OVERVIEW

All KEELOQ Encoders use a common code word format. They all pack the bits of information in the same number and position in the transmission sequence. While all the encoders offer a basic Pulse Width Modulation (PWM) method to produce a signal that is suitable for radio transmission, many Advanced KEELOQ Encoders (HCS360 and higher) offer alternative methods. One such alternative method is Manchester encoding. This Technical Brief shows PICmicro® microcontroller assembly routines to receive Manchester encoded transmissions.

MANCHESTER ENCODING

In Manchester Encoding, as in PWM, clock and data are encoded in a single synchronous bit stream. In this stream, each bit is represented by a transition. If the bit is a '0', the transition is from low to high. If the bit is a '1', the transition is from high to low (see Figure 1).

![Manchester Encoding Diagram](image)

In a typical data stream, there will always be a transition at the center of a bit (A), while at the beginning of a bit there will be a transition depending only on the value of the previous bit (B). The encoding may be alternatively viewed as a phase encoding where each bit is encoded by a positive 90 degree phase transition, or a negative 90 degree phase transition. Manchester code is therefore sometimes known as a bi-phase code.

A Manchester encoded signal contains frequent level transitions which allow the receiver to extract the clock signal easily and reliably. The penalty for introducing frequent transitions, is that the Manchester coded signal consumes more bandwidth than the original signal (sequence of logic ones and zeros or NRZ) but it still compares well with the bandwidth requirements of other encoding systems, such as PWM.

A MANCHESTER RECEIVER

The role of a Manchester receiver is that of separating the clock information from the transmission stream in order to properly extract the data. This can be done in hardware by means of a Digital Phase Locked Loop (DPLL) circuit or by means of software techniques, as we will show in the following sections.

Many different approaches to the problem are possible. This Technical Brief has the sole purpose of offering the reader a starting point for the creation of an interrupt-based Manchester receiver/decoder suitable for use with KEELOQ Encoders.

KEELOQ MANCHESTER FORMAT

When operating in the Manchester mode, Advanced KEELOQ Encoders build the code-word with a common specific sequence. The elementary period (T_E) will be used in the following as the measurement unit. T_E will vary from 100 µs to 800 µs according to the selected baud-rate.

As shown in Figure 2, the code-word is composed of:

- Preamble, consisting of 32 transitions (32xT_E)
- Synchronization Header, a pause low of 4xT_E duration
- Start bit, consisting of logic 1-bit encoding (2xT_E)
- Packet of 65+ bits, the actual data bits (N x 2xT_E)
- Stop bit, consisting of a logic 1-bit (2xT_E)
- Guard Time, a pause before the whole code word is repeated again (length can vary with encoder models).
FIGURE 3: MANCHESTER TRANSMISSION FORMAT

INTERRUPT RECEIVE TECHNIQUES

Interrupts are available on all Microchip 14-bit and 16-bit core PICmicro microcontrollers. Interrupts can be used to implement an efficient receiver for KeeLoq® Manchester Encoded transmissions.

The example routine presented in Appendix A is based on a simple principle. Timer0 is used to generate a constant period interrupt multiple of the desired baud rate Te. The multiplying (oversampling) factor RF_OVERS is a user configurable parameter. The value of the parameter (typically in the range of 4 to 8) should be selected as high as possible with the understanding that the higher values will provide more accuracy and flexibility, while causing the interrupt routine to use a larger portion of the CPU processing power.

While the sampling techniques employed in polling and interrupt receivers do not differ much, the whole architecture of a KeeLoq receiver/decoder contains certain advantages and disadvantages. Interrupt-driven receive routines work in the background, while the user main code executes, just as a hardware peripheral would. The only interface between the interrupt service routine and the main program are the buffer that holds the received data (Buffer) and the flag that signals the completion of the receiving process (RF_FULL). A significant advantage is that the code is cleaner and is theoretically easier to maintain.

The main disadvantage of interrupt receiving routines is reduced flexibility. The interrupt mechanism is actually stealing processing power from the CPU, and resources from the microcontroller (Timer0). Therefore, the routine length and complexity must be reduced to a minimum.

