AN008
Oversampling and data decision for the CC400/CC900

By K. T. Heien, T. A Lunder and K. H. Torvmark

Keywords

- *Oversampling*
- *Data decision*
- *Data synchronisation*
- *Microchip PIC*
- *Software RSSI / signal quality*
- *Resynchronisation*
- Software squelch

Introduction

One of the most important issues affecting the implementation of microcontroller software deals with the data-decision algorithm. Data-decision refers to decoding the DIO-pin from the CC400/CC900. Two main principles exist for decoding Manchester-coded data: Data decision based on timing the period between transitions, and data decision based on oversampling.

Decoding based on measuring the time between transitions is easy to implement but suffers in performance when used on a noisy signal. This has to do with the noise characteristic of the demodulated RF-signal. When a receiver reaches the sensitivity limit, the received data will contain “spikes” or short transients due to noise. If the detection is based on finding the time period between changes in the data level, decoding errors will occur because of these spikes. By using oversampling, taking multiple samples of the same bit, this noise can filtered out by using a majority vote decision.

This application note describes oversampling and data decision in detail. The principles are illustrated in flowcharts and C-code. This application note also contains optimized assembly code written for the Microchip PIC16F87x series of microcontrollers.

Chipcon is a supplier of RFICs for all kinds of short range communication devices. Chipcon has a world-wide distribution network.

Table of Contents

Why use oversampling?

The figure below illustrates the principle of oversampling. A data stream of Manchester-coded data (110) is transmitted. In Manchester code a logical 1 is coded as a high to low transition in the middle of the bit period, and a logical 0 is coded as a low to high transition in the middle of the bit period. This is done to ensure a constant DC-level. As illustrated in the figure each bit has a transition. Each level in one bit is called a chip, giving two chips per bit. Therefore, the baud-rate is twice the bit rate for Manchester coded data.

The received signal with noise illustrates how the noise can corrupt the data received when the receiver is near the sensitivity limit. This noise is often concentrated near the edges of each chip. Here the noise is illustrated as short transients, but there will also be some duty-cycle error and some jitter when the receiver is near the sensitivity limit.

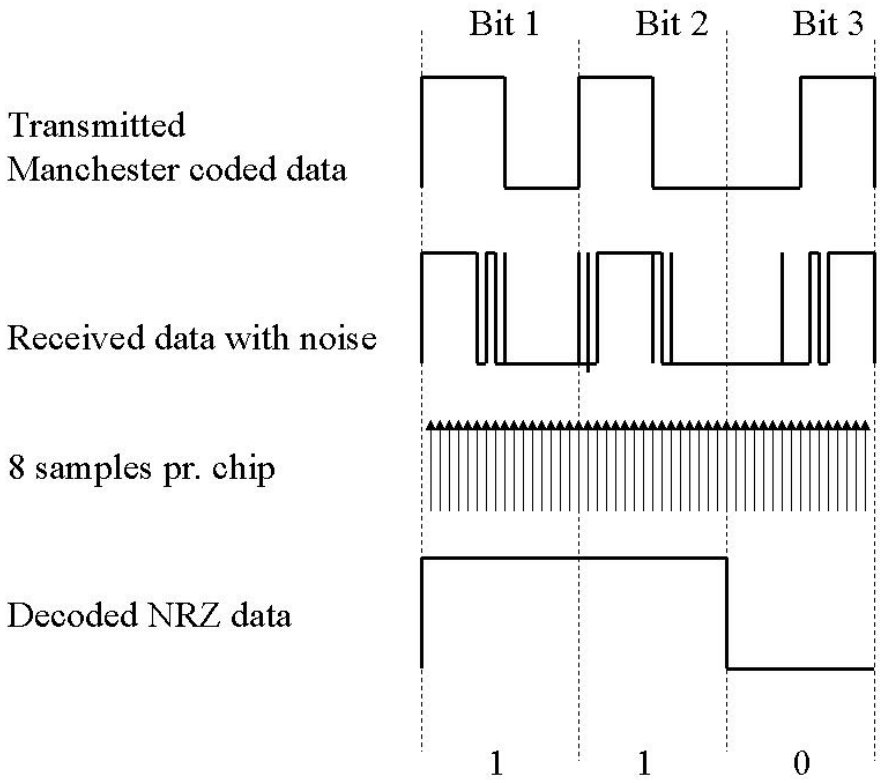


Figure 1 Principle of oversampling

By taking several samples per chip, the noise is filtered out and the decoded NRZ data recovered is error-free. This would not be the case if the decision had been made from measuring the time between level changes. The noise would have corrupted the resulting decoded data. For this reason we recommend implementing oversampling in the decision algorithm for best overall performance.

Microcontroller limitations

The main factor influencing the implementation of the oversampling routine is the speed of the microcontroller. The total number of machine cycles available for decoding the data is given as:

$$MachineCodePerBit = \frac{MCU_{ClockSpeed}}{Datarate}$$

If you use a 1.2 kbps data rate, and your microcontroller runs at 4MIPS (million instructions per second), you can use up to 3300 instructions per bit. Doing 4 samples per chip, or 8 samples per bit, gives 3300/8 = 416 instructions per sample.

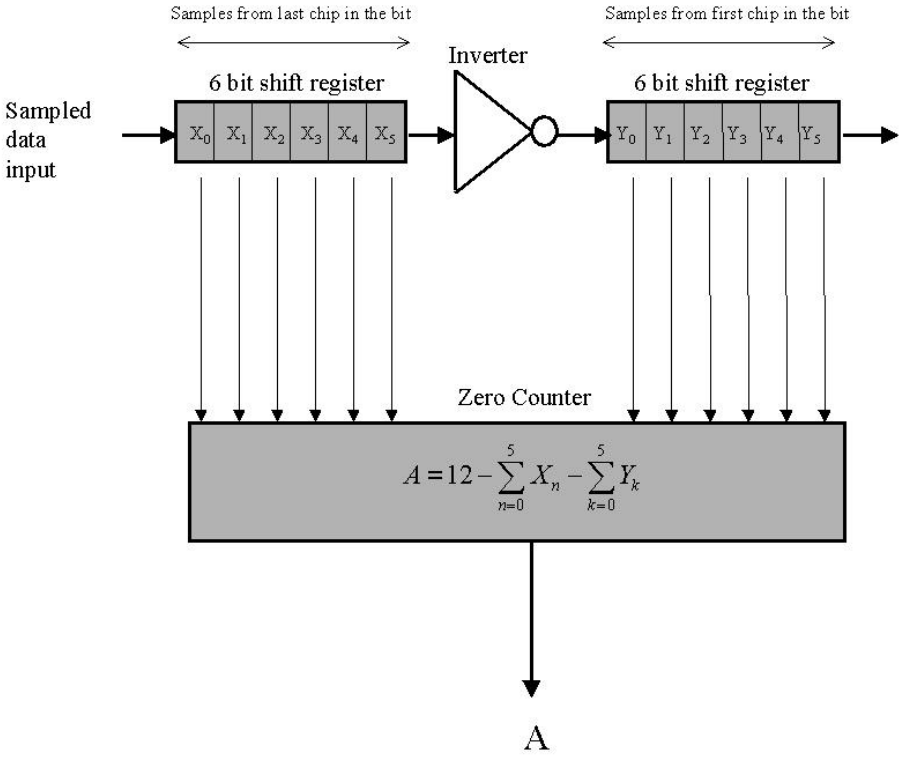
For most microcontrollers, speed is at a premium, and the decoding routine must be as efficient as possible. In most cases, an optimized assembly code routine should be used.

Data decision and synchronisation

There are different ways to decode the sampled data using oversampling. The easiest way is to count the number of high-level samples and the number of low-level samples. The level of the chip can then be decided by majority vote.

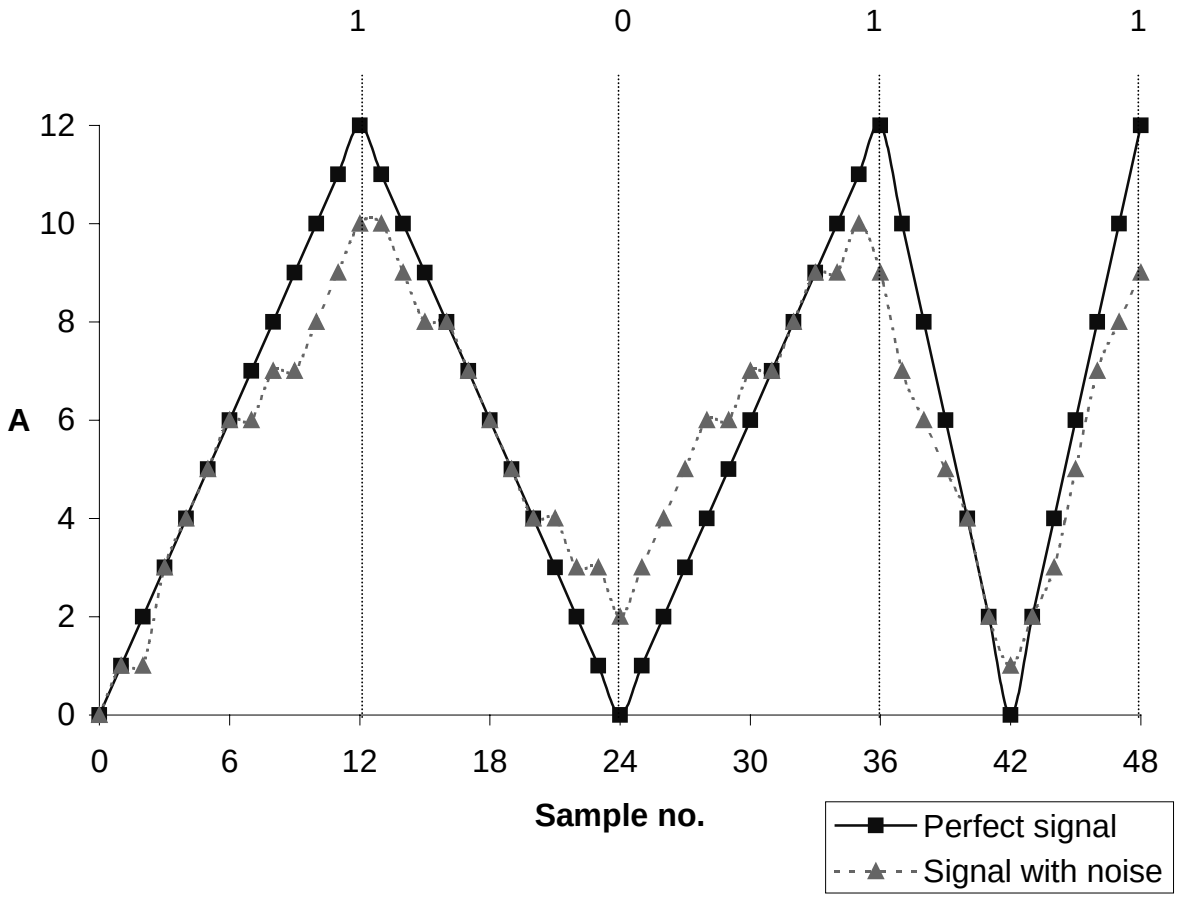
Since noise is more likely to appear near the edges of each chip, the algorithm can be improved by giving the samples in the middle of each chip higher priority than the samples near the edges.

