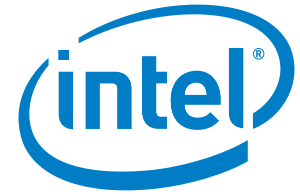
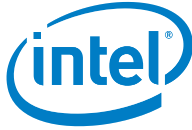


Maximum Likelihood Manchester Decoding on the Intel® Quark™ microcontroller D1000

White Paper

October 2015

Document Number: 332919–001US



You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

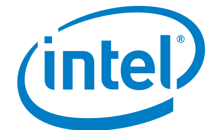
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting: http://www.intel.com/design/literature.htm

Intel, Intel® Quark™ and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2015, Intel Corporation. All rights reserved.



Contents

1 Abstract 5

2 Introduction 6

3 Methods 7

3.1 Sampling 7
3.2 Likelihood Lookup 9
3.3 Phase Tracking 11
3.4 Synchronization 11

4 Results 13

5 Discussion 19

6 Conclusion 20

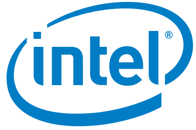
Appendix A – Maximum Likelihood Decoding Program 21

Appendix B –Likelihood Lookup Table Generator Program 26

Appendix C – Manchester-encoded Data Waveform Generator Program 28

Figures

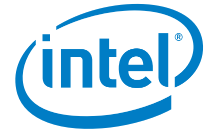
Figure 1. Waveforms of NRZ Data and Clock and the Corresponding Manchester-encoded Data 6
Figure 2. Tracking Symbol Phases 9
Figure 3. Sample Phases 10
Figure 4. Ten Phase Shifts of a Ten Sample Symbol Window 12
Figure 5. Start of Manchester Encoded Message 14
Figure 6. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing +357 ns Duty Cycle Distortion 15
Figure 7. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing -357 ns Duty Cycle Distortion 16
Figure 8. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing -3.0 % Bit Rate Offset 17
Figure 9. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing +3.5 % Bit Rate Offset 18



Revision History

Date	Revision	Description
October 2015	001	Initial release.

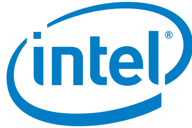
§



1 Abstract

A common use of low power microcontrollers is in RFID tags. One of the modulation schemes typically used by RFID tags is Manchester encoding. Manchester encoding is a form of binary phase shift key modulation. This paper presents a maximum likelihood method of decoding Manchester-encoded data. Maximum likelihood decoding tracks bit rate offset, tolerates duty cycle distortion, and is more robust in the presence of noise than edge decoding.

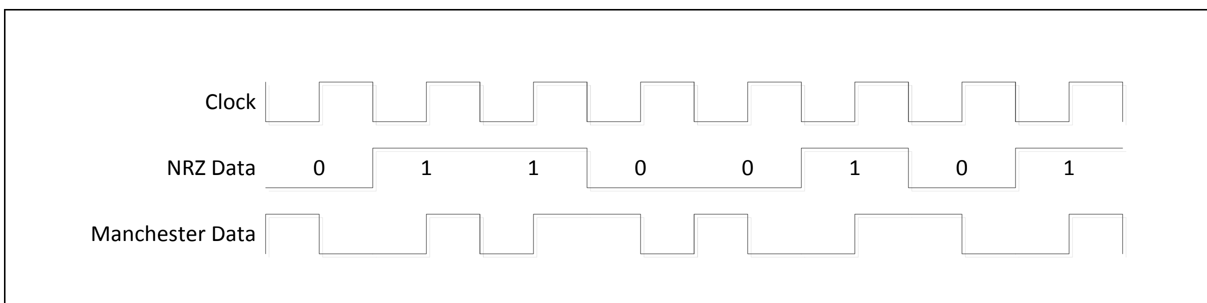
§



2 Introduction

Manchester encoding is a form of binary phase shift key modulation where the carrier is the bit rate clock (a square wave) phase shifted by zero or 180° depending on the data. Manchester-encoded waveforms are shown in Figure 1. In the Manchester data waveform, a zero bit is encoded with a falling edge at bit center and a one bit with a rising edge at bit center. The opposite may also be used.

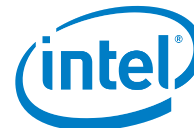
Figure 1. Waveforms of NRZ Data and Clock and the Corresponding Manchester-encoded Data



The ideal undistorted waveform that the application receives for each bit should be high for one half of the bit period and low for the other half. Which half is high and which half is low depends on the state of the transmitted bit. In reality, the received waveform might be noisy or distorted. The relative bit rate between transmitter and receiver may be slightly off. The task of maximum likelihood decoding is to choose the most likely transmitted bits given the waveform received. For this, the application uses Bayes theorem:

$$\mathrm{P}(x \mid y) = \mathrm{P}(y \mid x)\frac{\mathrm{P}(x)}{\mathrm{P}(y)}$$

…where *x* was received and *y* was transmitted. This is a very powerful theorem. It means that maximum likelihood decoding estimates the bits most likely transmitted given the waveform actually received and prior knowledge about the system. If the application has some prior knowledge about the probability of a one versus a zero being transmitted and what the ideal received waveform should look like for each, the application can build a table ahead of time containing the most likely transmitted bit for each possible received waveform. The application can then index the table using the actual received waveform to obtain the most likely transmitted bits.



3 Methods

This section presents the methods used to implement the maximum likelihood Manchester decoder. The requirements for the decoder are:

- Bit rate: 500 kbps ±3 %
- Duty cycle distortion: ±350 ns

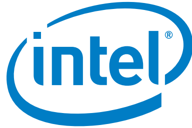
3.1 Sampling

There are multiple options for using the Intel® Quark™ microcontroller D1000to sample the received waveform:

- Comparator – this technique has the advantage of accepting analog waveforms. However, software must either poll the comparator or take an interrupt when the comparator detects a threshold crossing. In either case, it will not be possible to decode high bit rates in real time since there is no elasticity in the hardware, and software must be ready for each transition ahead of time.
- ADC – this technique also has the advantage of accepting analog waveforms. Moreover, it produces M-ary samples where $M = 2^N$ and N could be from 6 to 12 bits of resolution. This could prove extremely powerful if the application has some prior knowledge about the linear system (i.e. a linear model of the channel and receiver). However, once again it will not be possible to decode high bit rates in real time due to the large number of CPU cycles required to do this type of processing vs. the number of CPU cycles available in each bit period.
- SPI – this technique has the disadvantage of requiring a digital input. However, it has the advantage of elasticity provided by a FIFO implemented in hardware; up to 128 binary samples can be buffered in the FIFO. Software can then fall behind in the short term so long as it can keep up with the overall sample rate.

Due to the high bit rate, the comparator and ADC sample methods are off the table. This application will use SPI to sample the incoming signal, which frees up CPU cycles while the hardware FIFO is buffering samples. The next item is to decide on a sample rate (i.e. the number of samples taken in each bit period). Rather than computing likelihood in real time, which requires a larger number of CPU cycles, this application pre-computes the likelihood and stores it in a lookup table. This application uses the waveform samples to index the table. The group of samples used to index the table is commonly referred to as a symbol. A symbol contains a group of samples spanning one or more bit periods. If the symbol spans one bit period, the decision about what was transmitted will be a binary decision (for example, one of two states). If the symbol spans two bit periods, the decision will be 4-ary (for example, one of four states). If the symbol spans three bit periods, the decision will be 8-ary and so forth

The advantage of using symbols that span multiple bits is that the number of CPU cycles available to process each symbol increases with the number of bits spanned. Since decisions based on likelihood for each possible symbol have been pre-computed, the cycles required to process a symbol are independent on the number of bit periods spanned, and it is to the application’s advantage work with multi-bit symbols.



