NXP

Using the XGATE for Manchester Decoding

by: Lech Olmedo
Guadalajara, Mexico
Manuel Alves
Austin, USA

The objective of this document is to demonstrate the feasibility of using the XGATE programmable peripheral on the HCS12X microcontroller family as a Manchester decoder for radio frequency (RF) receiver applications. The XGATE is a reduced instruction set computing (RISC) peripheral that allows data transfers between peripherals and internal RAM. The process of decoding Manchester-coded signals requires intensive interrupt servicing from timer inputs, as well as high central processor unit (CPU) loading. These applications usually experience noisy environments, thus making Manchester decoding an even more CPU-intensive task. The XGATE programmable peripheral has been designed to offload CPU loading from the HCS12X core by treating interrupt service routines without stopping the program execution on the HCS12X core. This application note discusses a proposed implementation to decode Manchester-coded signals using the XGATE. The implementation is based on a typical automotive RF receiver application, such as remote keyless entry (RKE) or tire pressure monitoring (TPM). Beyond demonstrating the feasibility of Manchester decoding, this note highlights the additional benefits of using the XGATE.

Table of Contents

freescale™
semiconductor

1 Introduction

Manchester encoding is a synchronous clock encoding technique typically used in digital transmissions to encode clock and data in a single-bit stream. It is widely used in RF applications, such as remote keyless entry, home automation, monitor sensor, and tire pressure monitor sensing. Manchester encoding does not use a normal sequence of logic 1s and 0s, (non-return-to-zero—NRZ). Instead, a high-to-low level transition in the middle of the bit duration means a binary 1, and a low-to-high level transition in the middle of the bit duration means a binary 0.
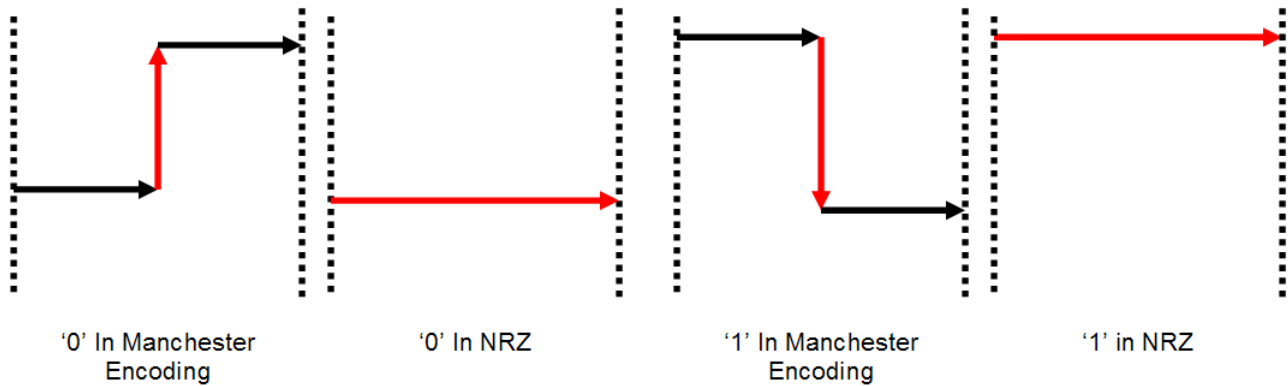

Figure 1. Manchester Encoding

1.1 XGATE Module in HCS12X Microcontrollers

This section describes a Manchester decoding implementation using the XGATE programmable peripheral of the HCS12X microcontroller. The XGATE is a reduced instruction set computing (RISC) peripheral that allows data transfers between peripherals and internal RAM. Most of its functions are designed to increase the HCS12X CPU's throughput by lowering its interrupt load. The Manchester Decoding algorithm implemented is ideally suited for execution using the XGATE, because it allows the HCS12X CPU to perform any other necessary function in an application, instead of being loaded with the Manchester decoding.

1.1.1 XGATE Module Features

The XGATE module features the following:

- Data movement between various targets (Flash, RAM, peripheral modules)
- Data manipulation through built-in RISC core
- Up to 112 XGATE channels
- Hardware semaphores shared between HCS12X CPU and XGATE
- Ability to operate in run and wait modes
- Fully programmable in C, similar to programming an common interrupt routine in a microcontroller unit

- Programming, compiling, and debugging is integrated in CodeWarrior environment, which makes XGATE a very easy to use programmable peripheral
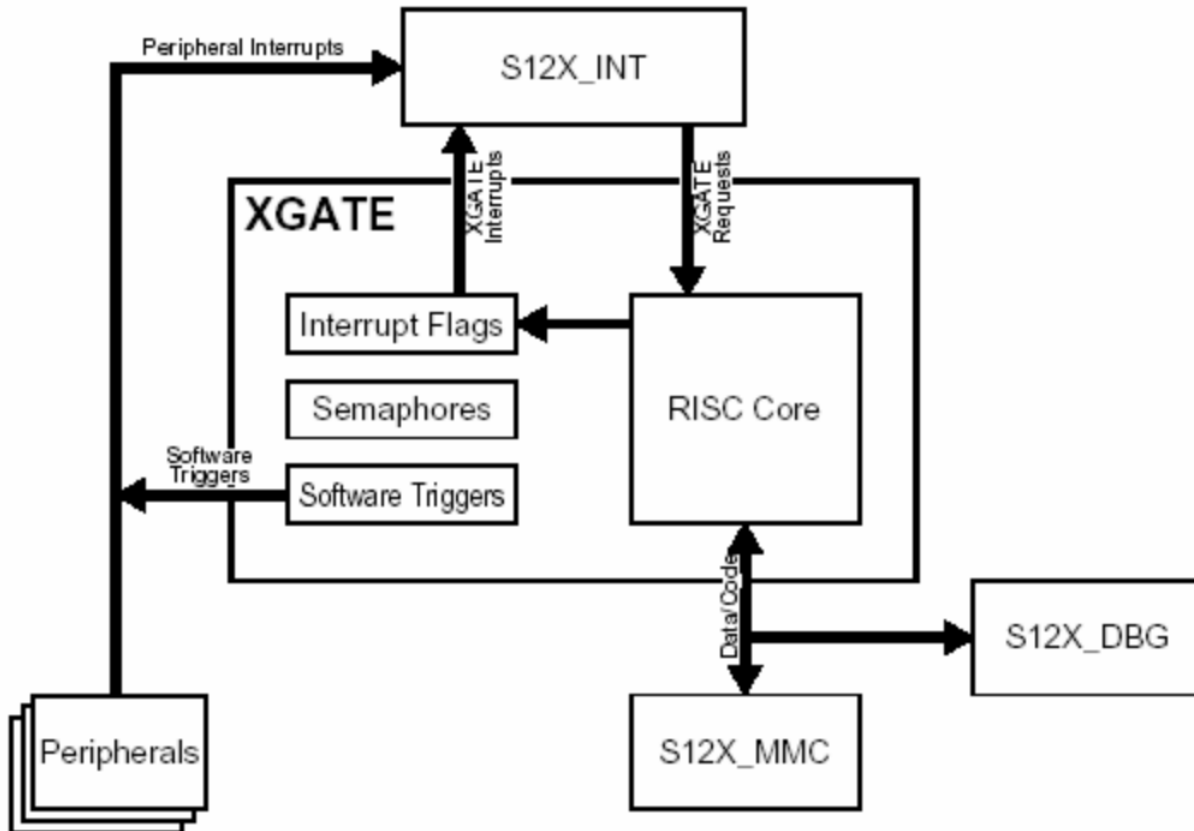
1.1.2 XGATE Block Diagram