Sharing resources with the interrupt routine is possible, but requires some attention. For example, the main program is not allowed to write/reset Timer0. However, it is possible to make an effective use of it and the interrupt timing itself to derive multiple software timers for use in the main loop. Enabling other interrupt sources, adding latency to the receiver interrupt servicing, might cause the receiver sampling point to be misplaced and is therefore to be considered very carefully. No timing can be achieved in the Main Loop by means of simple counters (loops) since the interrupt mechanism will interfere slightly randomizing the loops duration.

These recommendations are common to all the applications that make use of interrupts.

CODE DESCRIPTION

The Interrupt Service Routine (ISR) samples the radio receiver digital output and looks for transitions. The ISR operates as a simple state machine capable of four states (see Figure 4).

The sampling rate is determined by the continuous reloading of Timer0 (with TIMER_VALUE). It provides a fixed time base to schedule the execution of one of the state routines according to the value of the variable RF State. Each of these state routines is responsible for the advancement of the process to a previous or subsequent state according to a set of rules that determine the actual receiver working mechanism. Each state routine is responsible for loading a skip counter (RFSkip) with a value of one, to make the execution flow continuous. The skip counter may be loaded with a value larger than one to add a delay and postpone the execution of the next (same) state routine by a multiple of the interrupt timer period, when required.
The following is a brief description of the workings of the four state routines.

**State 0** (label TRFReset) is used at start up and whenever an error or time-out occurs in any of the previous states. Its role is simply that of repositioning the data buffer pointer to the top of the buffer, and clearing the bit counter.

**State 1** (label TRFSynch) is where the receiver waits for the SyncHeader (4 x Te pause) after the Preamble and derives from it the value of Te (actually 1.5 x Te) expressed in terms of interrupt periods count. This is essentially the way the receiver tries to recalibrate with the transmitter’s actual clock frequency, compensating for the ample variations possible due to temperature and battery voltage changes on the transmitters.

**State 2** (label TRFTDetect) is where the receiver waits for a transition to happen. This state is critical to the clock extraction process, since the synchronization of the data stream depends on the constant realignment of the receiving process with the input signal transitions. When an edge is detected, a delay of 1.5 x Te is set (RF_Skip = 1.5 x Te) before entering state (3) in order to reach an optimal bit sampling point.

**State 3** (label TRFSample), is where the actual data bit is extracted. The action of state ‘1’ combined with the delay of 1.5 x Te ensures that data sampling occurs at the ideal place, i.e., at the center of the first half of a bit period (see Figure 5).

The data bit is then rolled in the buffer and the bit counter advanced. When the required number of bits (RF_NBITS) has been received, a flag is set (RF_Full) and the receiver state machine loop idles, making the receiver data buffer (Buffer) available to the main program for decoding.

In order to re-enable the receiver, the main program resets the same flag (RF_Full) releasing control of the receive data buffer.

To complete the Interrupt Service Routine there are a few lines of code dealing with the context saving (at the top) and restoring (at the bottom). This code can be processor dependent, meaning that different models of PICmicro MCUs might actually require modifications to these portions. It is recommended that the reader refers to the specific PICmicro microcontroller datasheet for the suggested implementation of such code.

The InitRX routine shows an example of the portion of the PICmicro MCU initialization code required for the receiver start up, where the interrupt services are enabled and the timer options are selected.
PERFORMANCE

The code presented in the appendix shows an implementation for a clock of 20 MHz and a baud rate corresponding to a TE value of 200 \(\mu s\). The following formula is used to determine the interrupt timer period:

\[ \text{TIMER\_VAL} = \left( \frac{\text{XTAL} \times \text{BRATE}}{4 \times \text{RF\_OVERS}} \right) \]

Changing the parameters \(\text{XTAL}, \text{BRATE}, \text{RF\_OVERS}\) to fit different application requirements allows the user to adapt the code to different clock speeds and baud rates within the constraints of the available processing power. It is good practice to keep the percentage of time the processor spends in Interrupt Service Routines (ISR) to a minimum:

\[ \text{Pint}\% = \frac{\text{Time spent in ISR}}{\text{second}} \times 100 \]

or

\[ \text{Pint}\% = \frac{\text{ISR length}}{\text{Interrupt timer period}} \times 100 \]