However, the amount of memory available to hold the look-up table is limited, so the application needs to constrain the number of bits in a symbol. For best performance, the application should locate the table in SRAM. This puts a practical constraint on the size of the table at less than 4 kB[1] and limits the symbol size to less than 12 bits.

As shown later, the question of likelihood also bears on the choice of sample rate. In order to assess likelihood, it's necessary to measure the distance between all possible symbols and the most likely symbols. The most likely symbols are those corresponding to possible transmitted states (for example, 0 or 1 for binary symbols; 00, 01, 10 or 11 for 4-ary symbols; etc.) The greater the distance between the most likely symbols, the lower the probability of a received symbol having equal distance to more than one of the most likely symbols. In other words, there will be fewer ties when making decisions about which bits were transmitted. If the number of possible symbols is large (for example, the number of samples per symbol is large) and the number of possible transmitted states is few, the distance between the most likely symbols will be large. This condition corresponds to a high sample rate.

Another reason for using a high sample rate is to correct for bit rate offset by tracking symbol phase. In order to track symbol phase, the application needs to compare the likelihood of a received symbol with one shifted by a sample. The application then adjusts the symbol window toward the more likely symbol – phase shifted or not. If the number of samples per bit is too small, it will be impossible to distinguish between phase error and a different transmitted state. This condition is illustrated in Figure 2 (a). Increasing the sample rate remedies the condition as shown in Figure 2 (b).

Ultimately, the application needs to balance the choice of sample rate to the competing goals of maximizing CPU cycles per symbol, maximizing distance between most likely symbols, and constraining the size of the look-up table. The optimal balance depends on bit rate. The application needs to budget about 100 CPU cycles per symbol. At a bit rate of 500 kbps and CPU clock frequency of 32 MHz, 10-bit 4-ary symbols strike a good balance at 2 bits per symbol, 5 samples per encoded bit, and a 1024 entry lookup table.

The SPI sample rate is programmable in even divisions of the CPU clock frequency. The application needs the following sample rate:

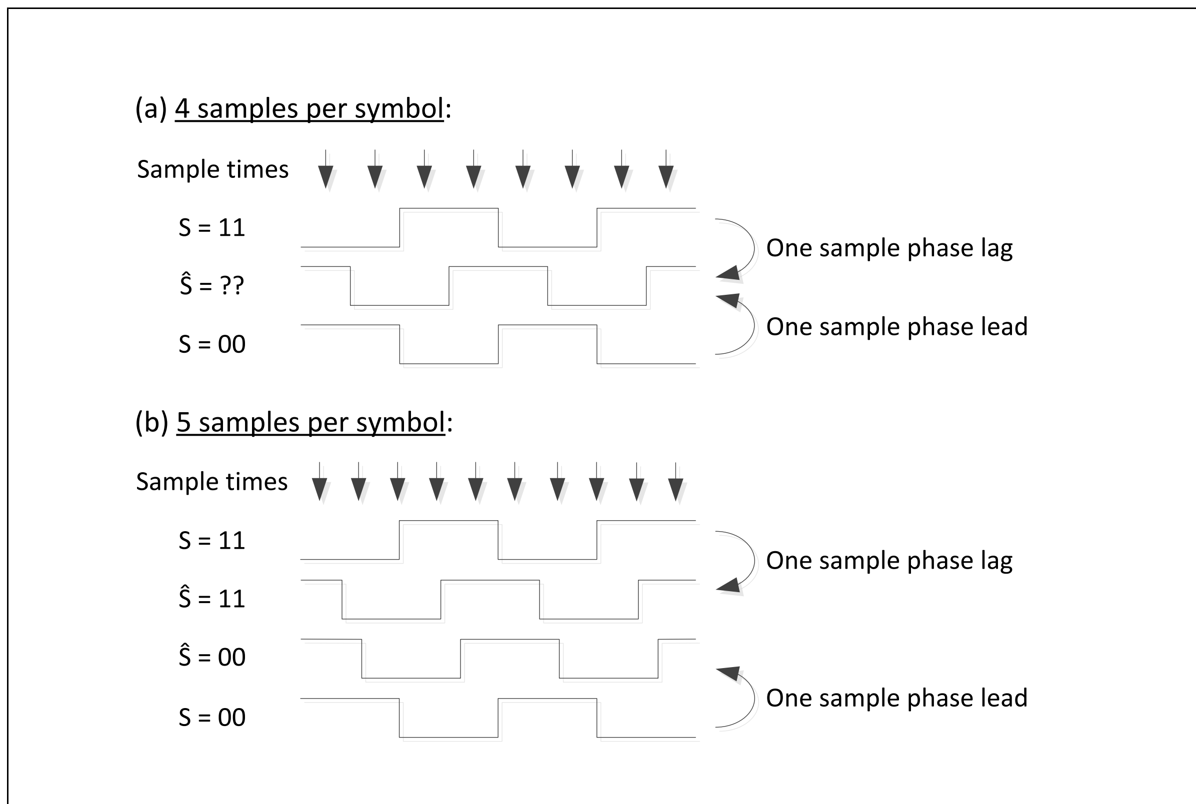
$$f_S \geq \rho f_B$$

…where ρ equals the minimum number of samples per bit and f_B is the bit rate. At five samples per bit and a 500 kbps bit rate, the application needs a sample rate of at least 2.5 MSps. A divisor of 12 yields a sample rate of 2.667 MSps. This produces 5 ⅓ samples per bit or 10 ⅔ samples per 4-ary symbol.

[1] While there is 8 kB of SRAM available, reprogramming flash can require up to 4 kB of buffer allocated on the stack. Also, less than 4 kB of data flash is available for storing initialized data.

As shown in Figure 2, at four samples per bit, a binary 1 waveform is shown that lags by one sample cannot be distinguished from a binary 0 waveform that leads by one sample. At five samples per bit, a waveform that lags or leads by one sample is closer to its transmitted waveform than it is to the alternate waveform.

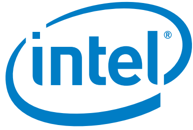
Figure 2. Tracking Symbol Phases



3.2 Likelihood Lookup

Ideal models for the 4-ary symbols are shown in Figure 3. The middle model for each 4-ary state (i.e. 00, 01, 10 and 11) corresponds to perfect phase alignment. Since phase can be adjusted only in discrete one sample increments, symbols that lead or lag in phase by less than ½ sample are equally likely and should be given the same weight as those with perfect alignment. These leading and lagging models are shown below and above the models with perfect alignment. Symbols that lead or lag by more than ½ sample can be brought closer to perfect alignment by shifting phase one or more samples. These must be weighted less than ideal symbols. The weight that the application assigns to each possible symbol from binary 0000000000 to 1111111111 should reflect how closely correlated it is to the most closely correlated ideal model. Logically, correlation can be expressed as the number of matching samples:

$$W_S = \sum_{n=0}^{9} \overline{R[n] \oplus Y_S[n]}$$



...where W_S is the correlation strength or weight corresponding to ideal symbol model for transmit state S, $R[n]$ is the nth sample of the received symbol, $Y_S[n]$ is the nth sample of the ideal symbol model for transmit state S, and $\oplus$ is the exclusive or operator.

As shown in Figure 3, at 10 ⅔ samples per symbol, each of the 4-ary symbols has three ideal models. Perfect phase alignment is shown in the middle model. Top models correspond to ⅓ sample of phase lag. Bottom models correspond to ⅓ sample phase lead. All twelve models carry the maximum weight of 10.