Figure 2. XGATE Block Diagram

1.1.3 XGATE Servicing Interrupts

- One of the key features of the XGATE is the ability to service interrupts that the HCS12X CPU would usually service. Each interrupt in HCS12X MCUs can be serviced by two different options: the HCS12X CPU or the XGATE programmable peripheral. To decide if an interrupt will be taken by the HCS12X CPU or the XGATE, when setting the priority of the interrupt in the interrupt request configuration data registers (INT_CFDATA0-7), indicate in the RQST field whether the interrupt will be taken by the HCS12X CPU or the XGATE. Setting the RQST bit high (1) allows the service interrupt to be taken by the XGATE. By default, it is low (0).

The resources used are the following:

- MC9S12XDP512 microcontroller
- Clocks and reset generator (CRG) module for timing configuration (working with external oscillator at 16 MHz, bus clock at 8 MHz)

- Enhanced capture timer (ECT) channel 0 input capture functionality (input capture interrupt, rising edge detection); necessary for the decoding algorithm

- Serial communication interface 0 (SCI0) modules for demo purposes, user communication (received interrupt enabled); only enabled in demo configuration

- Port B for LED

- 2 Kbytes of XGATE code in RAM

- 20% XGATE loading

Devices used for validation purposes:

- The Tango3 device or MC33493 is a phase-locked loop (PLL)-tuned, ultra-high frequency (UHF) transmitter that sends the signal in Manchester encoding using the software of AN2777 (software monitor for the MC33493)

- The Romeo2 device or MC33591 is the PLL UHF receiver; both devices can work in 315- and 434-MHz bands using OOK (on/off key) and FSK (frequency shift key) modulation

- The 434-MHz band was used for this project

- A Manchester encoder was implemented with an MC68HC908QY4 for testing the implementation without noise presence

For more information about Freescale devices, go to www.freescale.com.

2 Decoding Algorithm

The basic idea of the decoding algorithm is using the input capture capability of the timer configured for rising-edge detection; this is the main element of the decoding process. Only one timer channel is needed. When a Manchester-encoded signal is sampled in a channel with rising-edge detection, every rising edge is detected and generates an interrupt that usually is serviced by the HCS12X CPU. In this implementation, this interrupt is serviced by the XGATE programmable peripheral. The selection of the XGATE acting as the Manchester decoder is defined in the Manchester_Decoder_General.h file.

For decoding, the time when each rising is detected is saved and subtracted from the previous one. The pulse width between the rising detections defines the decoded value. In general terms, while a Manchester signal is being sampled, there are only three cases of time measurement: after one bit, after one and a half bits, and after two bits. The resulting decoded value is obtained by time measurement and the position of the rising edge detection (it can be at the middle of the data bit, or at the end of the data bit).
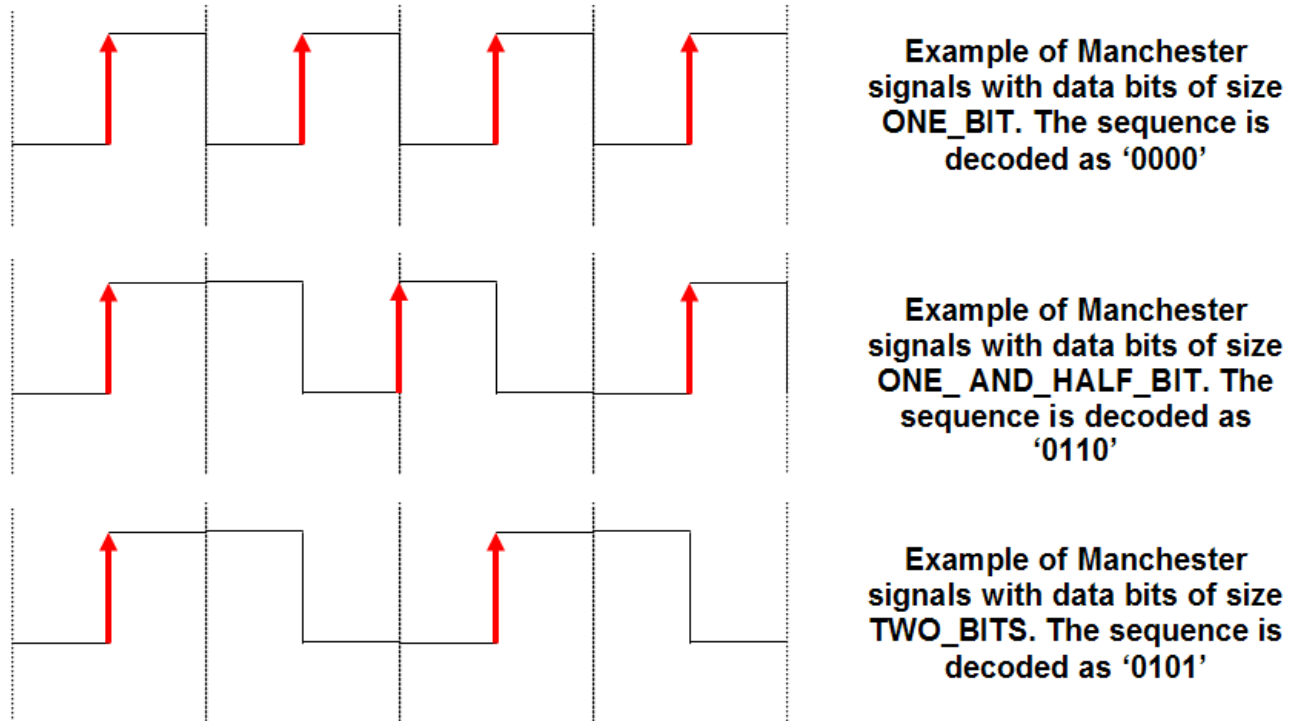
Figure 3. Different Types of Rising Edge Detections

Table 1 provides the logical truth table to decode time measurements into bits.