Due to the fact that the received signal is Manchester coded, a correlation algorithm can be used to synchronise and decode the sampling. A simple implementation is illustrated below. In this example we use 6 samples per chip.



This correlation algorithm takes 6 samples per chip. Each new sample goes into a shift register, and the previous samples are shifted to the right. The inverter between the two 6 bit shift registers is used because of the Manchester coding of the data. For each new sample the algorithm calculates A (correlation score). This value provides synchronisation information and an estimate of the bit value.

To illustrate this, let us take an example. If the value in the shift register is a logical '0' and the received bit stream is '1011', then value A will change as shown in the figure below.



As illustrated the value of A can vary between 0 and 12 when using 12 samples per bit. Since A is calculated for each sample, A can be used to determine synchronisation status. When noise is present, the change in A will not be linear, however. Due to the noise some samples will have level error, giving higher values for logical '0' and lower values for logical '1'. The maximum and minimum levels of A will still give the same information about the logical level and synchronisation.

When using Manchester-coding, the signal can change value both at the start of the bit as well as in the middle. The software must "know" which transition is which. To be able to do this, the software must be synchronised to a known bit pattern. The usual way to do this is to send a preamble before the data is transmitted. The receiver can then make a synchronisation on this preamble to get the timing correct before the data decision starts.

Using this correlation algorithm, it is easy to implement higher priority for the samples in the middle of the chip. This can be done by multiplying the value of the samples that are most

likely to be correct with a fixed value before calculating A. Further samples at the edges of each chip can be ignored in the calculation of A, to remove the uncertainty of the samples where the noise is most likely to appear. By implementing a correlation algorithm in software, it is easy to find the best noise reduction filtering parameters for the application.

Resynchronisation

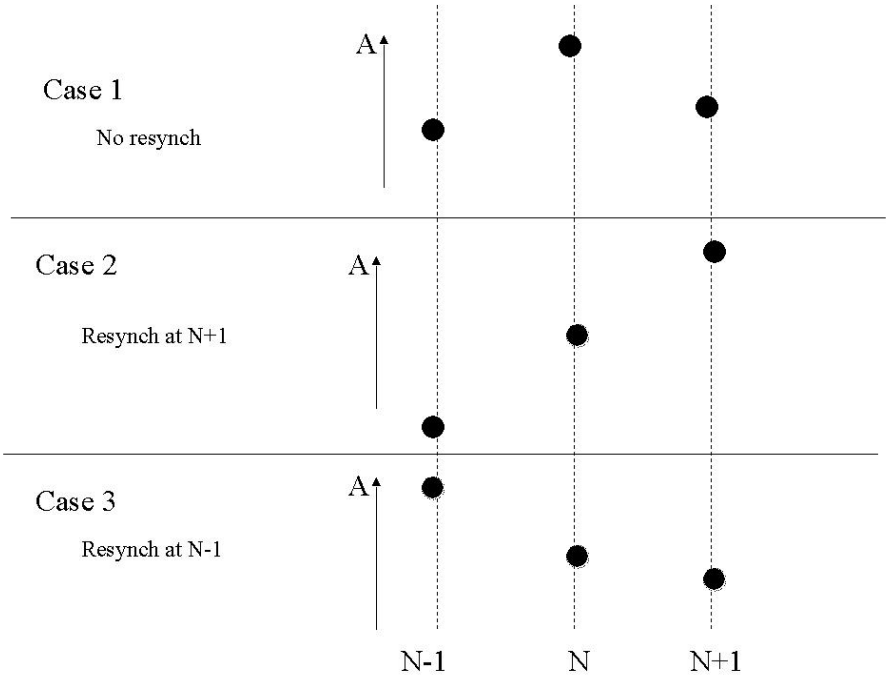
As explained above, the software algorithm needs to synchronise with the signal before the correlation algorithm can calculate valid values for A. Depending on the length of the data stream and the accuracy of the crystal, the sampling clock in the receiver will drift in comparison to the transmitter clock. This means that the synchronisation made at the beginning of the data stream not necessarily is correct at the end. The total drift of the data is given by:

$$\Delta T = T_M \bullet XTAL_{Accurancy}$$

Where T_M is the length of the message, and $XTAL_{Accurracy}$ is the relative accuracy of the receiver and the transmitter. That means that if the crystals for both the transmitter and the receiver are rated at +/- 10 ppm, then $XTAL_{Accurancy}$ is 20 ppm as a worse case.

If the total drift of the data (ΔT) is higher than the sampling rate, the correlation algorithm is not synchronised perfectly, and will not calculate the correct value of A even for a perfect signal. This means that when $T_s < \Delta T$, a resynchronisation must be performed before proceeding with the data decision. T_s is the sampling clock in the receiver.

There are many different ways of implementing a resynchronisation algorithm. A convenient method is to use the value of A in the correlation algorithm to decide if a resynchronisation is needed. The idea here is to resynchronise to the most likely sample near the bit decision, that is, at the maximum (logical 1) or minimum (logical 0) of A. To illustrate this, let us look closer to the value of A at different samples around a maximum were logical 1 is detected. This is illustrated for tree cases in the figure below.



At N the bit is decided based on the value of A. After the sample N+1 the algorithm must check if a resynchronisation should be performed before proceeding with the next bit. In the figure the value of A is illustrated at the sampling time N-1, N, and N+1 for three different cases. For all cases the data decision is done at N.

For case 1 we see that A has a lower value at sample N-1 and N+1 than for sample N. This means that the bit decision is done on the maximum of A, and the software does not need to make a resynchronisation.

For case 2 the value of A is higher for the sample N+1 than the case is for sample N. Also, the value of A at N-1 is lower than for the sample N. This means that the maximum of A is more likely to take place after the bit decision sample N. The correlation algorithm should be resynchronised to the sample N+1 before continuing.

Case 3 is very similar to case 2, but here the correlation algorithm should be resynchronised to sample N-1, since the maximum value of A is likely to take place before N.

There is also one other possibility that is not illustrated in the figure. If the value of A at bothN+1 and N-1 is higher than the value at N, then the algorithm does not know if it should resynchronise to an earlier or later sample. For this reason it should not do resynchronisation here, but wait until the next bit decision sample later in the message.

The description of the resynchronisation above is based on the samples near the bit decision samples for logical 1 (maximum of A). A similar calculation must be implemented for the decision of logical 0 (minimum of A). An overview of the possible cases for this simple resynchronisation algorithm is given in the table below.

Table 1 Resynchronization criteria

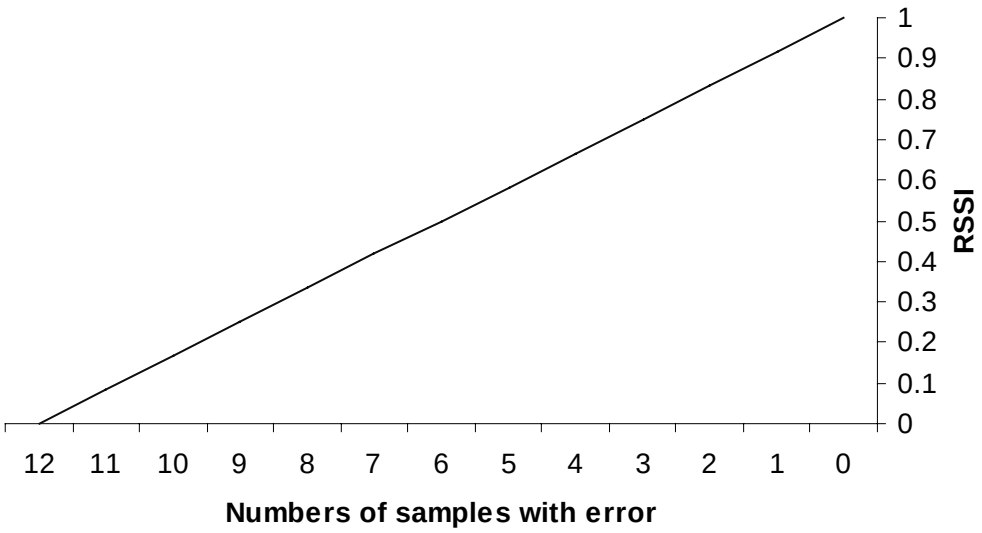
Logical level at N	Resynch to N-1	No Resynch	Resynch to N+1
1	$A_{N-1} > A_N > A_{N-+}$	$A_N > A_{N-+}$ AND $A_N > A_{N-1}$ OR $A_N < A_{N-+}$ AND $A_N < A_{N-1}$	$A_{N-1} < A_N < A_{N-+}$
0	$A_{N-1} < A_N < A_{N-+}$	$A_N > A_{N-+}$ AND $A_N > A_{N-1}$ OR $A_N < A_{N-+}$ AND $A_N < A_{N-1}$	$A_{N-1} > A_N > A_{N-+}$

Software RSSI / Signal Quality

With the information of the correlation value A, the software also has the opportunity to implement a kind of RSSI. By sending a preamble of only logical 1s, the signal quality on this received preamble can be calculated as:

$$RSSI = \frac{A_{noise}}{A_{perfect}}$$

Here $A_{perfect}$ is the correlation score for a perfect signal, and A_{noise} is the correlation score on the actual received data. If we use the correlation algorithm described above, the value of $A_{perfect}$ is 12 for a perfectly received logical 1. Depending on the number of samples with errors, the "RSSI" value will change as in the figure below.