Figure 3. Sample Phases

00 Symbol Models

1 1 1 0 0 1 1 1 0 0
1 1 0 0 0 1 1 1 0 0
1 1 0 0 0 1 1 0 0 0

01 Symbol Models

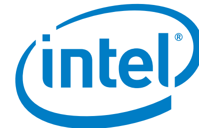
1 1 1 0 0 0 0 0 1 1
1 1 0 0 0 0 0 0 1 1
1 1 0 0 0 0 0 1 1 1

10 Symbol Models

0 0 0 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 0 0 0

11 Symbol Models

0 0 0 1 1 0 0 0 1 1
0 0 1 1 1 0 0 0 1 1
0 0 1 1 1 0 0 1 1 1



The likelihood lookup table contains 1024 entries (one for each possible received symbol R). Each entry contains the maximum correlation weight between the symbol R and the ideal symbol models:

$$W_R = \max(W_{00}, W_{01}, W_{10}, W_{11})$$

as well as the decision corresponding to the highest weight:

$$S_R \in \{00, 01, 10, 11\}$$

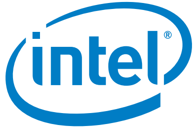
The application indexes the table with each received symbol to obtain a decision regarding the transmitted bits and a weight expressing the likelihood that that these bits were transmitted given that this symbol was received.

3.3 Phase Tracking

Since the number of samples per symbol is fractional but the time resolution of the sampling window is discrete, each time the application moves the symbol sample window forward it will be either lagging or leading in phase. For example: assuming perfect symbol alignment at the start, advancing the symbol sample window forward by 10 samples results in the next symbol lagging by ⅔ sample in phase. Advancing 11 samples results in the next symbol leading by ⅓ sample in phase. Furthermore, the application requirements state that bit rate offset between transmitter and receiver can be as high as ±3 %. Therefore, instead of always advancing the symbol sample window by some predetermined pattern (for example, 11, 11, 10 samples), the application will test both 10 and 11 sample advancements to see which results in the greatest correlation strength. This allows the application to track the symbol phase in the presence of bit rate offset.

3.4 Synchronization

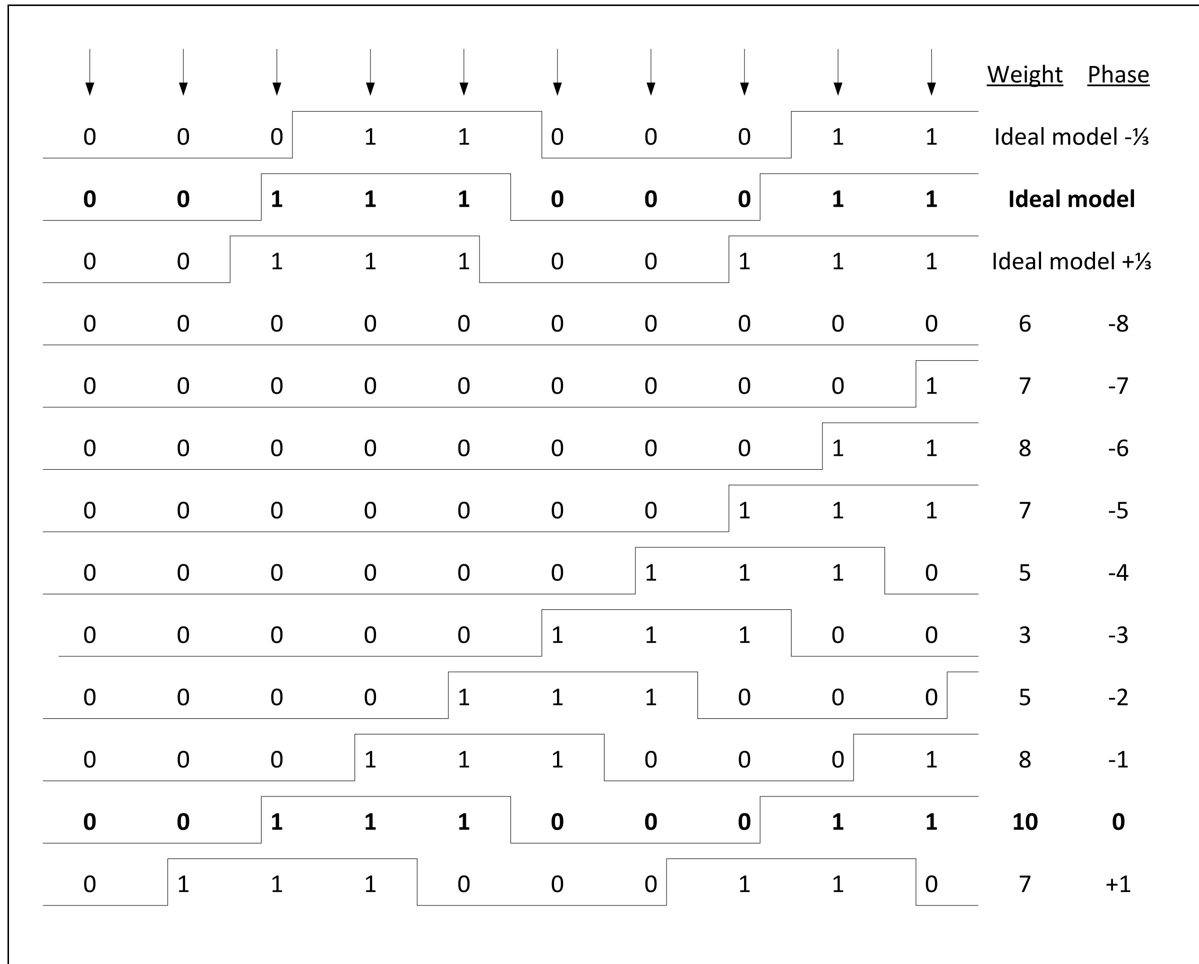
In order to properly decode symbols, the application must first synchronize the symbol sample window with the symbol boundaries in the received stream of samples. Figure 4 illustrates the procedure. Assume that the message begins with the symbol that encodes information bits 11 after receiving all zero samples prior to that. The application receives groups of ten samples from the SPI peripheral. Buffering up at least 20 samples, the application slides a ten-sample window over the 20 buffered samples, seeking alignment with the first symbol in the message. The first three waveforms show the ideal symbol models corresponding to the 11 state. The next ten waveforms show the ten-sample window obtained with each shift of the window as it slides over the twenty-sample buffer. The weight of each ten-sample window and its phase relative to the true start of message are listed in columns to the right of the waveforms. The perfectly aligned window and ideal symbol model are indicated in bold typeface. It can be seen that the highest weight of an unaligned window is eight. Any weight higher than this indicates that synchronization has been achieved. For this application, a value of nine was selected; this is large enough to ensure synchronization while tolerating one sample error. If none of the ten sample windows carry a weight of at least nine, the next ten-sample group received from SPI is appended as the oldest ten-sample group is ejected from the buffer and the next ten windows are tested. This continues until synchronization is achieved.



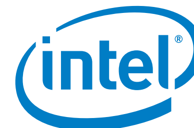
After synchronization is achieved, the symbol window advances 10 or 11 samples with each ten-sample group received from SPI. Eventually, the window will reach the edge of the twenty-sample buffer. When this happens, an additional ten-sample group from SPI is appended before the symbol window advances.

As shown in Figure 4, ten phase shifts of a ten-sample symbol window over 19 samples prior to and including the start of a message. Ideal symbol models for the known message preamble are shown in the first three rows. Bold highlighted rows are a phase aligned.

