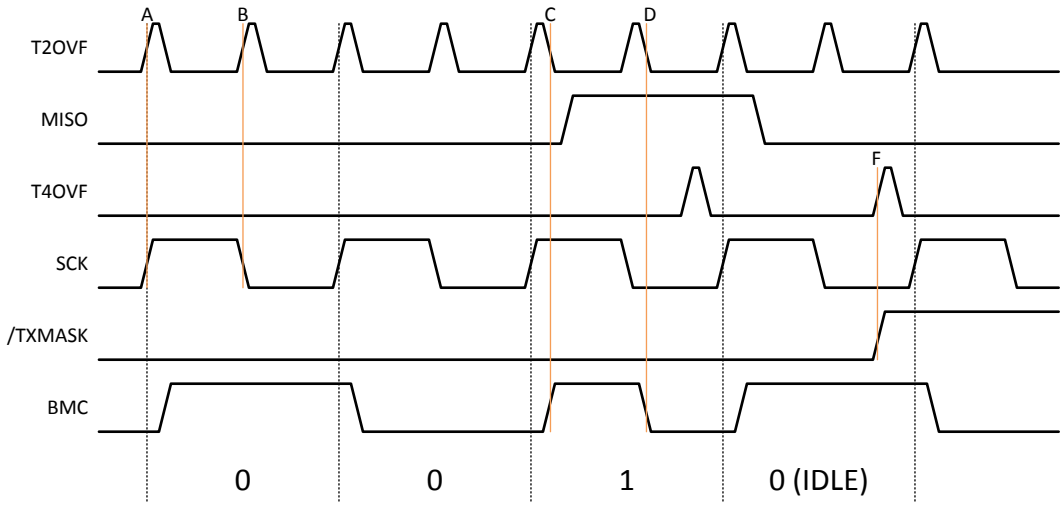


Figure 6.4. Timing Diagram of BMC Encoder - Last BMC Transition is a Rising Edge

Timer 2 is setup to overflow at 600 kHz to generate the 300 kbps SCK clock. Timer 4 is initialized to use SYSCLK/12 as its clock, and its timer is initialized to overflow after the required number of bits are sampled:

```
TMR4 = -(((uint32_t)(NUM_BITS * 2) * (uint16_t)(-TMR2RL) - 1) / 12);
TMR4RL = -((((uint32_t)(NUM_BITS * 2 + 2) * (uint16_t)(-TMR2RL) - 1) / 12 + TMR4);
```

The Timer 4 reload register is then set to a value such that it will overflow again one SPI clock later, after accounting for the rounding effects of the divide-by-12. The second overflow event, if it occurs, ensures that the last transition is always a falling edge. Timer 4 will be blocked from counting further because the CLU2 output has been assigned to the reload force select (RLFSEL) of the Timer. When a successful falling transition can be generated (when BMC is sampled high), Timer 4 will stop counting.

6.2.2 Firmware Example

The Biphase Mark encoder firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Biphase Mark Encoder**]. The example uses the SPI, CLU1, CLU2, CLU3, CLU4, Timer 2, Timer 4, and the previously-discussed configuration to implement the Biphase Mark encoder. The example transmits 12 bits of BMC encoded bits.

The picture below shows the scope capture of the SCK, /TXMASK and BMC.

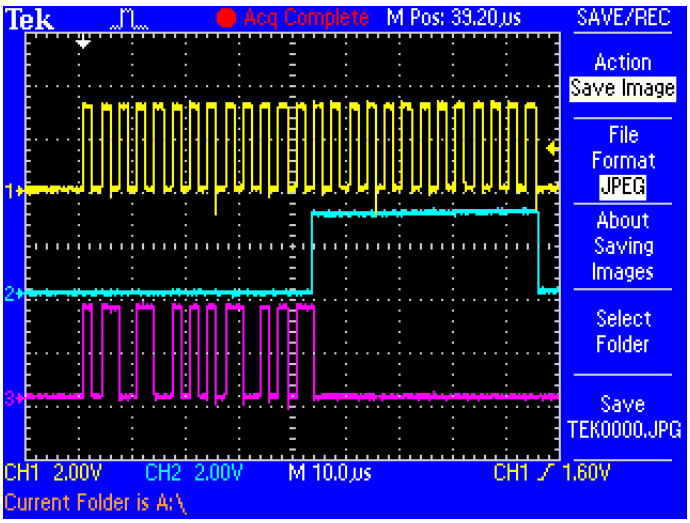


Figure 6.5. Scope Capture of SCK (CH1), /TXMASK (CH2) and BMC (CH3) signals

In the demo application, the firmware intentionally delays the servicing of the completion of transmission to demonstrate that there is no requirement to service this event within a strict time limit. In the above scope capture, we can observe the extra bit generated because the last data bit generated a rising edge transition.