This RSSI level can be used in a signal quality algorithm to implement a type of software based RSSI giving a measure of the quality of the received data. One use of this RSSI is to reduce the transmitted power and thus reduce the overall system power consumption.

Software squelch

When using Manchester coded data, keeping track of how many samples are equal during a bit can be used to implement software squelch. This is useful if the transmitter does not transmit continuously. In a proper Manchester coded signal, the two chips in a bit should be unequal. If most of the samples in a bit are equal, it is probable that the signal received is not a proper Manchester coded signal. If this occurs several bits in a row, a reasonable conclusion is that no proper signal is received, i.e. the signal received is noise. When this occurs, the MCU stops sending out clock and data signals, and waits for a synchronisation preamble to occur. Using software squelch prevents the application circuit from receiving noise.

MCU interface

To provide a practical example of an oversampling-based data-decision algorithm, we have written a program for the Microchip PIC16F87x series of microcontrollers. The code will also work with other PIC controllers, as long as they have the same instruction set and enough program memory. Both assembly code and C-code versions have been provided. Chipcon recommends using assembly code for production code, however.

For the purposes of the example, the PIC functions as an interface between the CC400/CC900 and an application circuit. When in transmit mode, the PIC transmits a clock to the application circuit, the application sends an NRZ-coded data stream back, the PIC then Manchester-encodes the data and sends it to the CC400/CC900. In receive mode, the CC400/CC900 sends Manchester-encoded data to the PIC, the PIC synchronises to the bit stream and decodes it into NRZ-form, which it sends to the application circuit together with a synchronised clock signal. The application circuit can switch the transceiver into receive, transmit or power-down modes by altering the PD and RX_TX signals. The PIC takes care of configuring the CC400/CC900 for the different modes.

After power-on, reset or a mode change, the MCU configures the CC400/CC900. It also initialises its own registers. It is not possible to change modes before the MCU is finished with configuring the current mode.

In PD mode, the MCU powers down the CC400/CC900 and then goes into sleep mode itself. The configuration takes 1768 instruction cycles. When in PD mode, the MCU can be awakened by changing the RX_TX or PD signals.

In TX mode, the MCU configures the CC400/CC900 for transmission. This takes 1781 instruction cycles. When the configuration is finished, data can be sent. A preamble should be sent before starting transmission of data, in order to allow the receiver to synchronise to the data rate. This is the responsibility of the application circuit.

Table 2 Mode pin coding

Pin		Mode
PD	**RX_TX**	
0	0	Transmit mode (TX)
0	1	Receive mode (RX)
1	X (Don't care)	Power-down mode (PD)

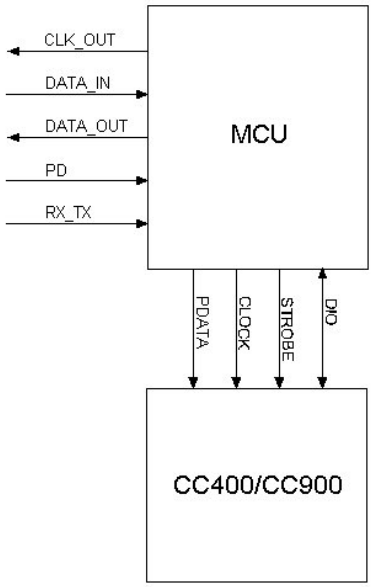


Figure 2 Block diagram of system

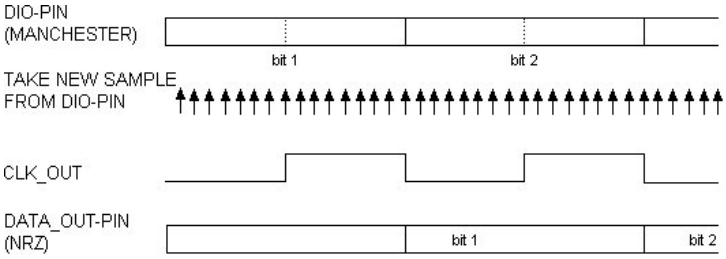


Figure 3 Timing in RX mode

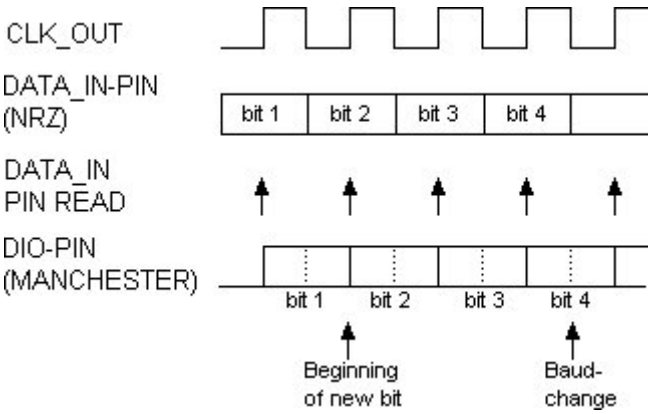


Figure 4 Timing in TX mode

Data decision algorithm

The software implements a data decision algorithm based on the principles discussed earlier in this document, using 8 samples per bit. This was found to be sufficient. In our tests, using 16 samples per bit did not improve performance enough to justify using twice as much processor time as 8 samples per bit. This code treats all samples equally, and does not implement the suggested sample prioritisation scheme; this is left as an exercise for the

reader. The data decision is based on a pure majority vote. If A is greater or equal to 4, the bit is a logical '1', else it is a '0'.

Synchronisation and resynchronisation is implemented. Before sending data, a preamble of '1's must be sent, the length of which depends on the value of the constants in the software and on the synchronisation required by the CC400/CC900. The receiver is able to synchronise on a data stream with a data rate within ±6.2% of its own.

The source code also implements data checking to determine whether it is receiving valid Manchester-coded data or not. If several consecutive bits fail this test, the software declares that it is out of synchronisation, and attempts to synchronise again. This is useful when receiving data from a transmitter that does not transmit continuously.

Using the software

This software was developed and tested using a Microchip PIC16F877. It can be ported to any controller using the same instruction set. To support all possible baud rates for the CC400/CC900, the controller must be run at 20 MHz or faster.

Both PORTB and PORTC are used, the pin allocations are shown below.

PIN NAME	DIRECTION	MCU-PIN	COMMENTS
CLK_OUT	OUT	RB0	In TX_mode: Data read from DATA_IN pin on the rising CLK_OUT edge. In RX_mode: New data valid on DATA_OUT on the falling CLK_OUT edge.
DATA_IN	IN	RB1	In TX_mode: Data read in NRZ format.
DATA_OUT	OUT	RB2	In RX_mode: Data from CCX00 in NRZ format.
PD	IN	RB4	Interrupt-on-change pin, used for mode decision: PD=1 -> Power down mode (PD). PD=0 -> Power on.
RX_TX	IN	RB5	Interrupt-on-change pin, used for mode decision: RX_TX=1 -> Receive mode (RX). RX_TX=0 -> Transmit mode (TX).
PDATA	OUT	RC0	Used for CCX00 configuration.
CLOCK	OUT	RC1	Used for CCX00 configuration.
STROBE	OUT	RC2	Used for CCX00 configuration.
DIO	IN/OUT	RC3	In RX-mode: MCU reads data from CCX00. In TX-mode: MCU writes data to CCX00. Signal is Manchester coded.
LOCK	OUT	RC4	Indicates whether MCU is synchronised or not: LOCK=1 -> MCU is synchronised LOCK=0 -> MCU is not synchronised

Depending on the application, some variables in the software may have to be changed. TIMING_RX, TIMING_TX, Rate_RX and Rate_TX must be changed for different data rates. The following table shows values for the most common combinations of oscillator frequencies and data rates. This table is **ONLY** valid for the assembly language version! Timing using C code is not predictable, and must be measured from simulations. Tables for the C code version are therefore not included.

Table 3 Assembly code timing

F_{osc}=20 MHz						
Data rate (kbps)	**TIMING_RX**	**Rate_RX**	**Error**	**TIMING_TX**	**Rate_TX**	**Error**
9.6	0xE5*	0xD0	0.160%	0x84	0xD0	0.224%
4.8	0xC5	0xD0	0.160%	0x02	0xD0	0.032%
2.4	0x84	0xD0	0.160%	0xFF	0xD1	0.064%
1.2	0x02	0xD0	0.160%	0x7F	0xD3	0.304%
0.6	0xFF	0xD1	0.160%	0x7E	0xD4	0.152%
0.3	0xFD	0xD2	0.032%	0x7E	0xD5	0.004%

F_{osc}=16 MHz						
Data rate (kbps)	**TIMING_RX**	**Rate_RX**	**Error**	**TIMING_TX**	**Rate_TX**	**Error**
9.6	-	-	-	-	-	-
4.8	0xD2	0xD0	0.160%	0x36	0xD0	0.080%
2.4	0x9E	0xD0	0.160%	0x33	0xD1	0.040%
1.2	0x36	0xD0	0.160%	0x31	0xD2	0.140%
0.6	0x33	0xD1	0.160%	0x31	0xD3	0.250%
0.3	0x31	0xD2	0.080%	0x30	0xD4	0.035%

F_{osc}=10 MHz						
Data rate (kbps)	**TIMING_RX**	**Rate_RX**	**Error**	**TIMING_TX**	**Rate_TX**	**Error**
9.6	-	-	-	-	-	-
4.8	0xE5	0xD0	0.160%	0x84	0xD0	0.224%
2.4	0xC5	0xD0	0.160%	0x02	0xD0	0.032%
1.2	0x84	0xD0	0.160%	0xFF	0xD1	0.064%
0.6	0x02	0xD0	0.160%	0x7F	0xD3	0.304%
0.3	0xFF	0xD1	0.160%	0x7E	0xD4	0.152%

F_{osc}=4 MHz						
Data rate (kbps)	**TIMING_RX**	**Rate_RX**	**Error**	**TIMING_TX**	**Rate_TX**	**Error**
9.6	-	-	-	-	-	-
4.8	-	-	-	-	-	-
2.4	-	-	-	-	-	-
1.2	0xD2	0xD0	0.160%	0x36	0xD0	0.080%
0.6	0x9E	0xD0	0.160%	0x33	0xD1	0.040%
0.3	0x36	0xD0	0.160%	0x31	0xD2	0.140%

F_{osc}=3.6864 MHz						
Data rate (kbps)	**TIMING_RX**	**Rate_RX**	**Error**	**TIMING_TX**	**Rate_TX**	**Error**
9.6	-	-	-	-	-	-
4.8	-	-	-	-	-	-
2.4	-	-	-	-	-	-
1.2	0xD6	0xD0	0%	0x46	0xD0	0.366%
0.6	0xA6	0xD0	0%	0x43	0xD1	0.130%
0.3	0x46	0xD0	0%	0x41	0xD2	0.326%

* Use one NOP instruction in *end_interrupt_RX*. Otherwise, remove it.