The inputs in Table 1 are the times measured and stored in gu16TimeMeasured. The other input is the position where the rising edge occurs within the bit. In this implementation, the variable storing this information is called gu8RisingPosition. A value of 0 means a rising detected in the middle of the bit, and 1 means a rising detected at the end of the bit.
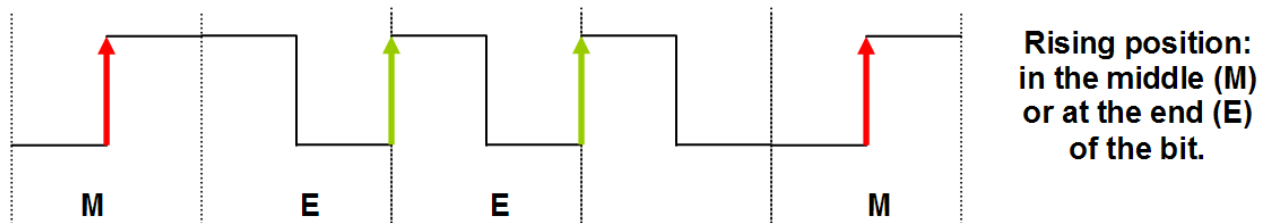

Figure 4. Rising Edge Position Measurement

The outputs are the values decoded and saved in a buffer (gau8BufferTemporalXX). In the cases of 1.5 and 2 bits, when the rising occurred in the middle of the incoming data bit, it is possible to decode the next bit as well. Finally, the last output is the value of the index that indicates the position of the buffer where the values are written (gu8DecodeIndex).

Table 1. Manchester Decoding Truth Table

Inputs		Outputs		
Time Measured: *gu16TimeMeasured*	Position of the Measured Rising: *gu8RisingPosition*	Value Decoded: gau8BufferTemporalXX [gu8DecodeIndex]	Next Value Decoded gau8BufferTemporalXX [gu8DecodeIndex+1]	gu8DecodeIndex
1 bit	In the Middle	0	NA	gu8DecodeIndex++
1 bit	At the End	1	NA	gu8DecodeIndex++
1.5 bits	In the Middle	1	0	gu8DecodeIndex+=2
1.5 bits	At the End	1	NA	gu8DecodeIndex++
2 bits	In the Middle	1	0	gu8DecodeIndex+=2

At the beginning of a decoding process, there is no certainty if the detection of the rising edge occurred in the middle of a data bit or at the end, but there is some information that is useful in finding out. This information is based on when the previous rising-edge detection was made.

- 1 bit: The place where the last rising edge was detected (in the middle or at the end) remains the same.
- 1.5 bits: The place where the last rising edge was detected (in the middle or at the end) has changed.
- 2 bits: The place where the last rising edge was detected (in the middle or at the end) is in the middle of the data bit.

Based on the previous statements, after a detection receives a 2-bit measurement, we can be sure that the position of the measured rising is in the middle of the data bit. For this reason, it is recommended to send a sequence containing a 2-bit measurement before starting sending useful data on the transmitter side.

The variable gu8RisingPosition is initialized with 0 after receiving the sequence containing the 2-bit measurement. After this pattern is detected, the information can retroactively be decoded based on the above rules.

RF frames usually contain a preamble, header, or elements before the data bits, so that one knows the value of the gu8RisingPosition variable before the data bits start being transmitted. This allows the user to initialize the variable correctly without waiting for a sequence with a 2-bit measurement.

The decoding algorithm must be robust enough to deal with noise disturbing the signal being decoded. On top of regular noise, a weak RF signal can translate into jitter or glitches on the Manchester-encoded signal coming out of the RF receiver chip. This means the pulse width of the signal can vary and affect the decoding process. A good solution is to allow time windows for each rising edge that occurs and determine the bit pattern based on that.

- Time accepted as 1-bit: 0.75 bit <= Time for 1 bit (BIT_1) < 1.25 bit
- Time accepted as 1.5-bit: 1.25 bit <= Time for 1.5 bit (BIT_1_5) < 1.75 bit
- Time accepted as 1-bit: 1.75 bit <= Time for 2 bits (BIT_2) < 2.25 bit

Any other measurement detected would be considered invalid.

In summary, four steps must be taken to decode a Manchester-encoded signal:

1. Initialize the decoding sequence by detecting the 010 2-bit length pattern.
2. Detect the rising edges.
3. Determine the bit pattern with time measurement between rising edges.
4. Decode based on the truth table (Table 1).

At this point, we know the decoding algorithm. It can be divided into three functions: detection (TimerChannel0Isr ()), time measurement (MDTimeMeasurement ()), and decoding (MDDecode ()), which follows the data in Table 1.

3 Software Implementation

So far, this discussion hasn't included important elements for a complete implementation of Manchester decoding. Topics like management of the storage of decoded data, background noise considerations, and a frame scheme are missing. This section will describe how these topics are handled and implemented.

3.1 Frame Scheme

The frame scheme chosen for this implementation is the same the Romeo2 data manager uses. The frame scheme contains a preamble, clock recovery period, ID, header, data, and end of message. The Romeo2 device works with two different modulations (just like the Tango3): OOK (on/off key) and FSK (frequency shift key) modulation. The frame scheme is the same for both modulations, except in the preamble stage.
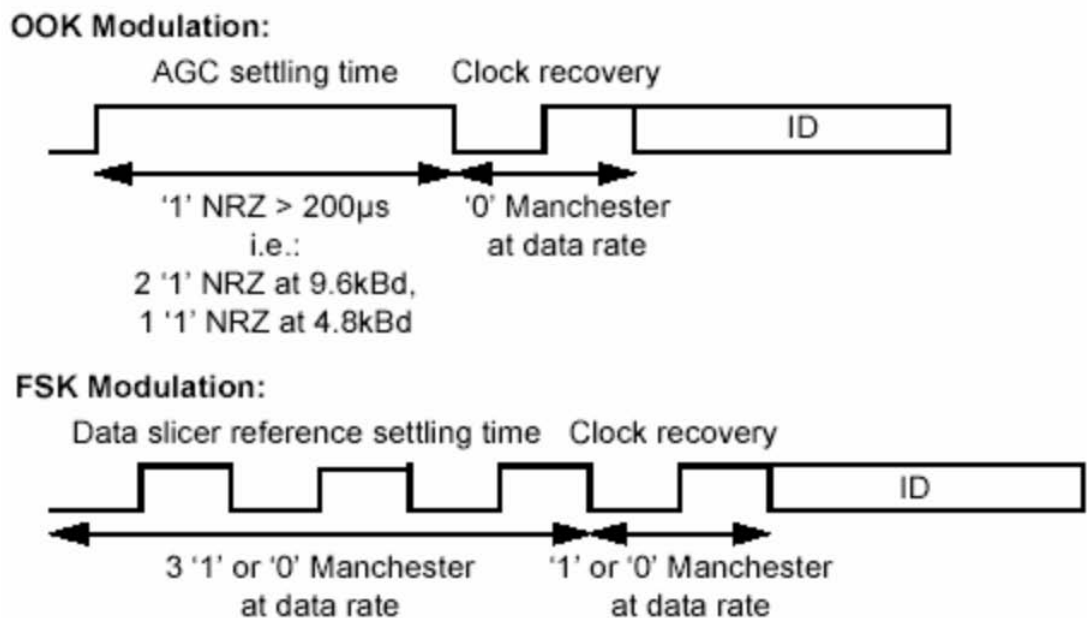

Figure 5. Preamble Differences

For the implementation presented in this application note, the OOK preamble is two 1s in NRZ at 9.6 kbaud. In FSK modulation, the preamble is three 0s in Manchester at 9.6 kbaud. The clock recovery for both cases is one 0 in Manchester encoding at 9.6 kbaud.