6.3 Biphase Mark Decoder

In this section, we will demonstrate how to use the CLUs, Timers, and SPI to decode the data with little CPU intervention. The example will receive a BMC-encoded signal at 300 kbps with the system clock operating at 24.5 MHz. This section also demonstrates how to use the CLUs and Timers to perform clock recovery of the signal.

6.3.1 Biphase Mark Decoder Implementation

The implementation of the Biphase Mark decoder will decode a single BMC-encoded signal into separate clock and data signals, which can be fed into the SPI module for reception. The SPI module allows its SCK slave clock and MOSI input to be taken from CLU outputs. In this implementation, we also demonstrate how to resynchronize the clock at every bit transition that occurs at the end of every bit. The block diagram of the decoder logic is shown below:

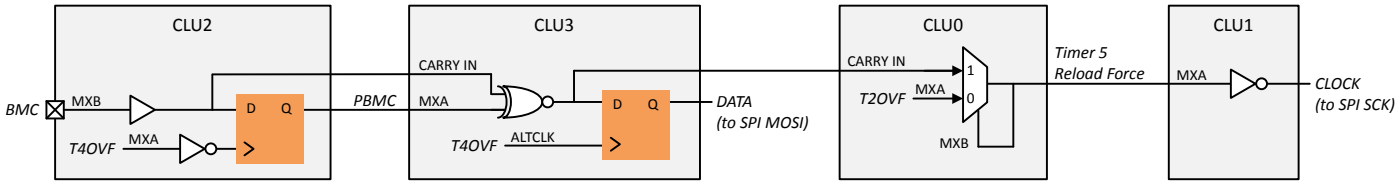


Figure 6.6. BMC Decoder Block Diagram

The timing diagram of the BMC encoded signal and the decoded SPI clock and data is shown below:

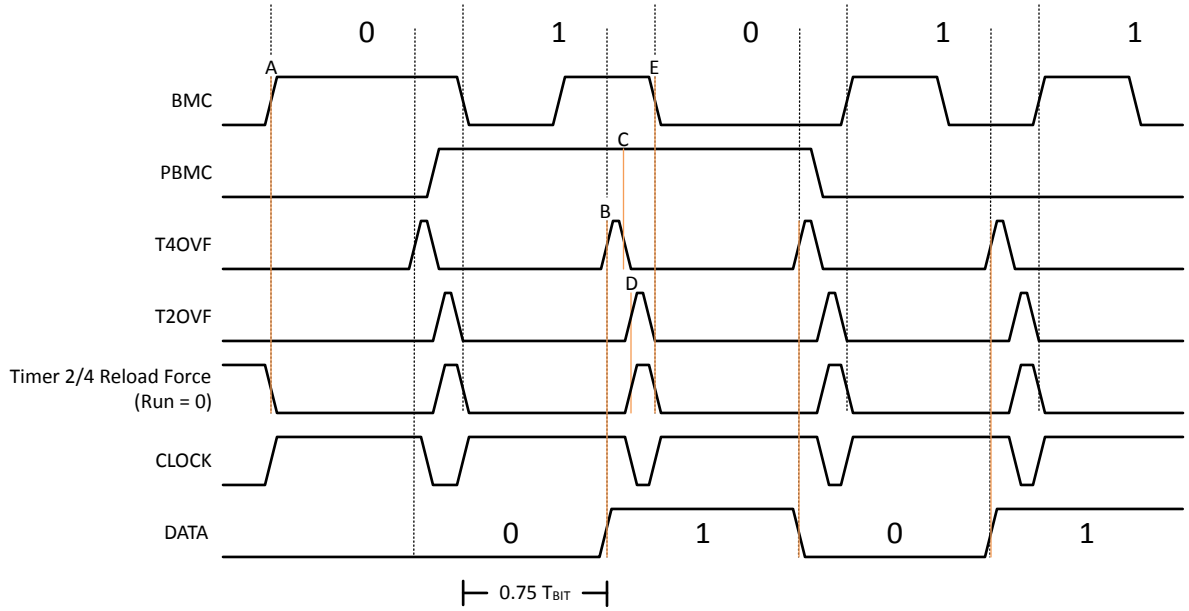


Figure 6.7. Timing Diagram of BMC Decoder

The implementation is based on the observation that comparing consecutive samples of the BMC signal at 0.75 bit period away from the bit boundary can determine the value of the data bit that is transmitted. For example, if the BMC state is 1 at 0.75 bit period away from the first bit boundary, then it can be determined that the BMC state will be 0 at the second bit boundary. If the BMC is encoded for DATA = 0 in the second bit period, a transition will not occur at 0.5 bit period after the second bit boundary; hence, the sample at 0.75 bit period after the second bit boundary will still be 0. If the BMC is encoded for DATA = 1 in the second bit period, a transition will occur at 0.5 bit period after the second bit boundary; hence, the sample at 0.75 bit period after the second bit boundary will be 1. Therefore, DATA can be implemented by an XNOR Boolean function in CLU3 that takes the current BMC signal and the prior BMC (PBMC) signal sampled 1 bit period earlier as its inputs.

The Timer 4 overflow is setup to overflow at 0.75 bit period after every bit boundary. The rising edge of the Timer 4 overflow signal will latch the output of the XNOR result out to the DATA (event B); then, its falling edge 1 SYSCLK cycle later will latch the current logic level of the current BMC into PBMC via CLU1 (event C). After this clocked PBMC event occurs, the output of the XNOR will be 1 because the current BMC and the PBMC should be the same logic level. Now, when the Timer 2 overflow occurs another clock cycle later (event G), it will cause the multiplexer in CLU0 to select the high input, which is the output of the XNOR result. Timers 2 and 4 have both selected the output of CLU0 to be the force reload signal. As the force reload is now high, this causes Timer 4 and Timer 2 to enter a reload state to continously reload the 0.75 bit period and 0.75 bit period + 2 sytem clocks respectively into the respective Timer registers, effectively stopping both counters.

The inputs of the XNOR are the BMC and the latched PBMC of the current bit period. When the current bit period is over, there is a guaranteed transition in the BMC (event E). This will cause the XNOR output to change to 0, which will cause the CLU0 carry in to transition to logic 0, forcing the selection to the Timer 2 overflow signal, which is also at logic 0. As the output of CLU0 is now 0, Timers

2 and 4 will restart counting with the reloaded values. This mechanism effectively causes re-synchronization at every bit boundary. This is desirable because it means that the clocks at both transmit and receive devices need not be exactly the same as the other clock.

Meanwhile, the Timer reload force signal from the output of CLU0 is inverted by CLU1, and the output of CLU1 is the CLOCK with the rising edge as the clocking edge. The CLOCK signal is also used for the Timer 5 reload force, where Timer 5 is used to detect if the BMC line has gone idle. If a Timer 5 overflow event occurs, it means that the BMC signal is inactive and no data is arriving. This is illustrated in the timing diagram below.