Calculating variables for desired oscillator frequency and data rate

The values for TIMING_RX, TIMING_TX, Rate_RX and Rate_TX are calculated as follows:

1. Determine the number of MCU instructions available per interrupt. A value must be calculated for both RX and TX interrupts. The data rate limitation for the system is set by the RX value, which cannot be less than 65 (The RX interrupt uses 65 instructions in the assembly language code). The number of instructions available are given by equations 1 and 2:

   RX : $\#\,Instructions / Interrupt = \frac{F_{OSC}}{32 \cdot R}$ [1]

   TX: $\#\,Instructions / Interrupt = \frac{F_{OSC}}{8 \cdot R}$ [2]

   F_{OSC} is the MCU clock frequency, and R is the data rate. The integer number closest to the value given by the equation will give the most exact interrupt timing.
2. Select the Rate_RX and Rate_TX values. These values determine the clock division ratio between the MCU clock frequency and the MCU timer. The possible values are 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128 and 1/256. The ratios Rate_RX and Rate_TX should be as small as possible in order to give as good timing accuracy as possible.

   $$TIMING\_RX = 256 - (\#\,Instructions / Interrupt - 12) \cdot Rate\_RX \quad [3]$$

   $$TIMING\_TX = 256 - (\#\,Instructions / Interrupt - 13) \cdot Rate\_TX \quad [4]$$

   The constants in the above equations are used to compensate for the time spent between the triggering of the timer interrupt and the resetting of the timer. If the code is modified so that resetting the timer occurs sooner or later, these constants must be modified. For instance, the C-code version uses 63 instructions for TX and 64 instructions for RX. Determining the values for these constants is done most easily by using a simulator and examining the instruction cycle counter.

3. In the end, we must check some conditions. TIMING_RX and TIMING_TX must be between 0 and 256. If the values do not satisfy these conditions, the next larger values of Rate_RX or Rate_TX must be chosen, and new TIMING_RX and TIMING_TX must be calculated.

Example:

MCU clock frequency F_{OSC}=16 MHz
Data rate: R=1200 bps

Equations 1 and 2 give the results:

RX : #Instructions / Interrupt = 417 (416.67)
TX : #Instructions / Interrupt = 1667 (1667.67)

Rate_RX should be as low as possible. We try ½, this results in TIMING_RX=53.5. This is not an integer number, we therefore round it to the closest possible integer, giving TIMING_RX=54. This number satisfies the conditions.

Rate_TX should also be as low as possible, but values of 1/2 and 1/4 result in negative values for TIMING_TX, and should not be used. Setting Rate_TX=1/8 results in TIMING_TX=49.25. We round this to the nearest integer, giving us a final value for TIMING_TX=49.

The final results are then:
TIMING_RX=54 (0x36)
TIMING_TX=49 (0x31)
Rate_RX = 1/2 (0xD0, see Microchip datasheet)
Rate_TX = 1/8 (0xD2, see Microchip datasheet)

Software implementation

The program is available in two versions; an assembly language version and a version written in ‘C’. The ‘C’ version is included as it is easier to read and understand, but Chipcon recommends using the assembly language version for performance reasons. All timing values given relate to the assembly language code. The timing of the ‘C’ version will vary according to optimisation settings and compiler settings. In our tests, using IAR’s C-compiler with all speed optimisations turned on, we managed to get the C routine to receive at 2400 bps using a 20 MHz oscillator clock for the PIC.

Both versions use the same algorithm, making it easy to refer to the ‘C’ version if the assembly version is difficult to understand.

Algorithm flowcharts

When the MCU is reset, the main program starts executing. The main part of the program first sets up I/O ports, then it runs ModeDecision. If the PD pin is high, the program configures the CC400/CC900 for power-down mode and the MCU goes into sleep mode. It will awaken when either the PD or the RX_TX pin changes state. If the PD pin is low, the program checks the RX_TX pin. If it is high, the program enters TX mode, configures the CC400/CC900 for transmission, and configures the DIO pin as outgoing. If the RX_TX pin is low, the program enters RX mode, configures the CC400/CC900 for receive, and configures the DIO pin as incoming.

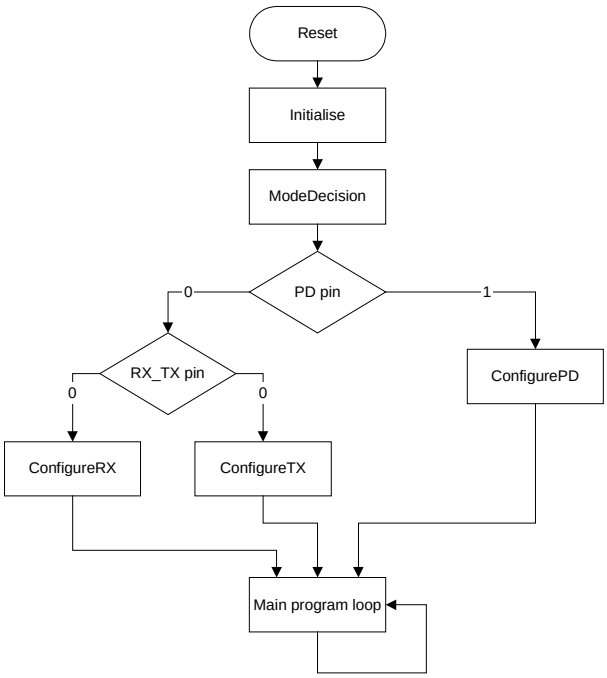


Figure 5 Main program algorithm

Configuring the CC400/CC900 is done using the CLOCK, PDATA and STROBE pins, a full configuration consists of 8 words of 16 bits each. The register values for each mode should be calculated using SmartRF Studio, and the user can then paste in the proper values into the constant declarations. In the 'C' version, the configuration data resides in three constant arrays, RX_CONFIG[], TX_CONFIG[] and PD_CONFIG[]. In the assembly language version, the value of each register resides in two constants for each mode. For example, the value of the A register in TX mode is located in A_TX_H_val and A_TX_L_val. The 8 most significant bits reside in A_TX_H_val, and the 8 least significant bits in A_TX_L_val. The data is sent with the most significant bit first.

When the configuration is done, the program enters the main program loop. This is typically where user code will be added. As written, the main loop merely writes the synchronisation status onto the SYNC pin.

In RX and TX mode, the main program loop will be interrupted when interrupts occur. If either the RX_TX or PD pin changes value, the ModeDecision routine will be executed, and the software changes mode. If a TMR0 interrupt occurs, the software will execute the RX or TX timer interrupt handler according to which mode is active.

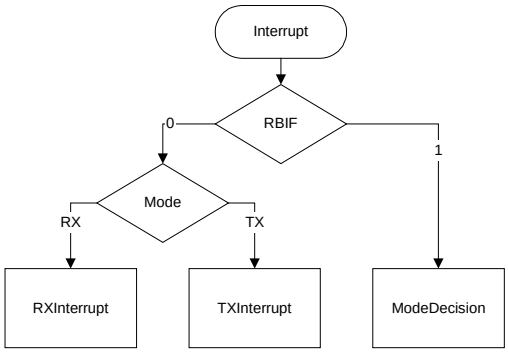


Figure 6 Interrupt handling

In TX mode 2 timer interrupts occur during one bit period. If Number is 0, the interrupt is the first one this bit period. The CLK_OUT pin is set high, the DATA_IN pin is read and stored in the Read variable. The DIO pin is set to the same value as DATA_IN, and Number is set to 1 This is the first baud of the Manchester coded signal. If Number is 1, the interrupt is the second this bit period. The CLK_OUT pin is cleared, the DIO pin is inverted and Number is set to 0. This represents the second baud of the Manchester coded signal. At the end of the TX interrupt handler, T0IF is cleared. This bit indicates that a TMR0 interrupt has occurred, and must be cleared before returning from the interrupt.

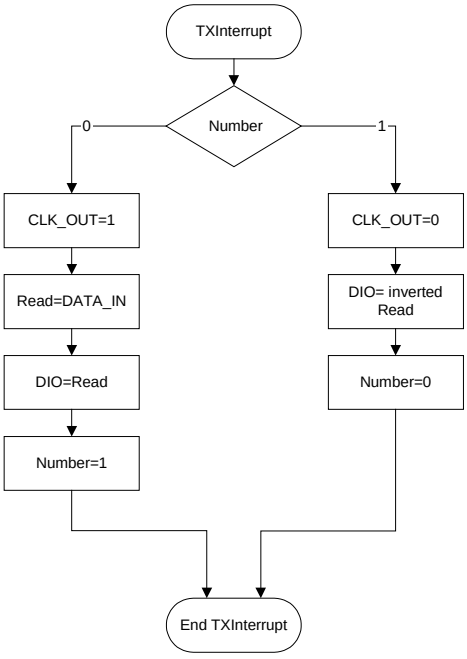


Figure 7 Timer interrupt in TX mode

In RX mode 8 timer interrupts occur during one bit period. Each time a timer interrupt occurs, a sample is taken of the DIO pin. The last two calculated values of A are stored in variables An1 and An2. These values are used for resynchronisation. A new value for A is then calculated. The Sample variable (called R in the assembly version) is used as a counter, and contains the number of the current sample. Resynchronisation is done by adjusting the value of this

variable, and can be done once per bit. The ResyncEnable variable (called R_ENABLE in the assembly version) is one when resynchronisation is to be performed. Resynchronisation is done at the start of each bit. Because of resynchronisation, Sample (R) can be 0 during two consecutive interrupts, and the ResyncEnable(R_ENABLE) variable is therefore needed to ensure that only one resynchronisation is done per bit.