Figure 4. Ten Phase Shifts of a Ten Sample Symbol Window



§



4 Results

The maximum likelihood Manchester decoder described in previous sections was tested using an Agilent* 33522A Arbitrary Waveform Generator to synthesize a known good 32-byte message with embedded 16-bit CRC. The firmware decodes the message and verifies the CRC. It then prints a message to the UART indicating how many bytes were received, the message content, and any errors encountered. Figure 5 shows the beginning portion of a test message. The 20 µs pulse that precedes the message is used to wake the D1000 from the 2 µA standby state. Some protocols, such as IEEE Std. 802.3, provide a preamble with which receivers can synchronize. In this case, the preamble can be used for both wakeup and synchronization and no wakeup pulse is required.

Tests of duty cycle distortion are shown in Figure 6 and Figure 7. In Figure 6, the high phase of the signal is stretched and the low phase shortened by 357 ns. In Figure 7, the low phase of the signal is stretched and the high phase shortened by 357 ns. In both cases, decoded messages were free of errors.

Tests of bit rate offset are shown in Figure 8 and Figure 9. In Figure 8, bit rate was decreased by 3.0 %. In Figure 9, bit rate was increased by 3.5 %. In both cases, decoded messages were free of errors.

Finally, sample error injection at rates of 0.005, 0.01, and 0.015 showed that the proposed method is indeed robust in the presence of noise. No message errors occurred in more than 4000 messages decoded in the 0.005 and 0.01 error rate cases. The message error rate was 0.266 in the 0.015 sample error rate case.

As shown in Figure 5, oscilloscope waveforms of the received signal in yellow and the core current in red show start of the Manchester-encoded message. A 20 µs wakeup pulse precedes the Manchester encoded message.

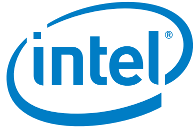
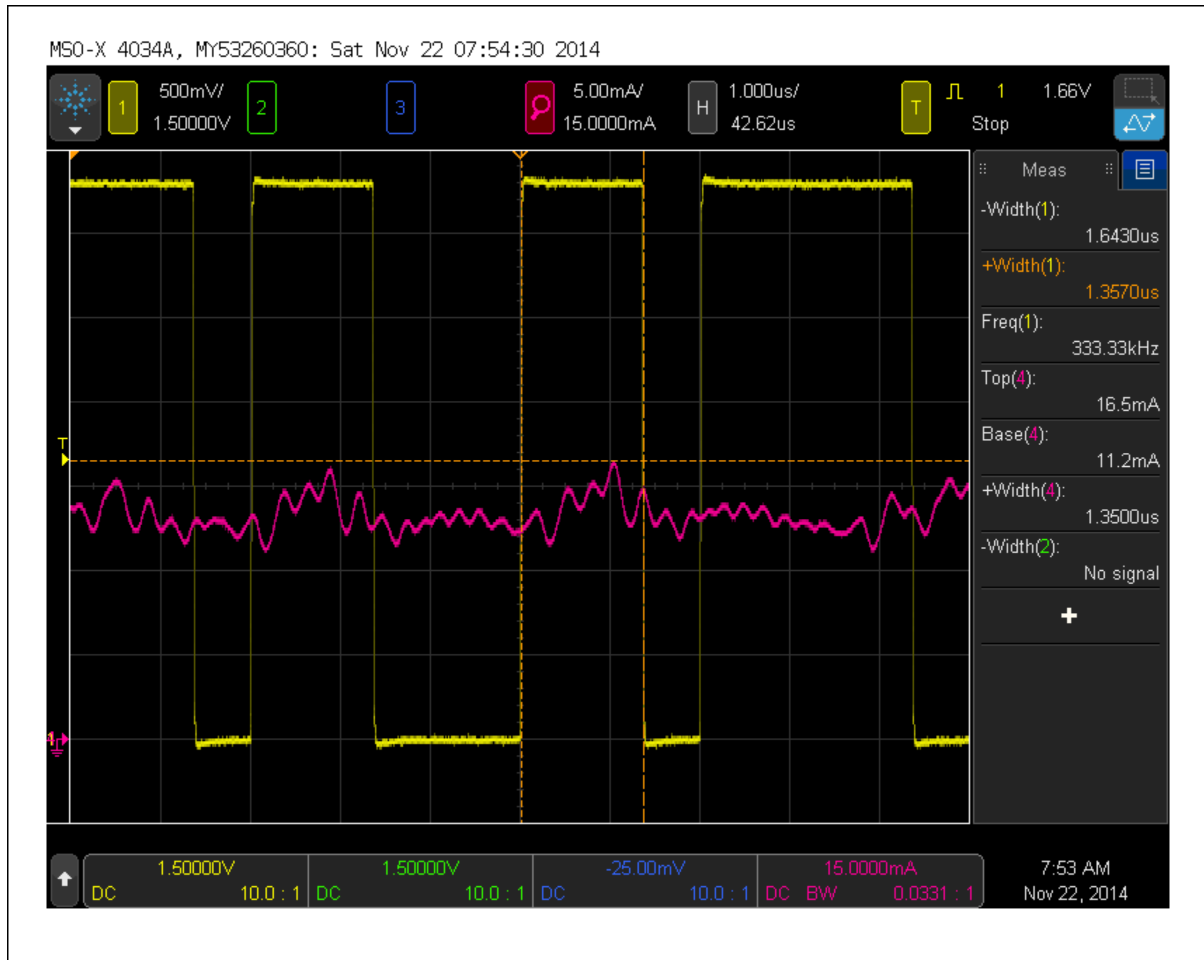


Figure 5. Start of Manchester Encoded Message

Figure 6. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing +357 ns Duty Cycle Distortion



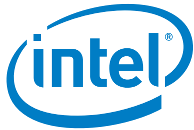


Figure 7. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing -357 ns Duty Cycle Distortion