Such a percentage determines the equivalent average clock speed of execution for the main program (running in foreground) as per the following formula:

\[ \text{FEQ} = \frac{\text{FOSC} \times (1 - \text{Pint}\% / 100)}{\text{FEQ} \text{ approaches zero}, \text{affecting the responsiveness of the whole application.}}\]

The absolute limit \(\text{Pint}\% = 100\), is actually reached when the interrupt timer period (determined by \(\text{TIMER\_VAL}\)) becomes equal or shorter than the Interrupt Service Routine length (about 60 instruction cycles in this case). In this situation, a new call to the interrupt service routine is made as soon as the processor returns from the previous one, and there is no time left to execute the main program.

When selecting values for a specific application, typically the baud rate will be given, as well as a maximum desired clock speed. The oversampling factor can be used then to optimize the Pint%.

Refer to Table 1 for some possible values:

<table>
<thead>
<tr>
<th>XTAL</th>
<th>20 MHz</th>
<th>10 MHz</th>
<th>8 MHz</th>
<th>4 MHz</th>
<th>4 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRATE</td>
<td>200 (\mu s)</td>
<td>400 (\mu s)</td>
<td>800 (\mu s)</td>
<td>800 (\mu s)</td>
<td>400 (\mu s)</td>
</tr>
<tr>
<td>RF_OVERS</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>TIMER_VAL</td>
<td>125</td>
<td>125</td>
<td>200</td>
<td>133</td>
<td>100</td>
</tr>
<tr>
<td>Pint%</td>
<td>32%</td>
<td>32%</td>
<td>20%</td>
<td>30%</td>
<td>40%</td>
</tr>
</tbody>
</table>

**Note:** It is important to verify that the value derived from the formula for \(\text{TIMER\_VAL}\), being a period for an 8-bit timer (Timer0), must be smaller than \((2^8 = 256)\). Compromising with the other parameters allows the user to obtain low use of CPU power (ideally Pint% = 40% or less), while achieving a satisfactory receiver performance.

MEMORY USAGE

Program Memory Words Used: 133
File Registers Used: 22

REFERENCES

- Secure Learning RKE Systems using KEELOQ Encoders  
  TB001 DS91000
- An Introduction to KEELOQ Code Hopping  
  TB003 DS91002
- KEELOQ CRC Verification Routines  
  TB043 DS91043
- Modular PICmicro Mid-Range MCU Code Hopping Decoder  
  AN742 DS00742
- Modular Mid-Range PICmicro KEELOQ Decoder in C  
  AN744 DS00744

KEYWORDS

KEELOQ, Manchester, Mid-Range, Receiver, Interrupt
APPENDIX A: SOURCE CODE

;**********************************************************************
;  Manchester encoding interrupt driven receiver routines
;  Uses: interrupt on TMR0
;  Accepts: Keeloq code words Manchester encoding
;  VERSION     1.00    1/16/01     Lucio Di Jasio
;**********************************************************************

#define XTAL .20      ; clock frequency (MHz)
#define BRATE .200     ; Te = 200 (us)
#define RF_OVERS .8    ; oversampling factor (4..8)

TIMER_VAL equ (XTAL * BRATE) / (4 * RF_OVERS)

IF (TIMER_VAL > .256) || (TIMER_VAL < .60)
    error "ERROR: Timer period exceeds limits"
ENDIF

; bit timing limits
#define SHORT_HEAD .3*RF_OVERS     ; minimum synch header length accepted
#define LONG_HEAD .6*RF_OVERS      ; maximum synch header length accepted
#define HIGH_TO .3*RF_OVERS        ; bit timeout
#define RF_NBITS 69               ; number of bits to capture

CBLOCK 0x20                  ; omit the address (0x20) if this include module is
        ; not the first to allocate RAM
    W_TEMP        ; context saving W reg.
    STATUS_TEMP  ; "    "    STATUS reg.
    PCLATH_TEMP  ; "    "    PCLATH reg.
    FSR_TEMP     ; "    "    FSR reg.
    RFP          ; receive buffer pointer
    RFbitc       ; bit counter
    RFSkip       ; skip sample counter
    RFsamp       ; samples counter
    RFSState     ; state variable
    RXFlags      ; various flags
    Te           ; time unit = 1/baud rate