ID: The ID is composed of eight bits in Manchester encoding. There is no limitation in the selected number for the ID; the only boundary is that the ID must not contain the same digits of the header. For this implementation, the ID used is 00001111 in Manchester encoding. If a frame contains a different ID, or a different ID is decoded from the one expected, the frame gets discarded.

Header: The header is the previous stage of data. Data must follow the header without any delay or preamble. The header is a 4-bit, Manchester-coded message. In the Romeo2 device, it can be 0110 or its complement; for this document the header is 0110. If a frame contains a different header from the one expected, the frame gets discarded.

Data: The content of this stage is not limited. The number of bits sent varies between implementation. For example, the Tango3 monitor AP2777 allows almost 200 data bits, which sends 10 data bits. One can change the number of data bits by changing the size of the buffers.

End of message: Indicates the end of a complete frame. The end of message is the same for both modulations. It is represented by two 1s in NRZ encoding at 9.6 kbaud.

3.2 Operating Modes and Demo

3.2.1 Complete Mode

The purpose of complete mode is to detect a complete frame (containing all the elements already mentioned) in a noisy environment. In this mode, a complete frame is sent only when the user decides to send it. It can send a complete frame at any moment, and the Manchester decoding implementation must detect it.
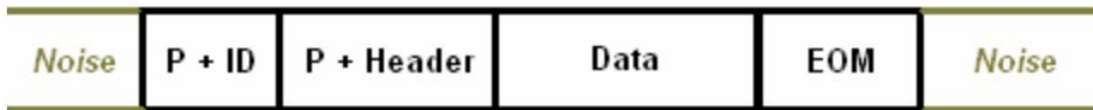

Figure 6. Frame Structure

3.2.2 Demo

- A demo has been prepared to show the implementations in a more interactive way. The demo implements serial communication between the HCS12X evaluation board and the PC serial interface, which is the HyperTerminal. It is configured at 9600 baud rate, 8 data bits, no parity, 1 stop bit, and no flow control.

- The SCI0 module has also been configured to follow this setup and start communication with the HCS12X evaluation board. Transmitter and receiver mode are enabled, but just the receiver interrupt is used.

- A frame is sent from the Tango device or the Manchester encoder, and every time a complete frame is received, a counter appears with the amount of completes frames already received and the last 10 bits of data in the final data buffer. The complete mode is implemented in FSK and OOK modulations.

- All the code generated by the SCI0 module and the interrupt service is executed by the HCS12X CPU. The XGATE handles all the functionality related to the Manchester decoding algorithm.
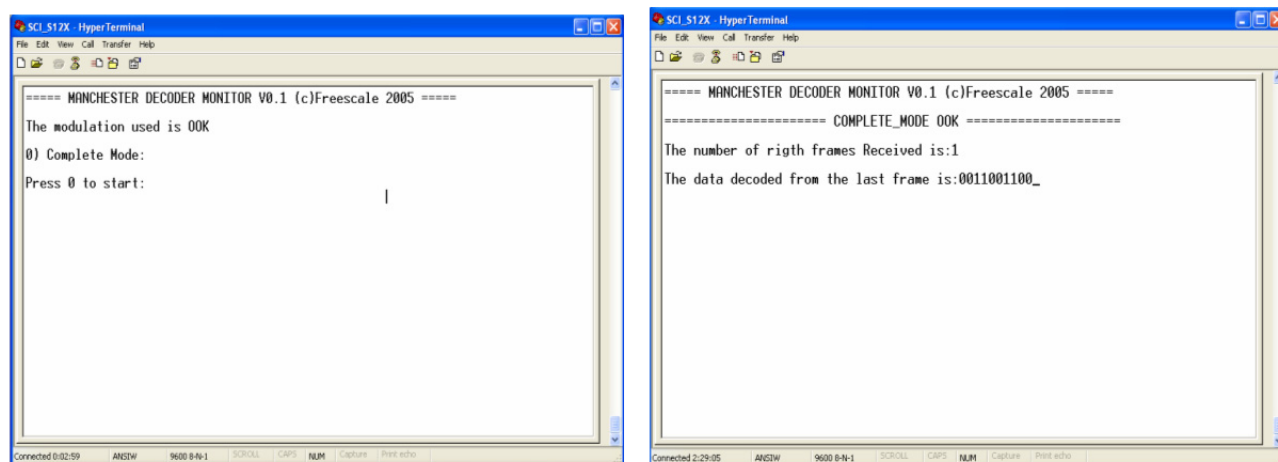

Figure 7. Demo Prompt (Left) and Demo Display (Right)

3.3   Files Summary

The purpose of this implementation is to use the XGATE programmable peripheral for the decoding algorithm; however, it is also implemented for the HCS12X CPU to work as a decoder. The selection of the core that will be performing the Manchester decoding is in Manchester_Decoder_General.h. Other decisions such as the mode used are also defined in this file.

This section describes the steps taken in this project and how the drivers and files are organized. It includes some brief comments about the Manchester decoding drivers.

After the project is opened in the code warrior IDE (the version employed for coding and testing was CW for HCS12X V4.0), there are some files and folders that are useful in allowing the user to see the implementation or change the functionality of the software.

Figure 8. File Structure

3.3.1   General Overview

- General folder: Contains the Manchester_Decoder_General.h file; the main header where the global definitions are present. This is also where important decisions are made, such as the core, mode of implementation, encoder, and enabling of the demo.

- Sources: Contains general files given by default when any XGATE project is created. The most important files for this implementation, or at least those files that can be modified by the user, are the following:

  — Manchester_Decoder_main.c: Contains the main function, global variables initialization, demo code, XGATE setup, timer channel 0 service routine (input capture rising edges detection), and SCI0 receiver service routine.

  — vector_s12x.c: Contains the interrupt vector table for the HCS12X CPU, as well as the prototypes for interrupt handlers serviced by the HCS12X CPU.

  — xgate.cxgate: The equivalent of the vector_s12X.c file, but for the XGATE module. It contains the interrupt vector table for the XGATE, and the service routine for those interrupts serviced by the XGATE. In this case, the only one is the timer channel 0 service routine (input

capture rising-edge detection), when the XGATE is the resource selected to perform the Manchester decoding.