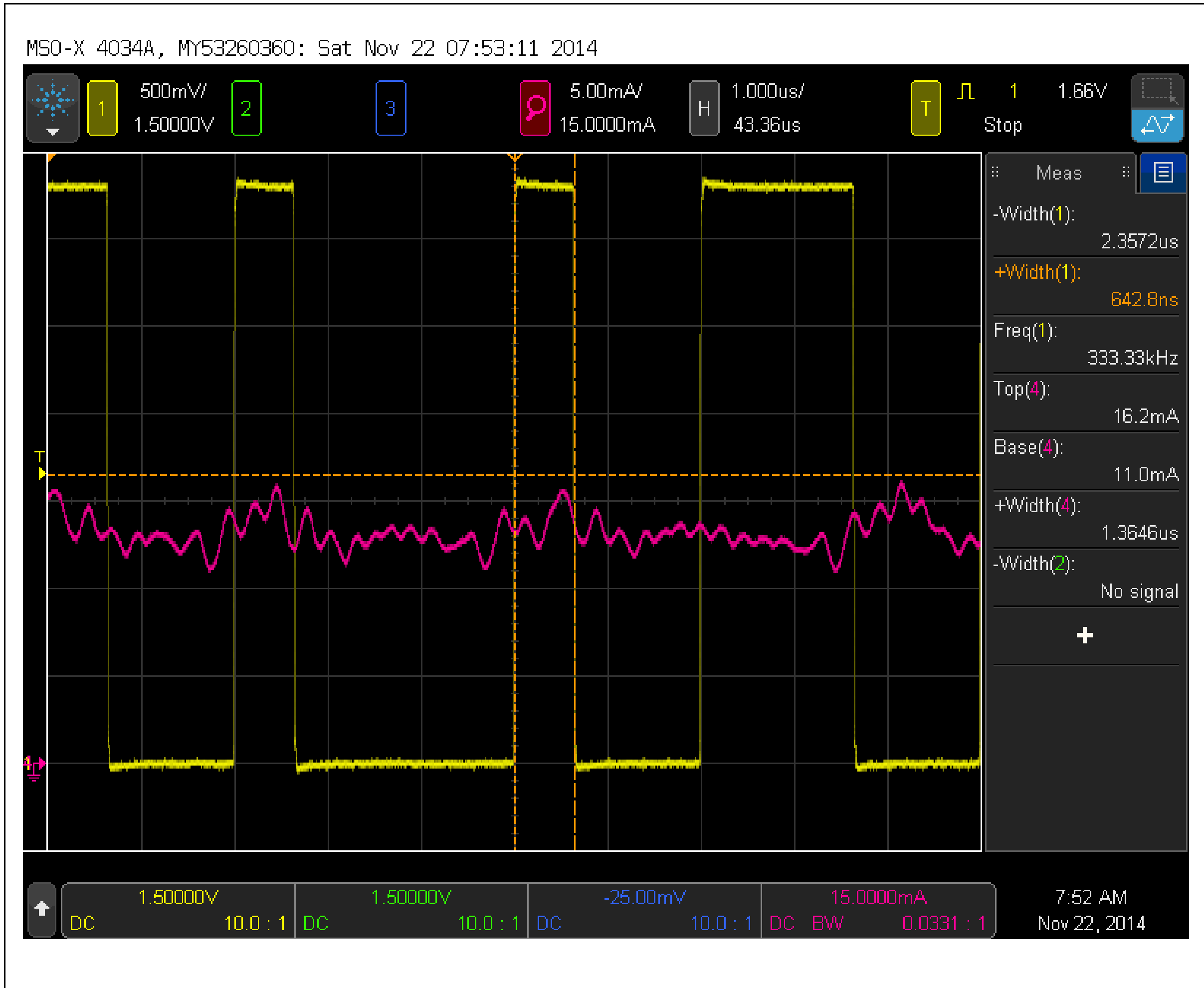
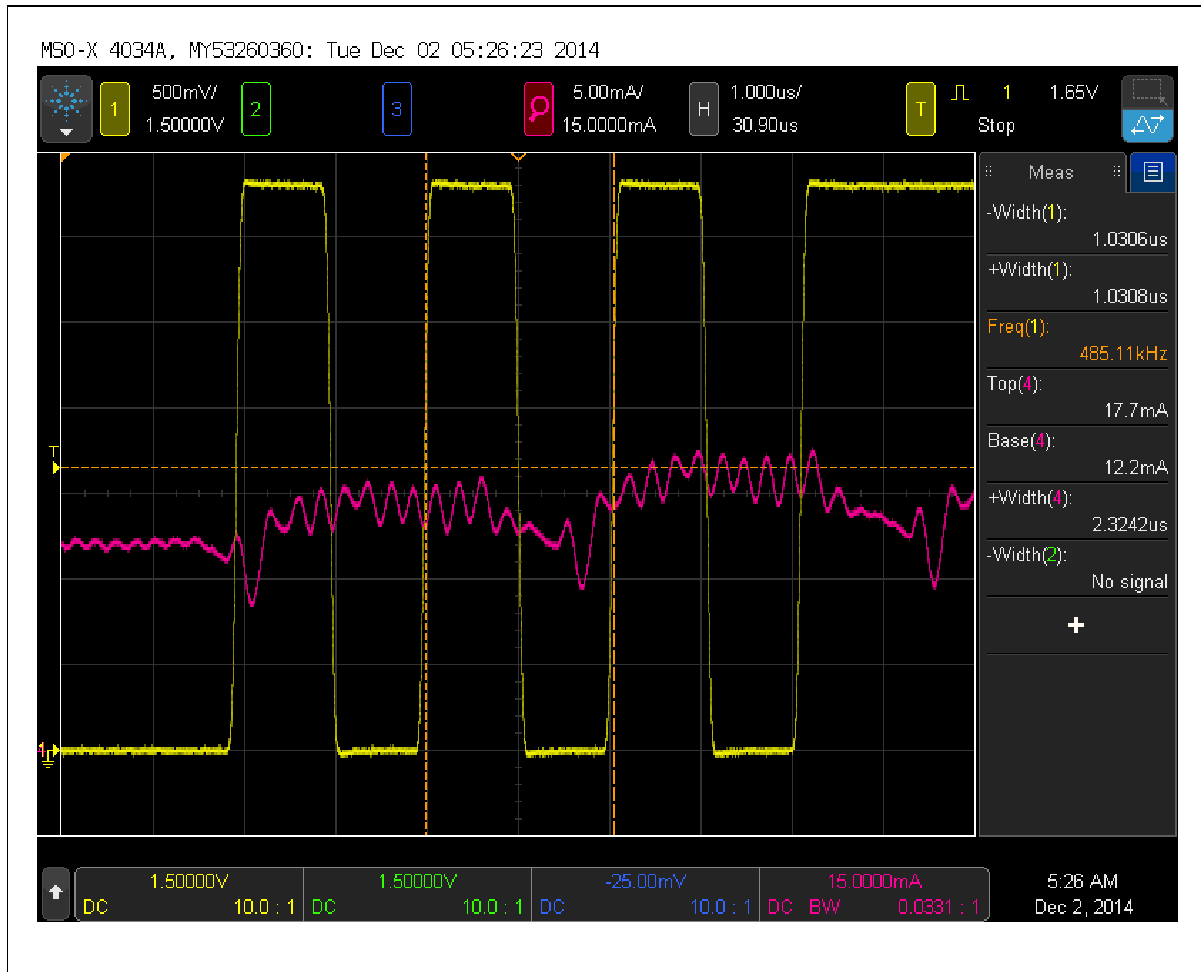


Figure 8. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing -3.0 % Bit Rate Offset