    Buffer0:9    ; receive data buffer

Software License Agreement
The software supplied herewith by Microchip Technology Incorporated (the “Company”) for its PICmicro® Microcontroller is intended and supplied to you, the Company’s customer, for use solely and exclusively on Microchip PICmicro Microcontroller products. The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
; add here extra software timers
XTMRH ; example 16 bit software timer
XTMRL
ENDC

; radio input
#define RFIN PORTB,0

; flags
#define RF_Full RXFlags,0 ; buffer full, receiver idle
#define LastBit RXFlags,1 ; used for transition detection

;//------------------------------------------------------------------------------------------
; Async_ISR
; ; NOTE: place this routine at loc. 0x04
;       context saving can be specific to the PICmicro model used
;       (ref. to PICmicro datasheet for suggested implementation)
;
; IntVector
movwf  W_TEMP ; context saving, save W first
swapf  STATUS,W
bcf    STATUS,RP0 ; assuming use of variables in bank0
movwf  STATUS_TEMP ; save status
movf   PCLATH,W
movwf  PCLATH_TEMP ; save PCLATH
clf    PCLATH ; assuming Async_ISR located in page0

;//------------------------------------------------------------------------------------------
Async_ISR
movlw  TIMER_VAL ; non disruptive timer reload
subwf  TMR0,F
bcf    INTCON,T0IF ; interrupt served
btfsc  RF_Full ; if buffer still full
goto   AsyncRFE ; idle (exit immediate)
decfsz RFSkip,F ; count the skips
goto   AsyncRFE ; delay (exit immediate)
movf   RFState,W ; launch appropriate state routine
andlw  03 ; limit to [0..3]
addwf  PCL,F ; offset in jump Table
RFTable
    goto TRFReset ; 0
    goto TRFSync  ; 1
    goto TRFTDetect ; 2
    goto TRFSample ; 3
RFTableEnd

    IF HIGH(RFTable) != HIGH(RFTableEnd)
        error “RFTable crosses page border”
    ENDIF

;//------------------------------------------------------------------------------------------
; State 1
; ; Waiting and measuring a Synch Header
; TRFSync
btfsc RFIN ; wait for a rising edge
goto TRFRise
incf RFsamp,F ; while input low count the samples
movf RFsamp,W
btfsc STATUS,Z
goto TRFReset ; check overflows
goto AsyncRFE ; remain in state 0

TRFRise
movlw SHORT_HEAD ; check minimum header length
subwf RFsamp,W
btfss STATUS,C
goto TRFReset ; proceed to reset if too short
movlw LONG_HEAD ; check maximum header length
subwf RFsamp,W
btfsc STATUS,Z
goto TRFReset ; proceed to reset if too long

TRCalibra
bcf STATUS,C ; RFsamp measure THeader
rrf RFsamp,F ; divide by two
bcf STATUS,C
rrf RFsamp,W ; divide by four
movwf Te ; Te = 1/4 THeader
movwf RFsamp
bcf STATUS,C
rrf RFsamp,W ; RFsamp = 1/8 THeader
addwf Te,F ; Te = 1/4THeader + 1/8THeader (that is 1.5xTE)
ingf Te,F ; round it up

TRinit
bsf LastBit ; init for detection of falling edges
crff RFsamp ; reset sample counter
incf RFState,F ; move on to transition detection(2)
ingf RFSkip,F ; skip=1
goto AsyncRFE ; done

;---------------------------------------------------------------
; State 2
;
; Transition Detection
;
; TRFTDetect
btfsc LastBit ; depending on last value of input
goto TRFUSet

; last bit = 0
btfsc RFIN ; detect a transition
goto TRFTransition
goto TRFNOTTransition

; last bit = 1
TRFUSet
btfss RFIN ; detect a transition
goto TRFTransition
goto TRFNOTransition

TRFTransition ; transition detected
  incf RFState,F ; move on to Sampling (State 3)
  clrf RFsamp
  movf Te,W ; skip = 1.5xTE
  movwf RFSkip
  goto AsyncRFE ; done