- Drivers: This group contains the decoding algorithm, definitions, and prototypes needed to achieve the implementation. The decoding algorithm is contained in these files.

  — HCS12X CPU sources / XGATE sources: As previously described, there are two modulation modes and one operating mode (complete mode), so each file is made for a modulation mode. Example: OOK modulation (Manchester_Decoding_ook.c / XGATE_Manchester_decoding_ook.cxgate), or FSK modulation (Manchester_decoding_fsk.c / XGATE_Manchester_decoding_fsk.cxgate).

  — Headers: Contain prototypes and definitions needed for the sources. Some definitions are the buffer sizes, bit sizes in timer counter terms, stage definitions, and error definitions. There are two files, one for each kind of modulation mode (Manchester_decoding_ook.h). The same header works for the HCS12X CPU or XGATE source.

  — The driver sources group contains Manchester_Decoding Module_Setup.c. This file has the initialization of the CRG, ECT, and SCI0 modules.

3.3.2 Manchester Drivers

This section contains descriptions of the functions of the decoding algorithm files and the module setup file.

3.3.2.1 Module Setup

This file includes the initialization of the CRG module. For this project, it is initialized by disabling real time interrupt (RTI) and computer operating properly (COP). The PLL is also disabled. The bus clock is equal to the system clock divided by two. The system clock is originated by an external clock source running at 16 MHz.

The ECT module is initialized for input capture in all the channels, but only channel 0 interrupt is enabled. Channel 0 is configured to detect only rising edges and generate an interrupt request when that occurs.

The SCI0 module is configured to work at 9600 baud. Transmitter and receiver are enabled, but only the receiver interrupt is enabled.

Besides the module setup, there are three routines for the management of the serial interface when using the demo: sending messages, receiving messages, and converting long numbers into valid ASCII characters.

3.3.2.2   Decoding Algorithm Functions

This section describes the functions contained in the Manchester decoding files for the XGATE and HCS12X CPU. The description is generic; the differences between modulation modes will be described in the next section.

- MDTimeMeasurement: Makes the subtraction between the last rising edge detection and the previous one to determine the type of bit measured.

- MDDecode: Follows Table 1 to decode a bit sent in Manchester encoding. It is divided into three main blocks; each block represents the type of bit measured by the MDTimeMeasurement () function (1 bit, 1.5 bits, or 2 bits).

- MDIncrement: Increments the values of the index variables of the buffers. It fills the final buffers (ID and data) when all the bits that are supposed to be decoded are done. This function is where stage transitions occur (ID to header, header to data, etc.).

- MDIdlePosition: The error case. Whenever there is a non-programmed event or an error, this function is called. It also initializes variables and gets the application ready to look for a new frame.

- MDStartListening: Originated from the RF tests, this is in charge of detecting the preamble of the frame even in noisy environments; also starts the measuring and decoding process.

3.4   Complete Mode Flowchart

The following flowchart shows the main functions and transitions by stage of the frame.

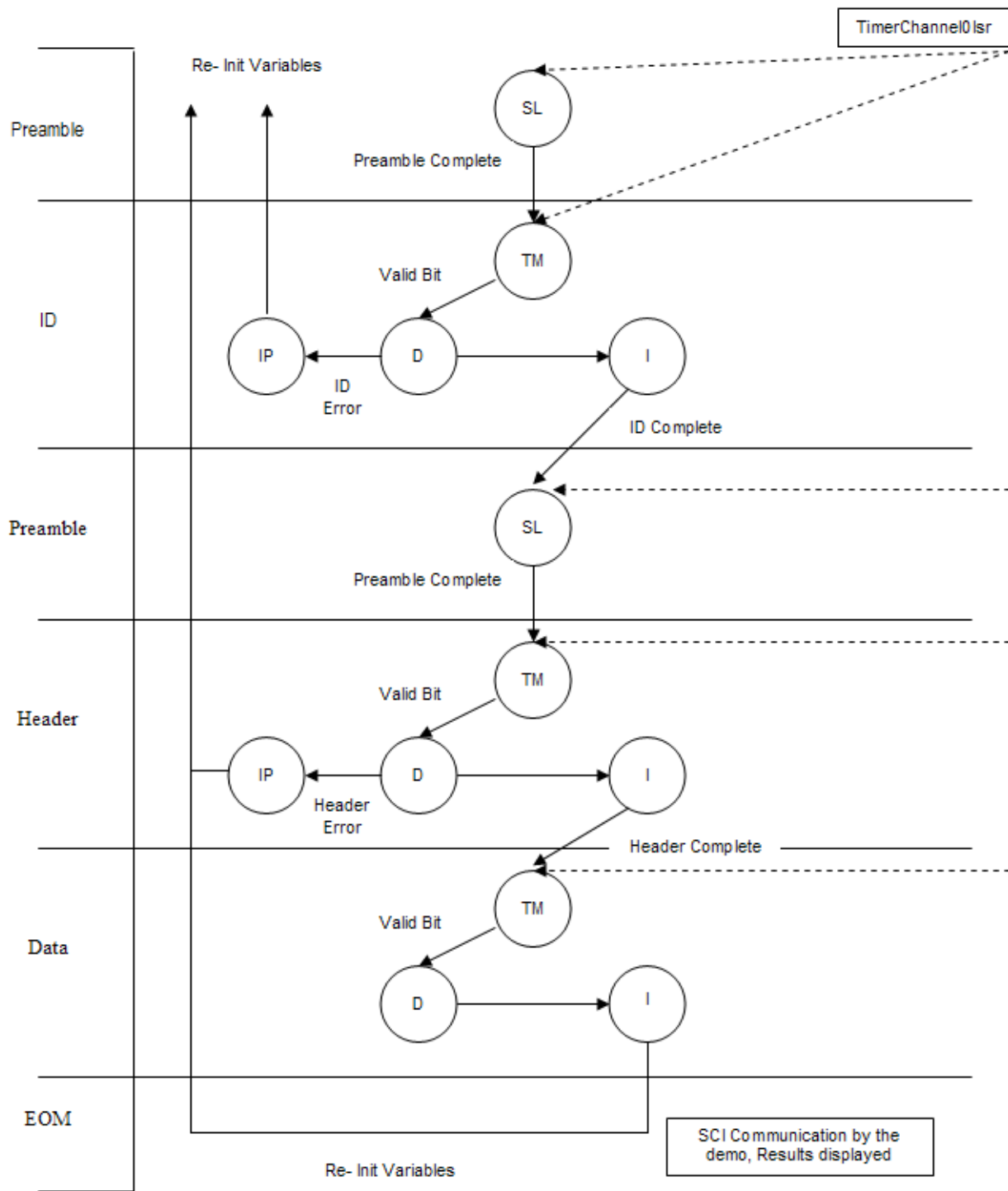SL: StartListening, TM: TimeMeasurement, IP: IdlePosition, D: Decode, I: Increment.