When Sample (R) is equal to 3, we are in the middle of the bit, and the CLK_OUT pin is set high. The DATA_OUT pin is valid when the CLK_OUT pin is set low, which occurs at the start of each bit.

When Sample (R) is equal to 7, an entire bit has been sampled. The program then checks the Sync variable. This flag is set if the system has been synchronised. As long as Sync is 0, no data is sent on the DATA_OUT pin, and the CLK_OUT pin remains high. If Sync is 0, the system attempts to synchronise itself. The system must a certain number of successive signals over a certain level. The value A_LIMIT2 defines the required level A must exceed, and can easily be changed in the code. COUNT_LIMIT defines the number of successive ‘1’s the system requires before it is synchronised. As a worst case, the system can require 4+COUNT_LIMIT consecutive 1’s before it is synchronised.

When Sample (R) is equal to 7 and Sync is set to 1, the software checks signal quality. If A is to close to the centre of possible values, the signal is regarded as being a poor-quality Manchester coded signal (both chips are equal), and the BitErrors variable is increased. If the value of this variable equals BIT_ERROR_LIMIT, the system has received BIT_ERROR_LIMIT number of bad bits in a row, and the software sets Sync to 0, requiring the software to synchronise on a preamble of 1’s again. This part of the code can be removed if the transmitter will be transmitting continuously, but in a typical system it is useful to determine if the receiver is receiving valid data or not.

After the data quality check, the data decision itself is performed. If A is larger than A_LIMIT, the signal received is a ‘1’, and the DATA_OUT pin is set high. If the signal is equal to or lower than A_LIMIT, the signal received is a ‘0’, and the DATA_OUT pin is set low. The Sample (R) variable is set to 0, and the ResyncEnable (R_ENABLE) flag is set, indicating the a resynchronisation should be performed at the next interrupt.

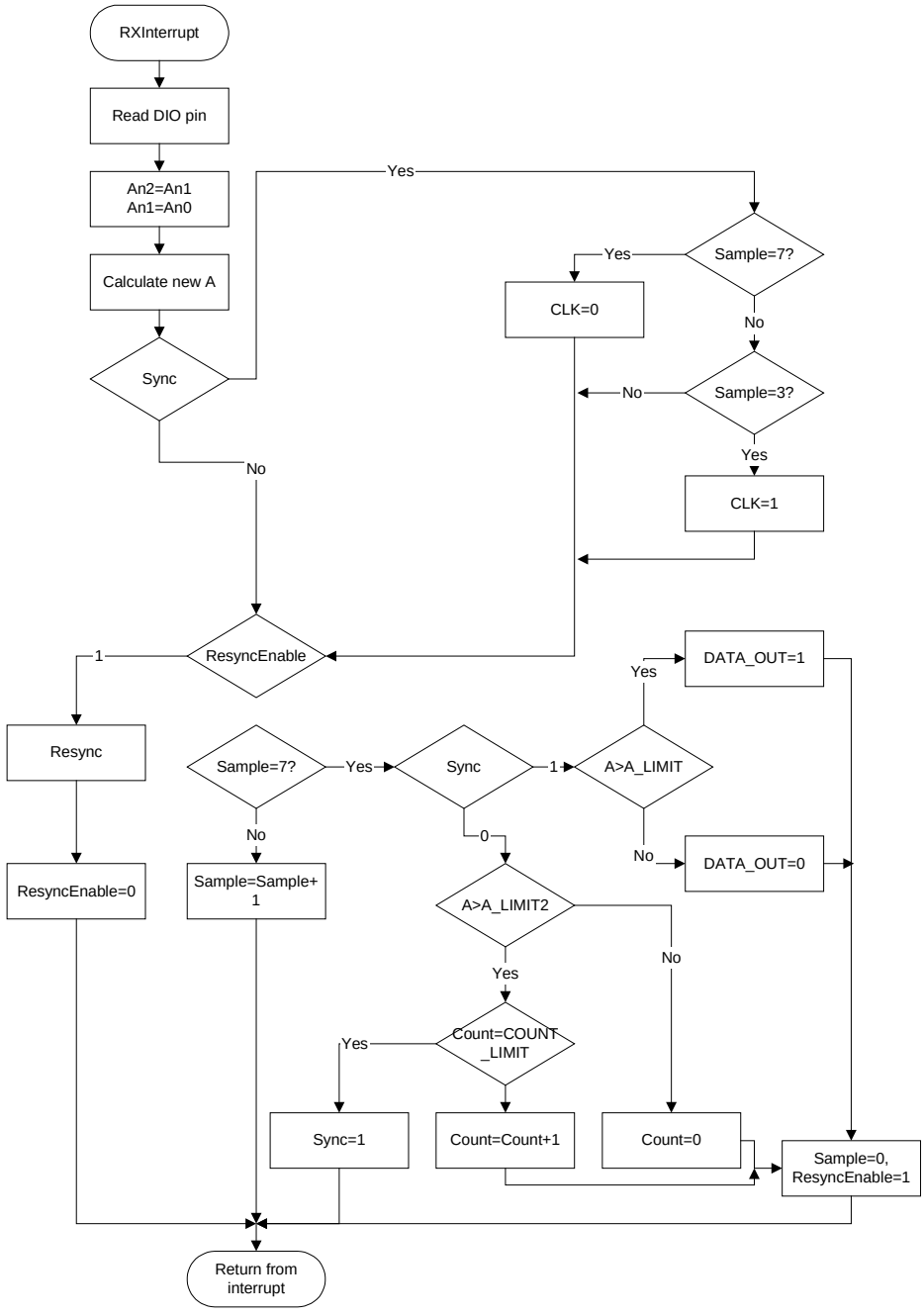


Figure 8 Timer interrupt in RX mode

When the ResyncEnable (R_ENABLE) flag is set, resynchronisation is performed. Normally the Sample (R) variable is increased by 1 every timer-interrupt, but resynchronisation can cause it to be increased by 2 or remain unchanged. This will make the next bit decision come earlier or later in time, respectively. At the time resynchronisation is performed, the value of A that was valid at the last bit decision is stored in variable Rn1. The values of A just before and just after the decision are stored in An2 and An0 (A), respectively. The resynchronisation is based on these values. If the timing is correct, An1 should be larger than the other two if a 1 was received, and lower if a 0 was received. If not, a resynchronisation may be necessary. If two or more of the values are the same, no resynchronisation is performed. The figure below

shows the different possibilities for the three values of A. The arrow indicates the time for the next bit decision, while the line in the middle (An1) is the time for the last decision. The algorithm differs slightly when the MCU is not synchronised. This is done to ensure that bit decisions will occur at maxima and not minima (synchronisation on 1's).