TRFNOTransition ; no transition detected
  incf RFSkip,F ; skip = 1
  incf RFsamp,F ; keep counting time between transitions
  movlw HIGH_TO ; check against Timeout value
  subwf RFsamp,W
  btfss STATUS,C ; if timeout fall through Reset
  goto AsyncRFE ; done

;------------------------------------------------------------------
; State 0,
;
; Reset receiver
;
TRFReset
  movlw 1
  movwf RFState ; move on to TRFSync (1)
  clrf RFsamp ; reset sample counter
  movlw Buffer0 ; reset buffer pointer
  movwf RFP
  clrf RFbitc ; reset bit counter
  incf RFSkip,F ; skip = 1
  goto AsyncRFE ; done

;------------------------------------------------------------------
; State 3,
;
; Data Sampling
;
TRFSample
  incf RFSkip,F ; skip = 1
  decf RFState,F ; next will be TDetect again (2)
  bcf LastBit ; update LastBit for next transition
  btfsc RFIN
  bsf LastBit
  movf FSR,W ; save FSR
  movwf FSR_TEMP
  movf RFP,W ; load current pointer
  movwf FSR ;
  bcf STATUS,C
  btfsc RFIN ; copy data bit in CARRY
  bsf STATUS,C
  rrf INDF,F ; roll in the buffer
  movf FSR_TEMP,W ; restore FSR
  movwf FSR
  incf RFbitc,F ; count the bit
movlw 07              ; every 8 bit
andwf RFbitc,W
btfsc STATUS,Z
incf RFP,F           ; increment the pointer
movf RFbitc,W        ; check all bit in already
xorlw RF_NBITS
btfss STATUS,Z
goto AsyncRFE        ; not yet, done
TRFFull                             ; received them all
movlw TRFReset        ; next state will be Reset
movwf RFState
bsf RF_Full         ; set the buffer full flag
goto AsyncRFE        ; done

AsyncRFE
;---------------------------------------------------------------------
; Example: how to use the interrupt period to derive more software timers
; showing a 16 bit TIMER that ticks at every interrupt period
; 1 tick (us) = 4 x TIMER_VAL / XTAL (MHz)
;
; incf XTMRH,F         ; update 16 bit timer
incfsz XTMRL,F
decf XTMRH,F
;
;---------------------------------------------------------------------
; Context Restore
;
ExitInt
movf PCLATH_TEMP,W   ; restore PCLATH (page)
movwf PCLATH
swapf STATUS_TEMP,W  ; restore status and bank
movwf STATUS
swapf W_TEMP,F       ; restore W reg.
swapf W_TEMP,W
retfie               ; exit re-enabling interrupts

;---------------------------------------------------------------------
; Init Receiver
;
; Example code for proper interrupt and timer set up
;
InitRX
clr rf RXFlags
clr rf INTCON         ; disable interrupts
clr rf RFState        ; init with Reset state
clr rf XTMRH          ; init demo software timer
clr rf XTMRL
movlw 1
movwf RFSkip          ; init skip
bsf STATUS,RP0        ; bank 1
movlw b'10001111'     ; init timer prescaler 1:1
movwf OPTION_REG      ; prescaler assigned to WDT 1:128
bcf STATUS,RP1        ; bank 0
movlw -TIMER_VAL      ; init timer
movwf TMRO
bsf INTCON, T0IE ; enable interrupts on TMR0 overflow
bsf INTCON, GIE  ; global interrupt enable
return
“All rights reserved. Copyright © 2001, Microchip Technology Incorporated, USA. Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights. The Microchip logo and name are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. All rights reserved. All other trademarks mentioned herein are the property of their respective companies. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.”

**Trademarks**

The Microchip name, logo, PIC, PICmicro, PICMASTER, PICSTART, PRO MATE, KEELQ, SEEVAL, MPLAB and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Total Endurance, ICSP, In-Circuit Serial Programming, FilterLab, MXDEV, microID, FlexROM, fuzzyLAB, MPASM, MPLINK, MPLIB, PICDEM, ICEPIC, Migratable Memory, FanSense, ECONOMONITOR, Select Mode and microPort are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Term Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies. © 2001, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.