Figure 9. Complete Mode Flowchart

3.4.1 Stages Flow

In all stages, the TimerChannel0Isr is sending information about rising edge detections that must be taken by the MDStartListening () or the MDTimeMeasurement () functions, depending on the stage. Only some variables will be mentioned in the explanation of the process.

1. Preamble (Id): MDStartListening () waits for the proper sequence to complete the preamble: either four Manchester '0's for FSK, or a time measurement longer than 10 bits for OOK. After it is completed, the stage is changed (gu8Stage = ID), and the decoding process is enabled (gu8StartDecoding = 1).

2. ID: Now MDTimeMeasurement () takes the interrupt's results; if the result of the time measured is a valid bit number (1, 1.5, or 2 bits), then MDDecode () will be called (gu8FlowControl == 1). MDDecode () will write the decoded values in gau8BufferTemporalId and compare with the ID default buffer. If there is a difference, MDIdlePorcess () is called and will re-initialize the global variable values. If there is no error and the ID was received okay (when gu8DecodeIndex reaches the LIMIT), then the stage changes (gu8stage = HEADER) and the final ID buffer is filled.

3. Preamble (Header): gu8StartDecoding is again disabled, so the interrupts will reach MDStartListening () again, which will look for the new preamble. This enables (gu8StartDecoding = 1) the decoding process.

4. Header: This is very similar to the ID stage; the only difference is that now the filled buffer is gau8BufferTemporalHeader. It also compares with the default value. If there is a difference, it will go to the MDIdlePosition (); if there is no error and the header is received properly (gu8DecodeIndex reaches the HEADER_LIMIT), the stage changes (gu8Stage = DATA).

5. Data: The data section only fills the temporal buffer with the data decoded. When gu8DecodeIndex reaches the BUFFER_LIMITS, the data is completed, and the final data buffers receive the data decoded that was stored in gau8BufferTemporalData. The next process is to re-initialize the variables for starting the process. ***In this section, when a complete frames is received, a counter is incremented (gu16CompleteFramesCnt++).***

6. EOM: If enabled or implemented, the variable gu16FinishFrameInd is set. This causes the gu16CompleteFramesCn value to display through SCI in the HyperTerminal, as well as the 10 bits of data in the gua8bufferFinalData.

3.4.2 OOK and FSK Differences

OOK and FSK differ in their preambles. For getting the OOK preamble, the code can wait for only one rising edge detection before starting the measuring function; for FSK, the code must wait for four rising edge detections. OOK does not use preamble variables like gu8PreambleCnt, while FSK does. This variable is used in the MDStartListening () function to count the rising edge detections allowed for the preamble stage before reaching the ID bits from the frame.

4   Manchester Encoder

This section describes the devices and elements used to test the Manchester decoding.

4.1   Devices Used

4.1.1   Manchester Encoder

A Manchester encoder was developed in an MC68HC908QY4 with board M68EVB908Q, Rev. 1.1. The general-purpose timer of the 8-bit microcontroller was configured to work in output compare mode. The idea behind the encoder is very simple: the counter has to reach half bit time for 9600-bd signal. After the counter reaches it, a general-purpose output toggles its value. This means that a transition will always occur at the middle of the bit. If a Manchester 0 is desired, the general-purpose output will be initialized low. When the counter reaches the half bit time, it will pass to high and remain there for another half bit period.

The encoder contains definitions and functions for sending every part of the frame scheme in both modulations. All the definitions and higher level functions are based on simple routines MESendZero (uchar times) and MESendOne (uchar times). They send a 0 or 1 the amount of times indicated in the parameter (times) in Manchester encoding. This encoder was ideal to proof the implementation without noise interference.

4.1.1.1   Procedure

The testing process procedure consists of connecting the general-purpose output (PTB3) to the timer channel 0 (PT0) pin of the HCS12X. This pin (PT0) will be working as an input, detecting rising edges and generating interrupts when that happens.

The selection of the modulation is made in the main.h file through defines. To transmit data, the microcontroller will send the frames when a push button connected to PTA0 is asserted. The software of the Manchester encoder already has the configuration set up for the KBI module of the QY4, as well as the interrupt routine for the push button activity. After the frame is sent to the HCS12X, the rest of the process is done by the XGATE/HCS12X CPU.

4.1.2   RF Devices

RF devices have been mentioned several times. This section discusses each device's application note, which describes the use of serial monitors. To be able to work in this way, it is necessary that a microcontroller help the RF devices in the serial communication interface (SCI) communication with the computer. Applications notes AN2777 for Tango3 and AN2818 for Romeo2 explain the process and give examples of how to use these monitors. Tango3 is joined to the DEMO9S08RG60 board that contains an RG60 microcontroller. The Romeo2 is joined to the DEMO908AP64 that contains an AP64. Both RF devices communicate with their respective microcontrollers by the serial peripheral interface (SPI). The microcontroller communicates with the PC through SCI.

Tango3 and Romeo2 monitors allow the user to configure these devices through the HyperTerminal and execute transfers. The results of the transmissions are also displayed in the receiver monitor.
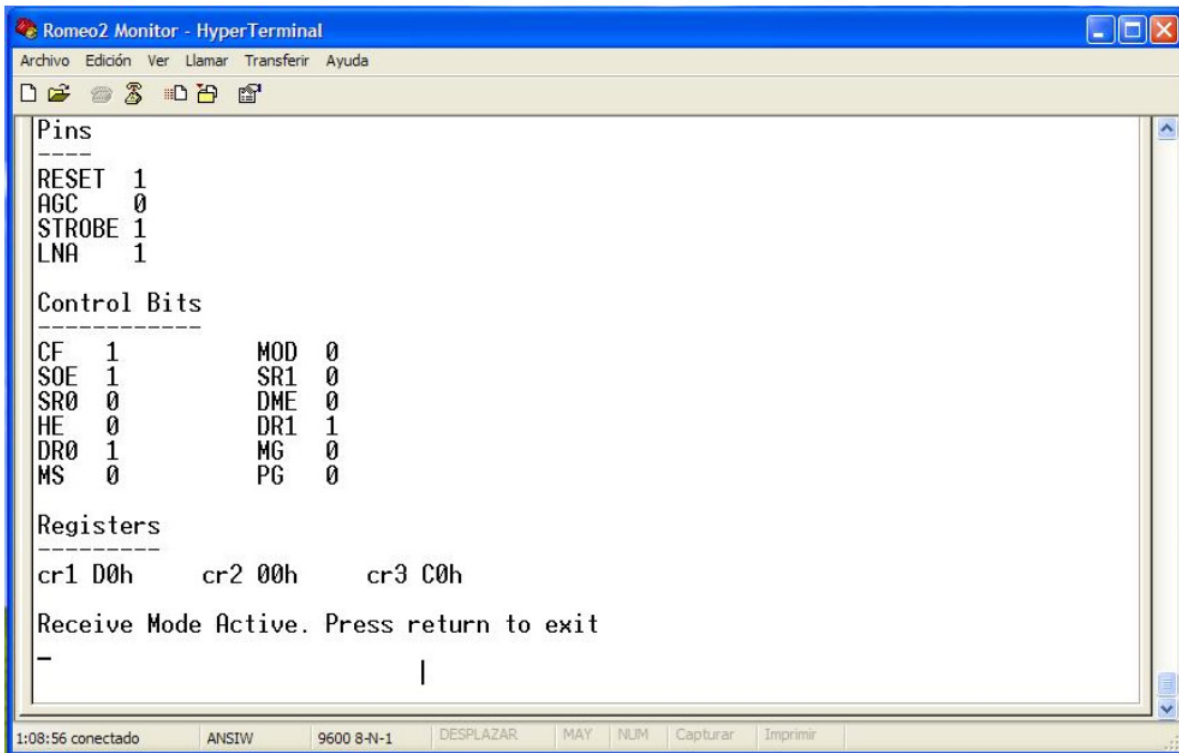