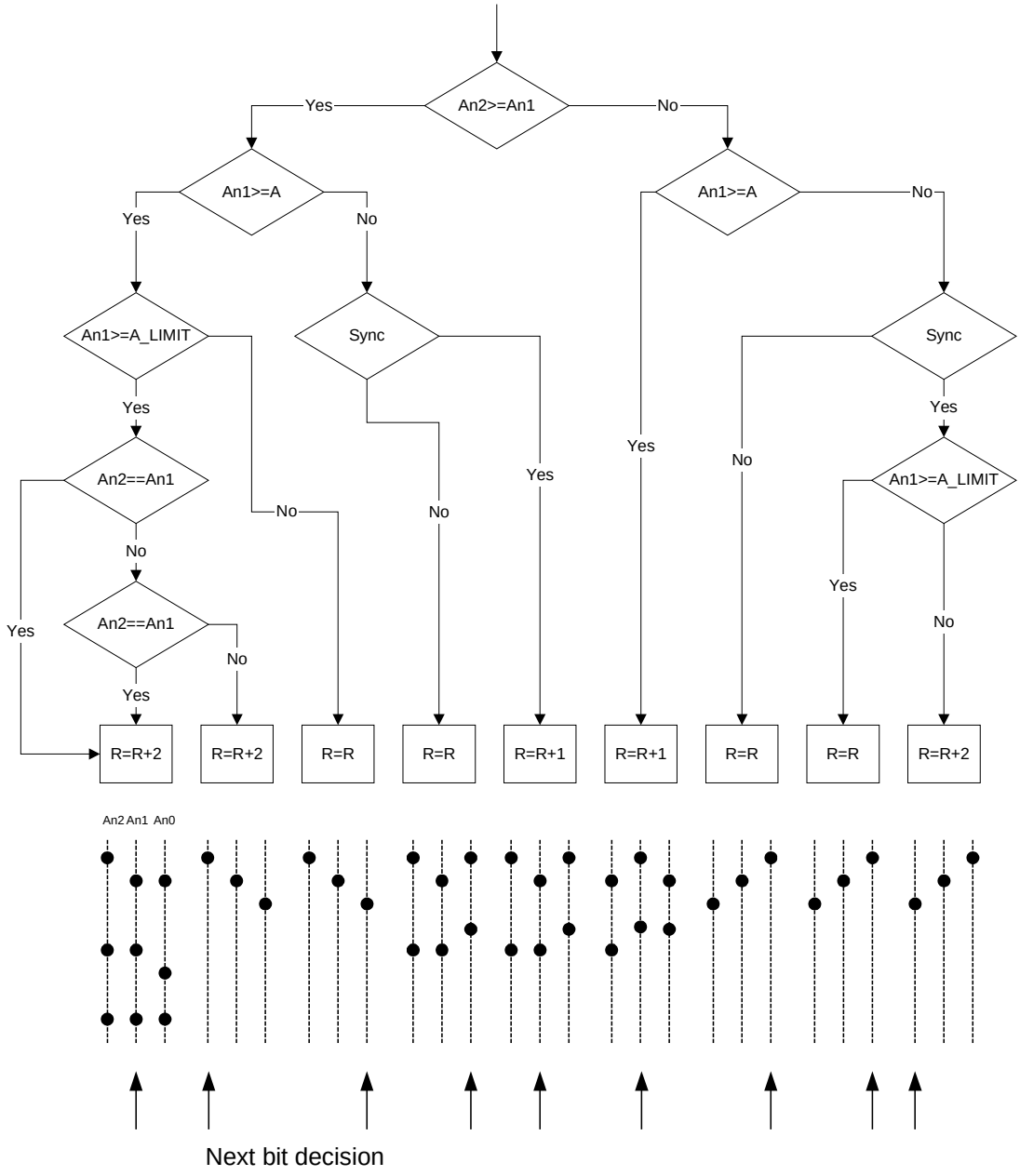


Figure 9 Resynchronization algorithm

Example source code – C version

```
/*****************************************************************************/
/*  Application note 008                                                     */
/*  Oversampling and data decision for the CC400/CC900                       */
/*                                                                           */
/*  File:          AN_008.c                                                  */
/*                                                                           */
/*  Microcontroller:                                                        */
/*                 Microchip PIC16F87x or compatible                         */
/*                 By changing hardware- and compiler-dependant code,        */
/*                 this code can be used with any C compiler.                */
/*                 Code was developed and tested using IAR's C Compiler for  */
/*                 mid-range PIC MCUs and a PIC16F877 MCU.                   */
/*                                                                           */
/*  Author:       Code written by Karl H. Torvmark based on an assembly      */
/*                language program by Kjell Tore Heien.                      */
/*                                                                           */
/*  Contact:      Chipcon AS +47 22 95 85 44                                 */
/*                wireless@chipcon.com                                       */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*      DESCRIPTION                                                          */
/*                                                                           */
/*  General                                                                  */
/*                                                                           */
/* This code implements oversampling of the output signal from a CC400/      */
/* CC900 RF-transceiver. The output signal from the transceiver is coded in  */
/* Manchester format. The MCU converts this signal to an NRZ signal by       */
/* taking 8 samples per bit. Signal decision is by majority vote based on    */
/* these samples, filtering out noise.                                       */
/*                                                                           */
/* The MCU is also used during transmission. The MCU then reads NRZ-coded    */
/* data, and passes it on to the RF transceiver in Manchester format.        */
/*                                                                           */
/* A power down mode is also implemented. The MCU puts the CC400/CC900 into  */
/* power down mode, then puts itself to sleep. It will awaken on a mode      */
/* change. Mode changes are controlled by setting the RX_TX and PD pins.     */
/* The MCU configures the CC400/CC900 for the appropriate mode by sending    */
/* it configuration data on the PDATA/CLOCK/STROBE pins.                     */
/*                                                                           */
/*****************************************************************************/
/* This code must be adapted to fit the user application. Transmission       */
/* rates and RF configuration data can be changed by changing the            */
/* appropriate constants. The values for the configuration data should be    */
/* calculated using SmartRF studio, and pasted into the source code.         */
/*                                                                           */
/* As written, the MCU merely functions as a protocol translator between     */
/* an application circuit communicating using NRZ data, and the RF chip      */
/* using Manchester coding. In a typical application, the MCU will have      */
/* other functions, this must be added to the software. This software as     */
/* written does not insert the required preamples in front of data           */
/* messages, this is left to the application circuit or to additional        */
/* software written by the user.                                             */
/*                                                                           */
/*****************************************************************************/
/* Implementation                                                            */
/*                                                                           */
/* An assembly language version of this program also exists. Chipcon         */
/* recommends using the assembly version rather than this C program, as the  */
/* assembly language program is faster and provides better timing.           */
/* This 'C' version of the software was written to make it easier for the    */
/* customer to understand the logic behind the algorithm.                    */
/* However, for low baud-rates (1200 bps or lower), the 'C' version is fast  */
/* enough to be useable.                                                     */
/*                                                                           */
/* This code was written and tested using the IAR C compiler for PIC16x MCUs*/
/* It was tested using a PIC16F877. The program should be easy to port       */
/* to other PIC MCUs by changing the I/O port definitions.                   */
/* For maximum performance, it should be compiled using maximum speed        */
/* optimisation in the compiler.                                             */
/*                                                                           */
/* Mode configuration:                                                       */
```

```
/*                                                                          */
/*     ******************************************************           */
/*     * PIN                 *           MODE              *          */
/*     ************************                            *          */
/*     * PD *  RX_TX         *                             *          */
/*     ******************************************************         */
/*     * 0  *  0             *      Transmit mode (TX)     *          */
/*     * 0  *  1             *      Receive mode (RX)      *          */
/*     * 1  *  X (Don't care)*      Power-down mode (PD)   *          */
/*     ******************************************************           */
/*                                                                        */
/*  Timing in TX mode:                                                    */
/*                                _  _  _  _  _  _  _  _  _  _          */
/*     CLK_OUT                 _/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/          */
/*                                                                        */
/*     DATA_IN pin (NRZ)        | b1| b2| b3| b4| b5| b6| b7| b8| b9|      */
/*     DATA_IN pin read        R   R   R   R   R   R   R   R   R   R       */
/*                              _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _       */
/*     DIO pin (Manchester)    |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|      */
/*                                                                        */
/*  Timing in RX mode:                                                    */
/*                               _______ _______ _______ _______ _____    */
/*     DIO pin (Manchester)    |_______|_______|_______|_______|_____     */
/*     Sample                   S S S S S S S S S S S S S S S S S S       */
/*                                      _______         _______           */
/*     CLOCK_OUT pin           ________/       \_______/       \_____     */
/*                              _________________ _________________ _____  */
/*     DATA_OUT pin            |_________________|_________________|_____  */
/*                                                                        */
/*                             |     Bit 1     |     Bit 2     |  Bit 3 ...*/
/**************************************************************************/



#include "io16f877.h"               /* Includes I/O definitions for 16F877 */
#include "inpic.h"                  /* Includes PIC intrinsic functions    */

/* Definitions for boolean expressions */
#define TRUE (1==1)
#define FALSE !TRUE

/**************************************************************************/
/* Overview of port usage:                                                */
/*                                                                        */
/*  PORT:           PORTB         User interface                          */
/*  pin 0           CLK_OUT       Clock out                               */
/*  pin 1           DATA_IN       Data in (TX mode)                       */
/*  pin 2           DATA_OUT      Data out (RX mode)                      */
/*  pin 3           unused                                                */
/*  pin 4           PD            System configuration (1=Power down)     */
/*  pin 5           RX_TX         System configuration (1=RX, 0=TX)       */
/*  pin 6           unused                                                */
/*  pin 7           unused                                                */
/*                                                                        */
/*  PORT:           PORTC         Communication with CC700/CC900          */
/*  pin 0           PDATA         Configuration pin                       */
/*  pin 1           CLOCK         Configuration pin                       */
/*  pin 2           STROBE        Configuration pin                       */
/*  pin 3           DIO           Bi-directional data pin                 */
/*  pin 4           SYNC          Synchronisation status                  */
/*  pin 5           unused                                                */
/*  pin 6           unused                                                */
/*  pin 7           unused                                                */
/**************************************************************************/

/* Pin usage definitions */
#define PDATA RC0
#define CLOCK RC1
#define STROBE RC2
#define DIO RC3
#define SYNC RC4

#define CLK_OUT RB0
#define DATA_IN RB1
#define DATA_OUT RB2
#define PD RB4
```

```
#define RX_TX RB5


/* Data quality constants */

/*****************************************************************************/
/* These values can be changed to optimise the program for different data    */
/* and noise characteristics.                                                */
/* A_LIMIT_VAL           - Decision limit. If A is smaller than this value,  */
/*                         the signal is a logical '0', else signal is '1'   */
/* A_LIMIT_VAL2          - Decision limit during synchronisation. If A is    */
/*                         smaller than this value signal is regarded as a   */
/*                         '0', else it is a '1'. This value should be       */
/*                         larger than A_LIMIT_VAL, and must be larger than  */
/*                         4.                                                */
/* COUNT_LIMIT_VAL       - Sets the synchronisation length. The value        */
/*                         determines how many 1's (A>=A_LIMIT_VAL2)         */
/*                         the MCU must receive before it is synchronised.   */
/*                         Should be >=4. The preamble sent by the           */
/*                         transmitter should be longer than this, as the    */
/*                         RF transceiver also needs time to synchronise.    */
/* A_SYNC_LIMIT_HI,                                                          */
/* A_SYNC_LIMIT_LO       - Sets the limits for a proper Manchester-coded     */
/*                         bit. If A is between these thresholds for a       */
/*                         number of bits (set by BIT_ERROR_LIMIT,           */
/*                         described below), the MCU goes out of             */
/*                         synchronisation. This can be used to detect when  */
/*                         valid data is no longer received. The MCU then    */
/*                         waits for a new preamble.                         */
/* BIT_ERROR_LIMIT       - Determines how many invalid bits must be          */
/*                         received before the MCU loses synchronisation.    */
/*****************************************************************************/


#define A_LIMIT_VAL 0x04
#define A_LIMIT_VAL2 0x05
#define COUNT_LIMIT_VAL 0x0A
#define A_SYNC_LIMIT_HI 0x05
#define A_SYNC_LIMIT_LO 0x03
#define BIT_ERROR_LIMIT 0x02


/*****************************************************************************/
/* These values must be changed for different combinations of bit-rate and   */
/* MCU clock frequency. They determine the frequency of the intermal TMR0    */
/* interrupt, used both in RX and TX mode. See documentation for details.    */
/*****************************************************************************/

/*****************************************************************************/
/* Fosc = 20 MHz                                                             */
/*****************************************************************************/
/* Data rate * TIMING_RX  * Rate_RX * Error  * TIMING_TX * Rate_TX * Error   */
/*****************************************************************************/
/* 2.4 kbps  * 0x9E       * 0xD0    * 0.160% * 0x0B      * 0xD1    * 0.160% */
/* 1.2 kbps  * 0x1C       * 0xD0    * 0.160% * 0x03      * 0xD2    * 0.032% */