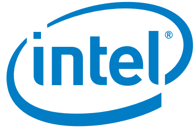
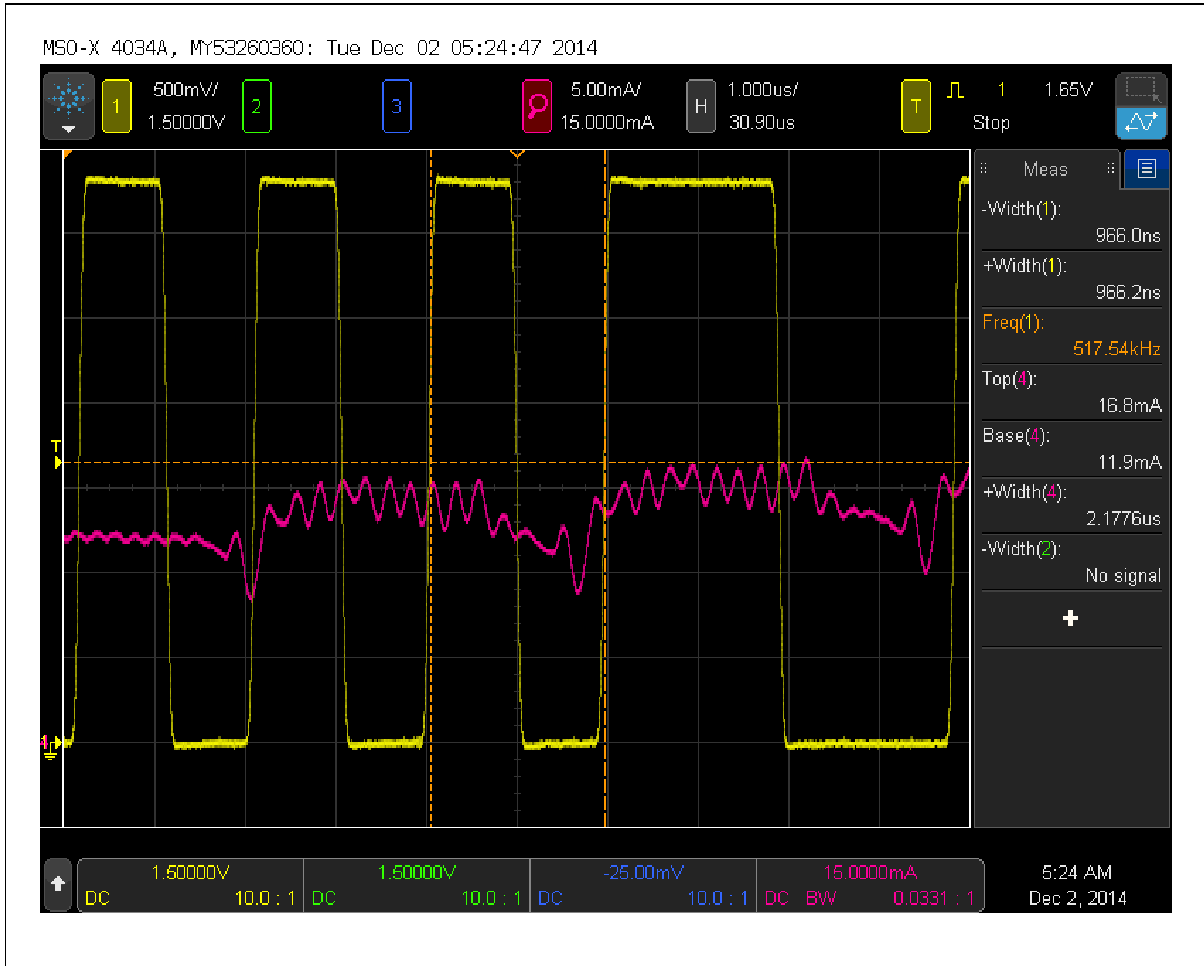
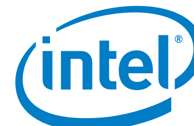


Figure 9. Oscilloscope Waveforms of Received Signal in Yellow and Core Current in Red Showing +3.5 % Bit Rate Offset



§



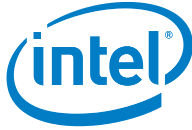
5 Discussion

During the synchronization phase, ten different phase shifts of the symbol sample window are tested for each new symbol received. The time required to complete these computations is greater than a symbol period. The receive FIFO in the SPI peripheral provides the elasticity needed to prevent loss of data. However, there is a limit to the amount of time the application can spend in the synchronization phase before the FIFO overflows. While the developers checked for an overflow condition and found none, the maximum time in the synchronization phase before the FIFO overflows has not been characterized. This could become an issue if the synchronization window opens long before a message arrives. If this situation is unavoidable, a check of each new symbol for a non-zero condition should be used as a necessary pre-requisite to enter the synchronization phase. This non-zero check is much faster than the synchronization check and will prevent FIFO overflows.

For this application, 10-bit 4-ary symbols were selected because they strike a good balance between competing goals of maximizing CPU cycles per symbol, maximizing distance between most likely symbols, and constraining the size of the look-up table. The optimal balance depends on bit rate, the amount of memory that can be allocated to the likelihood lookup table, and the CPU clock frequency. At lower bit rates, 10-bit 2-ary symbols would increase the distance between most likely symbols, improving robustness to errors and distortion. Alternately, if a larger table is possible, 12-bit 4-ary symbols would have the same effect to a lesser extent without compromising bit rate.

Bit rate offset is limited to ±3 % in this example. Larger offsets might also be accommodated with an increased number of likelihood tests at larger phase shifts. However, as bit rate offset increases, the mismatch between received and ideal symbol models also increases, reducing the likelihood that proper alignment can be maintained. Higher sample rates may mitigate this effect somewhat. Further experimentation is needed in order to characterize performance at larger bit rate offsets and higher sample rates.

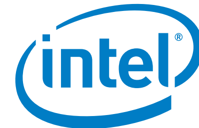
§



6 Conclusion

This paper presented a method for decoding Manchester-encoded data in the presence of noise and distortion. The selected method consists of over-sampling the received baseband waveform and looking up pre-computed likelihood and state decisions from a table. Prior to decoding, symbol synchronization is achieved by sliding a window over the received symbol stream until a strong likelihood indicates that a particular phase shift correctly aligns the symbol window with a transmitted symbol. Alignment is maintained during decoding by testing the likelihood of phase-leading and phase-lagging symbol windows and choosing the alignment with the greatest likelihood. Real-time 500 kbps decoding was achieved with CPU clock frequency of 32 MHz. Furthermore, this method proved robust in the presence of bit rate offset, duty cycle distortion, and noise.

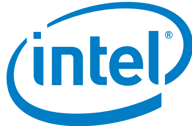
§



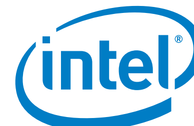
Appendix A – Maximum Likelihood Decoding Program

The following assembly language source code is for the timer 0 interrupt service routine, which contains the maximum likelihood Manchester decoding algorithm. Timer 0 was started when the trigger pulse was detected. Its expiration opens the synchronization window at ~10 µs prior to start of message. Once started, the SPI peripheral will deliver 160 10-bit symbols at a rate of 266667 symbols per second. The maximum likelihood lookup table `man_lut` is a 1024-byte table containing weights in bits 2-5 and decisions in bits 0-1.

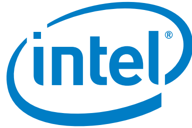
```
////////////////////////////////////////////////////////////////////////

  // define symbols and allocate storage for timer 0 interrupt handler

  ////////////////////////////////////////////////////////////////////////
  .global       timer0_interrupt_handler_stub
  .extern       check_message
  .extern       man_lut
  .extern       error
  .extern       PM_CMP_POL
  .extern       PM_CMP_INTSTAT
  .extern       PM_CMP_INT_EN
  .extern       PM_CMP_PWR
  .extern       LA_EOIR
  .extern       T0_CTL_REG
  .extern       T0_EOI
  .extern       PM_CLK_EN
  .extern       SM_SSIENR
  .extern       SM_DR
  .extern       SM_RISR
  .set          MAXBITS,256      // maximum message 256 bits
  .set          MINWEIGHT,(4*9+3)// minimum weight for synchronization
  .set          T0_CLKEN,(1<<4)  // mask for timer 0 clock enable
  .set          SPIM_CLKEN,(1<<1)//mask for SPI clock enable
  .set          ERR_OVERFLOW,(1<<0)// mask for overflow error
  .set          SM_BUSY,0        // SPI busy status bit
  .set          SM_RXOIR,3       // SPI RX FIFO overflow status bit
  .set          SM_RXFIR,4       // SPI RX FIFO full status bit
  .bss
  .align        4
  .global       rx_buffer
rx_buffer:  // storage for data bits
  .space        MAXBITS/8       // maximum message in bytes
  .global       rx_buffer_end
rx_buffer_end:
  .global       rx_buffer_next
rx_buffer_next: // pointer to next location in data buffer
  .space        4
```