Figure 10. Romeo Monitor

Some changes were made and code added in the Tango3 monitor (transmitter) to work with this implementation:

- Preamble and end of message (EOM) re-definitions: EOM changed from two 0s to two 1s in NRZ. The preamble for OOK changed to two 1s in NRZ, followed by the clock recovery bit that is a 0 in Manchester encoding.

- New commands:
  - APF1: Sends the frame for complete mode with the 10 bits of data as (0011111111).
  - APF2: Sends the frame for complete mode with the 10 bits of data as (1111100000).

  These commands are added in the SCI1Rx. Adding new commands caused changes in the displayed data.

- Send message modifications: In the Tango monitor, there is a function in charge of setting up the timer and sending the correct signal for preambles, end of messages, Manchester-encoded data, or NRZ data. This function is called SendMessage(). In this function, re-definitions of preamble and end of message were made. Also, conditional structures were added to determine whether the test would go to Manchester decoding implementation, or if the monitor would work as usual.

4.1.2.1   Procedure

The frames defined in the Tango monitor are sent by the Tango3 through RF and received by the Romeo2. This data is passed to the 8 bit-microcontroller (AP64) connected to the Romeo2 board in SPI. The MOSI pin in the AP64 board is connected to the timer channel 0 pin (PT0).

Setting up the RF devices' monitors is explained in AN2777 for Tango3 Monitor and AN2828 for Romeo2

Romeo2 (Receiver):

1. CF 1 (434-MHz carrier frequency) <return>.
2. MOD 0 (OOK modulation or 1 for FSK Modulation) <return>.
3. SOE 1 (strobe oscillator enabled) <return>.
4. SR0/1 10 (strobe oscillator ratio - value not important) <return>.
5. DME 0 (data manager disabled) <return>.
6. HE 0 (No header in message) <return>.
7. DR1/0 11 (8.6-10.6kBd data rate range) <return>.
8. MG 0 (normal mixer gain) <return>.
9. MS 0 (mixout pin set to mixer output) <return>.
10. G 1 (phase comparator gain = low gain mode) <return>.
11. Set low noise amplifier (LNA) pin to logic 1 using command LNA 1 <return>.
12. Set STROBE pin to logic 1 using command STROBE 1 <return>.
13. Set Automatic Gain Control (AGC) pin to logic 0 using command AGC 0 <return>.
14. Enable reception of messages using command RECEIVE.

Tango3 (Transmitter)

1. If BAUDRATE is different from 9600, set required baud rate with data rate DATARATE 9600 <return>.
2. To select FSK modulation, set the command MODE 1, for OOK MODE 0 <return>.
3. To enable the Tango3 IC, use the command ENABLE 1 <return>.
4. To send data, use the command APF1 or APF2 <return>.
5. To repeat the Application Frame x (APFx) command, press return repeatedly.

For more information about RF monitors, go to www.freescale.com.

Some main points:

• In the main file of the Tango monitor, there is a section for "Application test defines." Only one of the defines should be not commented. If APPLICATION_ENABLED is not commented, then the monitor can work for this test's project. If APPLICATION_NORMAL is not commented, then the monitor will work normally without the modifications.

Notes:

• The data manager device must be disabled (0) to run those tests. This configuration disables the Romeo2 module that is usually in charge of the decoding process when using only the Tango3 and Romeo2.

- It is important to have common ground between the HCS12X and the Romeo board, or the HCS12X and the QY4.

- For information regarding RF decoding problems while transmitting, see Section Appendix A, "Noise Elements During RF Transmissions in the Manchester Decoding Implementation."

5   Conclusion

The Manchester-decoding implementation can be divided into three main tasks: detection, time measurement, and decoding. In the chosen implementation, all tasks are performed by the XGATE programmable peripheral, completely offloading the HCS12X CPU.

From the results obtained, the CPU loading of the implementation for the XGATE in an average activity was about 20%, leaving the HCS12X CPU able to process other tasks. The same implementation run by the HCS12X CPU would take approximately 40% of CPU loading.

The decoding process can be very demanding in terms of interrupt processing. This only highlights the enhanced performance and flexibility of the XGATE, which was designed to address such issues. In this particular example, the XGATE coupled with a few timer channels can be seen as a Manchester-decoder peripheral from the HCS12X CPU standpoint.

This is just an sample of what the XGATE can do. There are obviously a lot of applications that require intensive interrupt processing from I/Os or peripherals, even if the application itself is not extremely demanding in terms of computing power. The HCS12X architecture featuring the XGATE programmable peripheral meets this kind of embedded system's needs perfectly.

Appendix A Noise Elements During RF Transmissions in the Manchester Decoding Implementation

A.1 Types of Noise

Noise is the biggest hindrance to proper decoding. This implementation dealt with some kind of noise resident in a working area, but depending on the place, noise can change. It wasn't the purpose of this application note to deliver an implementation that could deal with any kind of noise, so this section will briefly explain the conditions and problems found, and ways to avoid noise issues.

In general terms, we can divide detected noise in two classifications. The first kind is background noise caused by the environment, even if the transmitter is turned off or not sending data. This noise may corrupt or hide valid data among a constant chain of noise. The second kind is caused by low RF signal magnitudes, which create glitches, jitters, or variation in the pulse width of the signal.

Both types of noise are considered in this implementation, and there are some workarounds. Here we discuss the effects of noise and ways to avoid it.

A.2 Effects of Noise

A.2.1 Background Noise

Background noise is present in almost any environment. Reflections, electromagnetic devices, and other conditions can create a continuous chain of signals that can affect an efficient decoding. To see the amount of noise in a specific place, turn off the transmitter and let the receiver work as usual. From the Romeo2, we connected the master output slave input (MOSI) pin to the oscilloscope, and the result was the noise of the environment.