/* 600 bps   * 0x0C       * 0xD1    * 0.160% * 0x80      * 0xD4    * 0.257% */
/* 300 bps   * 0x04       * 0xD2    * 0.160% * 0x7F      * 0xD5    * 0.208% */
/*****************************************************************************/
/*                                                                           */
/* For other combinations of MCU clock frequencies and data-rates, see       */
/* application note for values and formulas.                                 */
/*                                                                           */
/*****************************************************************************/
/* Default values                                                            */
/*      20 MHz MCU clock, 1.2 kbps data rate                                 */
/*****************************************************************************/

#define RATE_RX 0xD0
#define RATE_TX 0xD4

#define TIMING_RX 0x1C
#define TIMING_TX 0xC1

/* The location of the interrupt handler vector. Must be changed for */
/* different PIC MCUs */
#define INTERRUPT_VECTOR 0x04
```

```
/* The CC400/CC900 configuration parameters can be modified here */
/* Calculate new values using SmartRF Studio */

/*****************************************************************************/
/* Default values:                                                           */
/*                                                                           */
/*      Chip used      :      CC400                                          */
/*                                                                           */
/*      X-tal frequency:      12.000000 MHz                                  */
/*      X-tal accuracy :      50 ppm                                         */
/*      RF frequency   :      433.920000                                     */
/*      IF Stage       :      200 kHz                                        */
/*      Frequency sep. :      10 kHz                                         */
/*      Data rate      :      1.2 kbit/s                                     */
/*      Power amp. class:     Class B                                        */
/*      RF output power       :       10 dBm                                */
/*      Receiver mode  :      Optimum sensitivity                            */
/*      LOCK indicator :      Continous                                      */
/*      VCO current    :      000 (Maximum)                                  */
/*****************************************************************************/

/*          A,      B,      C,      D,      E,      F,      G,      H      */
const short RX_CONFIG[8] =
          {0x002A, 0x230B, 0x4141, 0x6771, 0x8A00, 0xB803, 0xD24C, 0xE450 };
const short TX_CONFIG[8] =
          {0x082A, 0x230B, 0x4141, 0x67A4, 0x8A14, 0xB803, 0xD24C, 0xE860 };
const short PD_CONFIG[8] =
          {0x182A, 0x230B, 0x4151, 0x6771, 0x8A00, 0xB803, 0xD24C, 0xE860 };

/* global variables shared by main program and interrupt routine */

char ShiftReg;     /* All samples are shifted into this register */
char An0;          /* The current value of A */
char An1;          /* The value of A one sample ago */
char An2;          /* The value of A two samples ago */
char Sample;       /* Number of sample within bit, between 0 and 7 */
char Sync;         /* Set to 1 if SW is synchronised, 0 otherwise */
char Count;        /* Counts consecutive 1's received during synchronisation*/
char ResyncEnable; /* Set to 1 if resynchronisation should be performed */
                   /* the next interrupt */
char ALimit;       /* Decision limit. If A>ALimit, the bit received is a 1 */
char ALimit2;      /* Decision limit during synchronisation */
char CountLimit;   /* Number of consequtive 1's requried to synchronise */
char Number;       /* Is 0 during the first baud in TX mode, 1 during */
                   /* the second */
char Read;         /* Used to store value read from DATA_IN in TX mode */
char BitErrors;    /* Counts number of consecutive bad bits received */
enum {TXMODE=0, RXMODE=1, PDMODE=2} Mode;
                   /* Keeps track of which mode software is in */


/*****************************************************************************/
/* This routine initialises the MCU. Must be modified for different          */
/* applications.                                                             */
/*****************************************************************************/

void Initialise(void)
{
  /* Clear I/O port latches */
  PORTC=0x00;
  PORTB=0x00;

  /* PORTC : Set Pin 0, 1, 2 and 4 as output, Pin 3 as input */
  TRISC=0x08;

  /* PORT B : Set Pin 0 and 2 as output, pin 1, 4 and 5 as input */
  TRISB=0x32;
}

/*****************************************************************************/
/* This routine sends new configuration data to the CC400/CC900.             */
/*****************************************************************************/

void ConfigureCCX00(short const Configuration[8])
{
```

```
  char BitCounter;
  char WordCounter;
  union {         /* This union is used to easily access the most */
                  /* significant bit of the configuration data */
    unsigned short Data;
    struct
    {
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short :1;
      unsigned short MSB :1;
    };
  };

  STROBE=0;

  for (WordCounter=0;WordCounter<8;WordCounter++)
  {
    Data=Configuration[WordCounter];
    for (BitCounter=0;BitCounter<16;BitCounter++)
    {
      CLOCK=1;
      STROBE=0;
      PDATA=MSB;
      Data=Data<<1;
      CLOCK=0;
      if (BitCounter==15)
        STROBE=1;
      else
        STROBE=0;
    }
  }
  CLOCK=1;
  STROBE=0;
  CLOCK=0;
}

/*****************************************************************************/
/* This routine configures the software for RX mode. Also configures the    */
/* CC400/CC900 for RX.                                                       */
/*****************************************************************************/

void ConfigureRX(void)
{
  TRISC=0x08;          /* Set DIO as input */

  ConfigureCCX00(RX_CONFIG);

  /* Initialise variables for RX mode */

  ShiftReg=0;
  An0=0x08;
  An1=0x08;
  An2=0x08;
  Sample=0x00;
  Count=0x00;
  ResyncEnable=FALSE;
  Sync=FALSE;
  ALimit=A_LIMIT_VAL;
  ALimit2=A_LIMIT_VAL2;
  CountLimit=COUNT_LIMIT_VAL;
  Mode=RXMODE;

  /* Enable TMR0 interrupt */
```

```
  TMR0=0xFD;        /* Set a short time to next interrupt */
  OPTION=RATE_RX;
  INTCON=0xA8;      /* Enable interrupts */

}

/*****************************************************************************/
/* This routine configures the software for TX mode. Also configures the    */
/* CC400/CC900 for TX.                                                       */
/*****************************************************************************/
void ConfigureTX(void)
{
  TRISC=0x00;          /* Set DIO as output */

  ConfigureCCX00(TX_CONFIG);

  /* Initialise variables for TX mode */

  Mode=TXMODE;
  Number=0;

  PORTB=0x00;         /* Clear PORTB */

  /* Enable TMR0 interrupt */

  TMR0=0xFC;           /* Set a short time to next interrupt */
  OPTION=RATE_TX;
  INTCON=0xA8;        /* Enables interrupts */
}

/*****************************************************************************/
/* This routine configures the software for PD mode. Also configures the    */
/* CC400/CC900 for power-down. Puts the MCU into sleep mode.                 */
/*****************************************************************************/
void ConfigurePD(void)
{
  ConfigureCCX00(PD_CONFIG);

  CLK_OUT=0;
  DATA_OUT=0;
  Mode=PDMODE;

  INTCON=0x88;    /* Enable interrupts */
  __sleep();      /* Put MCU to sleep */
}

/*****************************************************************************/
/* Determines which mode the MCU should be in by reading the mode pins.     */
/*****************************************************************************/
void ModeDecision()
{
  if (PD)             /* PD has higher priority than RX_TX */
    ConfigurePD();
  else if (RX_TX)
    ConfigureRX();
  else
    ConfigureTX();
}

/*****************************************************************************/
/* Logic to resynchonise the MCU to the data stream. See documentation for  */
/* details regarding the resynchronisation algorithm. Called by the RX      */
/* interrupt routine.                                                        */
/*****************************************************************************/
void Resync(void)
{
  if (An2>=An1) {
      if (An1>=An0) {
        if (An1>=ALimit) {
          if (An2==An1)
            Sample++;
          else if (An1==An0)
            Sample++;
          else {
            Sample+=2;
            ResyncEnable=FALSE;
          }
```

```
        }
        else {
          ResyncEnable=FALSE;
        }
      }
      else if (Sync) {
        Sample++;
        ResyncEnable=FALSE;
      }
      else
        ResyncEnable=FALSE;
    }
  else if (An1>=An0)
    Sample++;
  else {
    if (Sync) {
      if (An1>=ALimit)
        ResyncEnable=FALSE;
      else {
        Sample+=2;
        ResyncEnable=FALSE;
      }
    }
    else
      ResyncEnable=FALSE;
  }
}

/*****************************************************************************/
/* This routine is called when TMR0 interrupts occur in RX mode. It          */
/* oversamples the incoming signal and performs synchronisation and          */
/* resynchronisation.                                                        */
/*****************************************************************************/
void RXInterrupt(void)
{
  char OldValue;
  char NewValue;


  /* Reinitialise timer, this should be done as quickly as possible */
  TMR0=TIMING_RX;

  /* Store old values of A */
  An2=An1;
  An1=An0;


  OldValue=ShiftReg&0x01;
  NewValue=DIO;

  /* Shift new sample into shift register */
  ShiftReg=ShiftReg>>1 | ((NewValue==0)?0x00:0x80);
  /* Invert new middle bit (Manchester coding) */
  ShiftReg^=0x08;

  /* Update A according to values shifted in and out */
  /* (We are really counting the number of 0's in the shift register) */
  if ((OldValue==1)&&(NewValue==0))
    An0++;
  if ((OldValue==0)&&(NewValue==1))
    An0--;

  if ((ShiftReg&0x08)==0)
    An0++;
  else
    An0--;

  /* Update clock at the appropriate time */
  if (Sync) {
    if (Sample==3)
      CLK_OUT=1;
    if (Sample==7)
      CLK_OUT=0;
  }

  if (ResyncEnable) {
    /* Resync */
```

```
    Resync();
  }
  else if (Sample==7)
    /* Finished with entire bit */

    /* Are we in sync? */
    if (Sync) {

      /* Check if this is an invalid Manchester bit. If it is, */
      /* increase the BitErrors count. If not, clear the count */
      if ((An0<=A_SYNC_LIMIT_HI)&&(An0>A_SYNC_LIMIT_LO)) {
        BitErrors++;
        if (BitErrors==BIT_ERROR_LIMIT) {
          Sync=FALSE;
          BitErrors=0;
        }
      }
      else
        BitErrors=0;

      /* Output data */
      DATA_OUT=(An0>ALimit);
      CLK_OUT=0;
      Sample=0;
      ResyncEnable=TRUE;
    }
    else {
      /* We are not in sync, synchronise */
      Sample=0;
      if (An0>=ALimit2) {
        if (Count==CountLimit)
          Sync=TRUE;
        Count++;

        ResyncEnable=TRUE;
      }
      else {
        Count=0;
        ResyncEnable=TRUE;
      }

  }
  else
    /* Ready for next sample */
    Sample++;

  /* Must clear interrupt flag before returning from interrupt */
  T0IF=0;
}

/*************************************************************************/
/* This routine is called when TMR0 interrupts occur in TX mode. It outputs */
/* clock data and converts incoming NRZ data to Manchester format.          */
/*************************************************************************/
void TXInterrupt(void)
{
  /* Reinitialise timer, this should be done as quickly as possible */
  TMR0=TIMING_TX;
  if (Number==0) {
    /* First baud in bit */
    CLK_OUT=1;
    Read=DATA_IN;
    DIO=Read;
    Number=1;
  }
  else {
    /* Second baud in bit */
    CLK_OUT=0;
    DIO=~Read;
    Number=0;
  }
  /* Must clear interrupt flag before returning from interrupt */
  T0IF=0;
}

/*************************************************************************/
```

```
/* This code is called every time an interrupt occurs. If a mode pin has    */
/* changed status, the software is initialised to the new mode. Otherwise,  */
/* a TMR0 interrupt has occured, and the handling routine appropriate to    */
/* the current mode is called.                                              */
/****************************************************************************/

#pragma vector=INTERRUPT_VECTOR
__interrupt void InterruptHandler(void)
{
  if (RBIF){
    /* Mode bits have changed */
    RBIF=0;
    ModeDecision();
   }
  else {
    /* Must have been the timer interrupt */
    switch(Mode) {
      case RXMODE : RXInterrupt();
           break;
      case TXMODE : TXInterrupt();
           break;
    }
  }
}

/****************************************************************************/
/* Main program. The MCU is initialised and put into the correct mode. The  */
/* software then goes into an infinite loop. User code will typically be    */
/* inserted here. By default, the only thing the software does is indicate  */
/* synchronised/not synchronised status on the RC4 pin                      */
/****************************************************************************/
void main(void)
{
  Initialise();
  ModeDecision();

  while (TRUE)        /* Loop forever */
  {
  /* Main loop of program here */
  if(Sync)
    SYNC=1;
  else
    SYNC=0;
  }
}
```