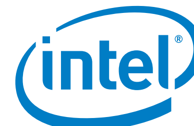
```
  .text
  .align       16
timer0_interrupt_handler_stub:  // begin handler
  pushl %eax                    // save context
  pushl %ebx
  pushl %ecx
  pushl %edx
  pushl %ebp
  pushl %edi
  pushl %esi
  pushl %esi                    // reserve local variable on stack

tm0_isr_stop_timer:  // stop timer and clear interrupts
  xorl  %eax,%eax               // clear register
  movl  %eax,T0_CTL_REG         // stop timer
  movl  T0_EOI,%ebx             // clear interrupt from timer
  movl  %eax,LA_EOIR            // clear interrupt from APIC
  andl  $(~T0_CLKEN),PM_CLK_EN  // disable timer clock

tm0_isr_start_spi:  // start SPI sampling
  orl   $SPIM_CLKEN,PM_CLK_EN   // enable SPI clock
  movl  $1,SM_SSIENR            // enable SPI
  movl  %eax,SM_DR              // write dummy word to SPI TX FIFO

tm0_isr_initial_values:  // initialize the various counters and pointers
  lea   man_lut,%edi            // get pointer to maximum likelihood LUT
  lea   rx_buffer,%esi          // get pointer to data buffer
  movl  $0x3FF,%ebp             // 10-bit symbol mask
  movl  %eax,error              // clear error flags
  movb  $3,(%esp)               // initial data decision down counter

tm0_isr_s0_symbol_0:  // get first symbol
  btl   $SM_RXFIR,SM_RISR       // test RX FIFO full bit
  jae   tm0_isr_s0_symbol_1     // loop until symbol is available
  movl  SM_DR,%ebx              // read RX FIFO

tm0_isr_s0_symbol_1:  // get second symbol
  btl   $SM_RXFIR,SM_RISR       // test RX FIFO full bit
  jae   tm0_isr_s0_symbol_1     // loop until symbol is available
  shll  $10,%ebx                // make room for next symbol
  orl   SM_DR,%ebx              // or in new symbol

tm0_isr_s0_symbol_2:  // get third symbol
  btl   $SM_RXFIR,SM_RISR       // test RX FIFO full bit
  jae   tm0_isr_s0_symbol_2     // loop until symbol is available
  shll  $10,%ebx                // make room for next symbol
  orl   SM_DR,%ebx              // or in new symbol

tm0_isr_s0_phi_0:  // get likelihood at 0 degrees
  movb  $19,%cl                 // initial phase shift
  movl  %ebx,%eax               // get last three symbols
  shrl  %cl,%eax                // shift phase
  andl  %ebp,%eax               // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh       // get likelihood
  cmpb  $MINWEIGHT,%dh          // does this weight meet the minimum?
  jae   tm0_isr_s1_entry        // if so, start processing packet

tm0_isr_s0_phi_1:  // get likelihood at +36 degrees
  decb  %cl                     // decrement phase shift
```



```
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_2:  // get likelihood at +72 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_3:  // get likelihood at +108 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_4:  // get likelihood at +144 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_5:  // get likelihood at +180 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_6:  // get likelihood at +216 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
  jae   tm0_isr_s1_entry         // if so, start processing packet

tm0_isr_s0_phi_7:  // get likelihood at +252 degrees
  decb  %cl                      // decrement phase shift
  movl  %ebx,%eax                // get last three symbols
  shrl  %cl,%eax                 // shift phase
  andl  %ebp,%eax                // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh        // get likelihood
  cmpb  $MINWEIGHT,%dh           // does this weight meet the minimum?
```

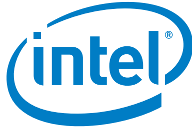


```
  jae   tm0_isr_s1_entry        // if so, start processing packet

tm0_isr_s0_phi_8:  // get likelihood at +288 degrees
  decb  %cl                     // decrement phase shift
  movl  %ebx,%eax               // get last three symbols
  shrl  %cl,%eax                // shift phase
  andl  %ebp,%eax               // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh       // get likelihood
  cmpb  $MINWEIGHT,%dh          // does this weight meet the minimum?
  jae   tm0_isr_s1_entry        // if so, start processing packet

tm0_isr_s0_phi_9:  // get likelihood at +324 degrees
  decb  %cl                     // decrement phase shift
  movl  %ebx,%eax               // get last three symbols
  shrl  %cl,%eax                // shift phase
  andl  %ebp,%eax               // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh       // get likelihood
  cmpb  $MINWEIGHT,%dh          // does this weight meet the minimum?
  jae   tm0_isr_s1_entry        // if so, start processing packet

tm0_isr_s0_spi_check:  // check for SPI still sampling
  btl   $SM_BUSY,SM_SR          // check if SPI is still sampling
  jb    tm0_isr_s0_symbol_2     // if so, get next symbol from RX FIFO
  jmp   tm0_isr_exit            // otherwise, exit now

  .align        16
tm0_isr_s1_symbol_0:  // get symbol
  btl   $SM_RXFIR,SM_RISR       // test RX FIFO full bit
  jae   tm0_isr_s1_symbol_0     // loop until symbol is available
  shll  $10,%ebx                // make room for next symbol
  orl   SM_DR,%ebx              // or in new symbol

tm0_isr_s1_phi_0:  // get likelihood at 0 degrees
  movl  %ebx,%eax               // get last three symbols
  shrl  %cl,%eax                // shift phase
  andl  %ebp,%eax               // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dh       // get likelihood

tm0_isr_s1_entry:  // entry point to decode phase
  movb  %cl,%ch                 // tentative optimal phase shift

tm0_isr_s1_phi_1:  // get likelihood at +36 degrees
  decb  %cl                     // decrement phase shift
  movl  %ebx,%eax               // get last three symbols
  shrl  %cl,%eax                // shift phase
  andl  %ebp,%eax               // mask off all but phase shifted symbol
  movb  (%edi,%eax,1),%dl       // get likelihood
  cmpb  %dl,%dh                 // is shifted symbol more or equally likely?
  ja    tm0_isr_data_bit        // if not, store most likely data bit
  movb  %dl,%dh                 // if so, update weight
  movb  %cl,%ch                 // update optimal phase shift

tm0_isr_data_bit:  // store the data bits
  shrb  $1,%dh                  // shift the hard decision MSB into carry
  rclb  $1,(%esi)               // shift it into the data byte
  shrb  $1,%dh                  // shift the hard decision LSB into carry
  rclb  $1,(%esi)               // shift it into the data byte
  decb  (%esp)                  // decrement the data decision counter
  movb  %ch,%cl                 // recall optimal phase shift
```



```
  jns   tm0_isr_s1_phi_chk_0    // if not data LSB, go bounds check phase
  movb  $3,(%esp)               // re-initialize data decision down counter
  inc   %esi                    // increment pointer to next data byte
  cmpl  $rx_buffer_end,%esi     // is data buffer full?
  jae   tm0_isr_exit            // if so, exit now

tm0_isr_s1_phi_chk_0:  // check for maximum phase shift
  cmpb  $20,%cl                 // maximum phase shift?
  jb    tm0_isr_s1_phi_chk_1    // if not, check for minimum
  subb  $10,%cl                 // if so, skip ahead one symbol
  jmp   tm0_isr_s1_phi_0        // decode it without reading another symbol

tm0_isr_s1_phi_chk_1:  // check for miniumum phase shift
  cmpb  $0,%cl                  // minimum phase shift?
  ja    tm0_isr_s1_spi_check    // if not, continue decoding
  btl   $SM_BUSY,SM_SR          // if so, check if SPI is still sampling
  jae   tm0_isr_exit            // if not, exit now

tm0_isr_s1_symbol_1:  // get symbol
  btl   $SM_RXFIR,SM_RISR       // test RX FIFO full bit
  jae   tm0_isr_s1_symbol_1     // loop until symbol is available
  shll  $10,%ebx                // make room for next symbol
  orl   SM_DR,%ebx              // or in new symbol
  addb  $10,%cl                 // adjust phase shift

tm0_isr_s1_spi_check:  // check for SPI still sampling
  btl   $SM_BUSY,SM_SR          // check if SPI is still sampling
  jb    tm0_isr_s1_symbol_0     // if so, get next symbol

tm0_isr_exit:  // all bits have been stored
  btl   $SM_RXOIR,SM_RISR       // did RX FIFO overflow?
  jae   tm0_isr_spi_off         // if not, go disable SPI
  orl   $ERR_OVERFLOW,error     // otherwise, set the overflow error flag

tm0_isr_spi_off:  // disable SPI
  movl  $0,SM_SSIENR            // disable SPI
  btl   $SM_BUSY,SM_SR          // test for SPI busy
  jb    tm0_isr_spi_off         // loop until no longer busy
  andl  $(~SPIM_CLKEN),PM_CLK_EN// disable SPI clock

tm0_isr_check_msg:  // check message for errors
  movl  %esi,rx_buffer_next     // store where we left off
  call  check_message           // check message content
  orl   $MAN_WK,PM_CMP_PWR      // power up wake comparator

  // restore context and return from interrupt
  popl  %esi                    // remove local variable from stack
  popl  %esi                    // restore context
  popl  %edi
  popl  %ebp
  popl  %edx
  popl  %ecx
  popl  %ebx
  popl  %eax
  iret
```