The noise received when the Romeo2 was configured for receiving OOK frames was different from the one for FSK frames. FSK noise is more constant and visible.
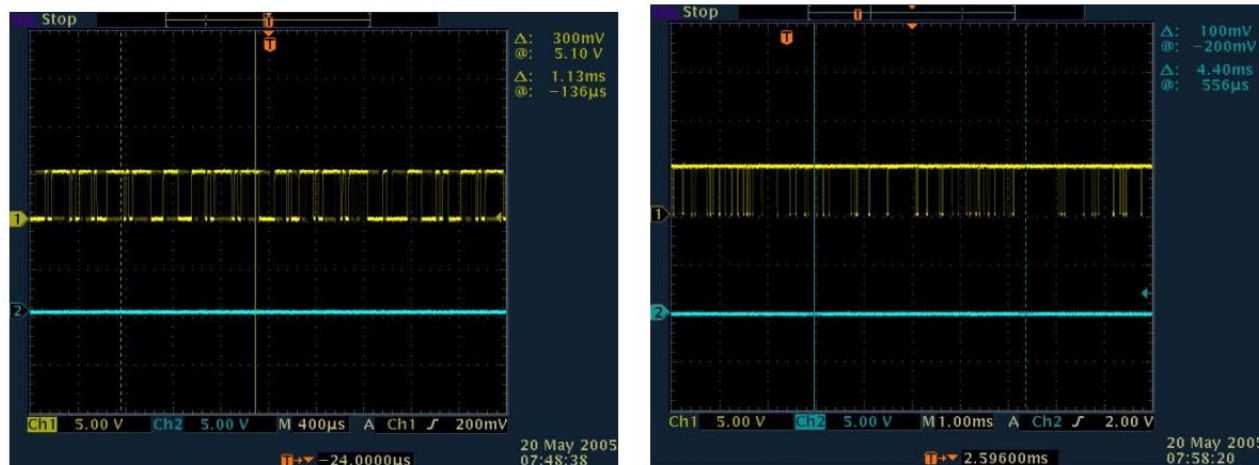
Figure 11. Background Noise in FSK (left) and OOK (right)

The most common problems due to background noise vary depending on the modulation mode. Most of the problems have to do with the interaction between preamble and noise. In other words, detecting the frame from noise was the biggest challenge.

1. In FSK modulation, the valid data frame beginning from the preamble sometimes merged with noise, so instead of having four 0s in Manchester coding, it was usual to find a different number of 0s. Any change in the number of 0s detected in the preamble would cause problems. If fewer were detected, it would never reach the decoding stage and the frame would be lost. If more were detected, then it would pass the preamble stage before the real preamble finished and produce ID errors. When this happened, the rest of the frame were was too.

2. In OOK modulation, most of the issues were caused by easy access of the preamble. In FSK modulation, to access ID or posterior stages, it is necessary to pass through the preamble stage. This stage (MDStartListening () function) demands the sequence of four 0s in Manchester coding. The noise may be abundant, but it is hard to find this exact pattern most of the times. OOK is totally different: the preamble stage just waits for one rising detection, which is very easy. However, for security, the measurement value of this rising and the previous one should be an established amount of time.

A.2.2   Low RF Magnitude Noise

While the emitter moves away from the receiver, the RF signal becomes weaker. This condition causes three main problems:

1. When the emitter is farther, the pulse width of the signal becomes narrower. This means that if the duty cycle used to be near 50%, it will become lower and lower. This will affect the decoding process due to the distance between the rising edges of the signal: they will be more separated and the time counted between them will also increase. If the pulse width is extremely narrow, measurements counted as 1 bit can be taken as bigger, causing decoding errors in the frame. A frame with this kind of pulse might not even access the ID or header stages.

2. When the RF signal magnitude becomes very close to noise magnitude, glitches will start appearing. Any glitch will generate rising edge detection, followed by an interrupt, and finally a bad measurement or decoding.

3. Another effect of a weaker signal is the jitter condition, where the signal seen through an oscilloscope seems to be shaking. This creates a pulse width variation of number 1, which may cause errors in the decoding process and even discard the frame.

A.3   Workaround for Noise Effects

A.3.1   Background Noise

The workaround (already considered in the code) helps to avoid noise. However, there are many options, depending on other circumstances.

1. For the FSK problem of noise merging with the preamble, send a high signal for a given amount of time before the preamble (four Manchester 0s). This way, there is no merging of the preamble and no errors for that condition.
2. For the OOK preamble problem, wait in the MDStartListening () function for a long time before getting the clock recovery.
3. Setting the monitor of the Romeo2 PG to 0 (command PG 0) helps reduce noise because the PLL gain is reduced, thus reducing the signal and noise. In large distances, this may produce loss of the signal.

A.3.2   Low RF Magnitude Noise

1. With duty cycle variation, the use of ranges instead of a specific number helps change duty cycle. Currently the range for each bit value (1, 1.5, and 2) goes from –0.25 bit to +.025 bit. This configuration can easily be changed to make it different.
   a) Another workaround can be done before the decoding process. It consists of sending a frame where the bit timings are present (1, 1.5, and 2) and measuring each one. This way, the real time taken is the basis of the measurement, and this time may be different from the ideal or what is used in normal conditions.
      i. When the signal is weak, send a sequence (000000000 in Manchester) containing only time measurements that are supposed to be 1 bit, measure them, and take the average.
      ii. The process is the same with 1.5 bits (011001100110 in Manchester) and 2 bits (0101010101 in Manchester). Now we have the real timing for each case.
         1. REALTIME_1_BIT
         2. REALTIME_1_5_BIT
         3. REALTIME_2_BIT

Example if using the values obtained from measuring before having established values for bits:

```
/* Compare of measured times with fixed values */

if  (gu16TimeMeasured <  (0.75 * REALTIME_1_BIT) ){

gu8TimeType = LESS;

gu8FlowControl = 0;

} if (gu16TimeMeasured >= (0.75 * REALTIME_1_BIT)){
```

```
gu8TimeType = ONE_BIT;

gu8FlowControl = 1;

} if (gu16TimeMeasured >= ((REALTIME_1_BIT + REALTIME_1_5_BIT)/2)) {

gu8TimeType = ONE_AND_HALF_BIT;

gu8FlowControl = 1;

} if (gu16TimeMeasured >= ((REALTIME_1_5_BIT + REALTIME_2_BIT)/2)) {

gu8TimeType = TWO_BITS;

gu8FlowControl = 1;

} if (gu16TimeMeasured > BIT_4_5){

gu8TimeType = INITIAL;

gu8FlowControl = 0;

}
```

  b) For glitches in the MDTimeMeasuring () function, a measurement of less than a bit gets discarded or sent to MDIdlePosition (). If another timer channel would detect falling edges, the glitch process would be to get the values between a rising and a falling edge. This method (adding falling edge detection) may be more complete and can deliver more decoding opportunities to the application, but the loading due to constant interrupts is also higher.

This page intentionally left blank.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN3015
Rev. 0, 10/2005

freescale™
semiconductor