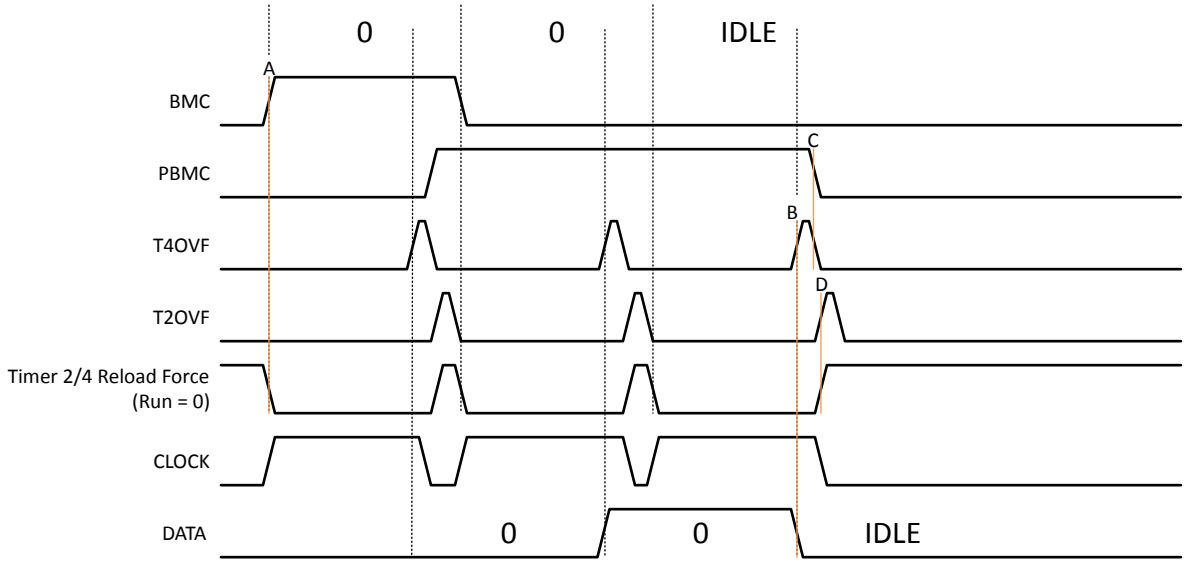


Figure 6.8. Timing Diagram of BMC Going Idle

The reason for setting the SPI clock to stay low idle is because there are no more BMC transitions when the line is idle. This means that the falling edge event on Timer 2/4 reload force (event E in Figure 6.7 Timing Diagram of BMC Decoder on page 20) will not occur. This effectively stops Timers 2 and 4 while allowing Timer 5 to eventually overflow because the SPI clock will no longer return to high to force reload Timer 5.

The SPI SCK and MOSI inputs are connected directly to the CLU outputs CLOCK and DATA respectly. CLOCK and DATA are not routed to GPIO pins to conserve GPIO usage.

6.3.2 Firmware Example

The Biphase Mark decoder firmware example can be found in Simplicity Studio under [**Software Examples**]>[**Kit: EFM8LB1/ EFM8BB3 Starter Kit**]>[**Configurable Logic**]>[**Biphase Mark Decoder**]. The example uses the CLU0, CLU1, CLU2, CLU3, Timer 2, Timer 4, Timer 5, and the previously-discussed configuration to implement the Biphase Mark decoder.

The picture below shows the scope capture of the BMC, CLOCK (SPI SCK), and DATA (SPI MOSI). The first rising edge of CLOCK is ignored as the SPI is not yet enabled until after the first transition of the BMC signal.

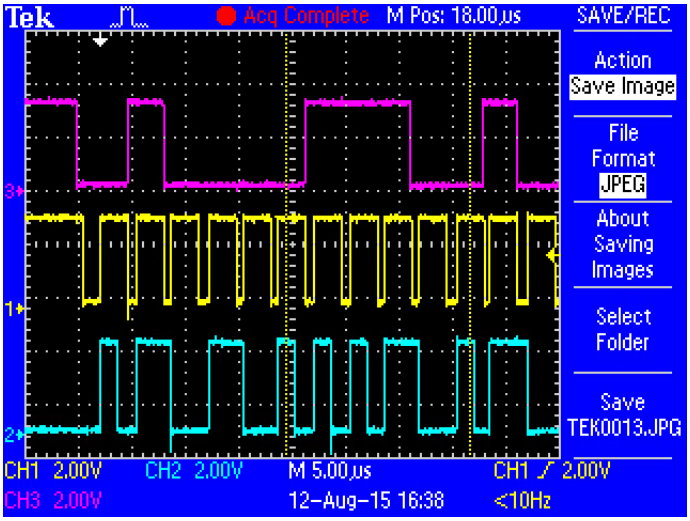


Figure 6.9. Scope Capture of BMC (CH3), CLOCK (CH1) and DATA (CH2) Signals

Simplicity Studio

One-click access to MCU tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

www.silabs.com/simplicity

MCU Portfolio
www.silabs.com/mcu

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com