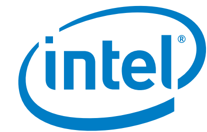
§



Appendix B –Likelihood Lookup Table Generator Program

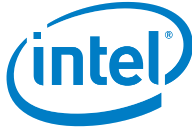
The following MATLAB* source code is generates the likelihood lookup table:

```
% all possible symbols in Boolean matrix form
A = [(mod(0:1023,1024)>511)', ...
     (mod(0:1023,512)>255)', ...
     (mod(0:1023,256)>127)', ...
     (mod(0:1023,128)>63)', ...
     (mod(0:1023,64)>31)', ...
     (mod(0:1023,32)>15)', ...
     (mod(0:1023,16)>7)', ...
     (mod(0:1023,8)>3)', ...
     (mod(0:1023,4)>1)', ...
     (mod(0:1023,2)>0)'];

% ideal symbol models in Boolean matrix form
S = [false,false,true,true,true,false,false,false,true,true; ...  %11
     false,false,true,true,true,false,false,true,true,true; ...   %11
     false,false,false,true,true,false,false,false,true,true; ... %11
     true,true,false,false,false,true,true,true,false,false; ...  %00
     true,true,false,false,false,true,true,false,false,false; ... %00
     true,true,true,false,false,true,true,true,false,false; ...   %00
     true,true,false,false,false,false,false,false,true,true; ... %01
     true,true,false,false,false,false,false,true,true,true; ...  %01
     true,true,true,false,false,false,false,false,true,true; ...  %01
     false,false,true,true,true,true,true,true,false,false; ...   %10
     false,false,true,true,true,true,true,false,false,false; ...  %10
     false,false,false,true,true,true,true,true,false,false];     %10

% generate likelihood lookup table
lut = uint8(zeros(size(A,1),2)); % look-up table in integer
lh = uint8(zeros(1,size(S,1)));  % likelihood table in integer
for k = 1:size(A,1)
    for l = 1:size(S,1)
        lh(l) = sum(not(xor(S(l,:),A(k,:)))); % number of correct samples
    end
    [val, loc] = max(lh); % find maximum likelihood
    lut(k,2) = val;       % store maximum likelihood
    switch (loc);         % store decision bits
        case {1,2,3}; lut(k,1) = 3;
        case {4,5,6}; lut(k,1) = 0;
        case {7,8,9}; lut(k,1) = 2;
        otherwise;    lut(k,1) = 1;
    end
end

% write likelihood lookup table as C array
output = uint8(4*lut(:,2)+lut(:,1));
fid = fopen('man_lut.h','w');
```



```
fprintf(fid,'unsigned char man_lut[] = {\n');
fprintf(fid,'\t%u,\n',output(1:end-1));
fprintf(fid,'\t%u};\n',output(end));

fclose(fid);
```

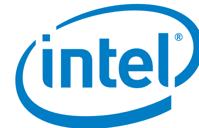
§



Appendix C – Manchester-encoded Data Waveform Generator Program

The following MATLAB* source code generates the waveform file used by the 33522A Arbitrary Waveform Generator to synthesize a known good Manchester-encoded 32 byte message with embedded 16-bit CRC, duty cycle distortion, and noise:

```
%% Parameters
bitRate = 500000;
overSampling = 56;
distortion = 0;
dataPoints = 1000000;
ioVDD = 3.3;
FS = 32767;
errorRate = 0.005;
sampleRate = bitRate*overSampling;
triggerWidth = round(20e-6*sampleRate);
triggerTail = round(26e-6*sampleRate);
hiWidth = overSampling/2;
loWidth = overSampling/2;
zeroBit = [true(hiWidth,1);false(loWidth,1)];
oneBit = [false(loWidth,1);true(hiWidth,1)];
msg = [
    'f4';
    'c1';
    '00';
    '00';
    '00';
    '02';
    '00';
    '00';
    '00';
    '00';
    '00';
    '00';
    '00';
    '18';
    '4b';
    '71';
    '08';
    '07';
    'b2';
    'c2';
    '5b';
    '88';
    '40';
    '00';
    '68';
    '00';
    '00';
    '00';
```



```
    '01';
    'b0';
    'ae';
    '2e'
    ];

%% Manchester encode message
n = size(msg,1);
encodedMsg = false(overSampling,8*n);
for k = 1:n;
    byte = uint8(hex2dec(msg(k,:)));
    for b = 7:-1:0;
        m = (k - 1)*8 + 8 - b;
        if bitand(byte,2^b);
            encodedMsg(:,m) = oneBit;
        else
            encodedMsg(:,m) = zeroBit;
        end
    end
end

%% duty cycle distortion
d = abs(distortion);
if distortion < 0;
    distortedMsg = ...
        and([encodedMsg(:);false(d,1)],[false(d,1);encodedMsg(:)]);
else
    distortedMsg = ...
        or([encodedMsg(:);false(d,1)],[false(d,1);encodedMsg(:)]);
end

%% noise
noise = rand(numel(distortedMsg),1) <= errorRate;
noiseMsg = xor(distortedMsg,noise);
tail = false(dataPoints - triggerWidth - triggerTail - numel(noiseMsg),1);
outputSignal = [true(triggerWidth,1); false(triggerTail,1); noiseMsg; tail];

%% Open file and write data
fileName = 'manchester.arb';
str = input(['Enter file name [',fileName,']: '],'s');
if ~isempty(str); fileName = str; end
fileId = fopen(fileName,'w');
fprintf(fileId,'Copyright:Intel Corp., 2015\n');
fprintf(fileId,'File Format:1.10\n');
fprintf(fileId,'Channel Count:1\n');
fprintf(fileId,'Sample Rate:%d\n',sampleRate);
fprintf(fileId,'High Level:%d\n',ioVDD);
fprintf(fileId,'Low Level:0\n');
fprintf(fileId,'Data Type:"short"\n');
fprintf(fileId,'Data Points:%d\n',dataPoints);
fprintf(fileId,'Data:\n');
fprintf(fileId,'%d\n',FS*outputSignal);
fclose(fileId);
```

§