Example source code – assembly version

```
;*******************************************************************************
;       APPLICATION NOTE 008
;       Oversampling and data decision for the CC400/CC900
;
;       File :          AN_008.asm
;
;       Microcontroller:
;                       Microchip PIC16F87x or compatible
;                       Can be adapted to all mid-range
;                       PIC controllers with enough program
;                       memory by changing I/O definitions
;
;       Author :        Code originally written by Kjell Tore Heien
;                       Modified by Karl H. Torvmark
;
;       Contact :       Chipcon AS +47 22 95 85 44
;                       wireless@chipcon.com
;
;       Version :       1.2 - 2001-08-14
;
;*******************************************************************************

;*******************************************************************************
;       DESCRIPTION
;  General:
;       This code oversamples the output signal from a
;       CC400/CC900 RF-transceiver. The output signal is coded
;       in Manchester format, the MCU converts to NRZ format by
;       taking 8 samples/bit. Signal decision is based on the
;       majority of these samples, so noise will be filtered out.
;
;       The MCU can also be used to transmision. The MCU reads NRZ
;       signals, sending it to the transceiver in Manchester format.
;
;       The configuration of CCX00 is done by the MCU, with three
;       possible modes, Transceive(TX), Receive (RX) and Power Down (PD)
;       In PowerDown mode, both the transceiver and MCU are powered
;       down (sleep)
;
;       The code needs to be changed to fit the application. The values
;       for the configuration registers in CCX00 must be calculated in
;       SmartRF Studio for the actual parameters and copied into the code.
;       (A_RX_H_val to H_PD_L_val)
;
;       Also some timing variables must be changed to match the data
;       rate (TIMING_RX, TIMING_TX, Rate_RX and Rate_TX).
;       See application note for formulas and values.
;
;  Resources:
;       Program memory :        428 words
;       I/O pins       :        Port B pins 0,1,2,4 and 5
;                               Port C pins 0,1,2,3 and 4
;       Peripherals    :        TMR0
;
;
;  Mode configuration:
;
;       ********************************************************
;       * PIN                   *           MODE               *
;       ************************                                 *
;       * PD *  RX_TX           *                              *
;       ********************************************************
;       * 0  *  0               *       Transmit mode (TX)     *
;       * 0  *  1               *       Receive mode (RX)      *
;       * 1  *  X (Don't care) *       Power-down mode (PD)   *
;       ********************************************************
;
;  Timing in TX mode:
;                                  _   _   _   _   _   _   _   _   _
;       CLK_OUT                 _/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/
;
;       DATA_IN pin (NRZ)        | b1| b2| b3| b4| b5| b6| b7| b8| b9|
;       DATA_IN pin read        R   R   R   R   R   R   R   R   R   R
```

```
;
;       DIO pin (Manchester)   |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
;
;  Timing in RX mode:
;                               _______ _______ _______ _______ _____
;       DIO pin (Manchester)   |_______|_______|_______|_______|_____
;       Sample                  S S S S S S S S S S S S S S S S S S S
;                                       _______         _______
;       CLOCK_OUT pin          ________/       \_______/       \_____
;                               _______________ _______________ _____
;       DATA_OUT pin           |_______________|_______________|_____
;
;                              |     Bit 1     |     Bit 2     |  Bit 3 ...
;
;
;*****************************************************************************


;*****************************************************************************
; This part gives names to predefined MCU registers
; Overview of ports:
;
;       PORT:           PORTB           - User interface
;       pin 0           CLK_OUT         - Clock out
;       pin 1           DATA_IN         - Data in (TX mode)
;       pin 2           DATA_OUT        - Data out (RX mode)
;       pin 3           unused          -
;       pin 4           PD              - System configuration (PD(1))
;       pin 5           RX_TX           - System configuration (RX(1)/TX(0))
;       pin 6           unused          -
;       pin 7           unused          -
;
;       PORT:           PORTC           - Communication with CC700/CC900
;       pin 0           PDATA           - Configuration pin
;       pin 1           CLOCK           - Configuration pin
;       pin 2           STROBE          - Configuration pin
;       pin 3           DIO             - Bi-directional data pin
;       pin 4           SYNC            - Synchronisation status
;       pin 5           unused          -
;       pin 6           unused          -
;       pin 7           unused          -
;*****************************************************************************

#define OPTION_REG      0x01
#define TMR0            0x01
#define PCL                   0x02
#define STATUS          0x03
#define CARRY           0
#define Z               2
#define RP0             5
#define RP1             6
#define PORTB           0x06
#define CLK_OUT         0
#define DATA_IN         1
#define DATA_OUT        2
#define PD              4
#define RX_TX           5
#define PORTC           0x07
#define PDATA           0
#define CLOCK           1
#define STROBE          2
#define DIO             3
#define TRISB           0x06
#define TRISC           0x07
#define INTCON          0x0B
#define RBIF            0
#define INTF            1
#define T0IF            2
#define RBIE            3
#define T0IE            5
#define GIE             7


;*****************************************************************************
; This part gives names to variables and status registers
; mapped in the general purpose registers
;*****************************************************************************
```

```
#define X               0x20
#define A               0x22
#define An1             0x23
#define An2             0x24
#define A_LIMIT         0x25
#define A_LIMIT2       0x2F
#define R               0x26
#define COUNT           0x27
#define COUNT_LIMIT     0x28
#define CONTROL         0x29
#define R_ENABLE        0
#define SYNC            1
#define CONF_LOOP       2
#define TEMP            0x2A
#define TEMP1           0x2B
#define TEMP2           0x2C
#define INPUT           0x2D
#define DI              0
#define MODE            0x2E
#define RXTX            0
#define NUMBER          1
#define READ            2
#define CONFIG_REG     0x40
#define TABLE_PT       0x41
#define REG_COUNTER    0x42
#define BIT_COUNTER    0x43
#define BIT_ERRORS     0x44

;*************************************************************************
; Status flags in registers are defined as follows :
;
; CONTROL (address 0x29)
; Bit number   Flag
;       0      R_ENABLE
;       1      SYNC
;       2      CONF_LOOP
;       3      unused
;       4      unused
;       5      unused
;       6      unused
;       7      unused
;
; INPUT (address 0x2D)
; Bit number   Flag
;       0      DI
;       1      unused
;       2      unused
;       3      unused
;       4      unused
;       5      unused
;       6      unused
;       7      unused
;
; MODE (address 0x2E)
; Bit number   Flag
;       0      RXTX
;       1      NUMBER
;       2      READ
;       3      unused
;       4      unused
;       5      unused
;       6      unused
;       7      unused
;
;*************************************************************************


;*************************************************************************
; Configurations value for CC700/CC900, given by SmartRF Studio
; These values must be changed to match the application.
; The 3 most significant bits of A_RX_H_val contain the address of register A
; in the CC700/CC900. The rest of A_RX_H_val contains the 5 bits in
; register A for RX mode. A_RX_L_val contains the 8 least significant bits of
; register A (Similar for register B-H).
;*************************************************************************
```

```
;*******************************************************************************
; Default values:
;
;       Chip used       :       CC400
;
;       X-tal frequency         :       12.000000 MHz
;       X-tal accuracy :        50 ppm
;       RF frequency    :       433.920000
;       IF Stage        :       200 kHz
;       Frequency sep. :        10 kHz
;       Data rate       :       1.2 kbit/s
;       Power amp. class:       Class B
;       RF output power         :       10 dBm
;       Receiver mode   :       Optimum sensitivity
;       LOCK indicator :        Continous
;       VCO current     :       000 (Maximum)
;*******************************************************************************


A_RX_H_val      EQU     0x00
A_RX_L_val      EQU     0x2A
B_RX_H_val      EQU     0x23
B_RX_L_val      EQU     0x0B
C_RX_H_val      EQU     0x41
C_RX_L_val      EQU     0x41
D_RX_H_val      EQU     0x67
D_RX_L_val      EQU     0x71
E_RX_H_val      EQU     0x8A
E_RX_L_val      EQU     0x00
F_RX_H_val      EQU     0xB8
F_RX_L_val      EQU     0x03
G_RX_H_val      EQU     0xD2
G_RX_L_val      EQU     0x4C
H_RX_H_val      EQU     0xE4
H_RX_L_val      EQU     0x50

A_TX_H_val      EQU     0x08
A_TX_L_val      EQU     0x2A
B_TX_H_val      EQU     0x23
B_TX_L_val      EQU     0x0B
C_TX_H_val      EQU     0x41
C_TX_L_val      EQU     0x41
D_TX_H_val      EQU     0x67
D_TX_L_val      EQU     0xA4
E_TX_H_val      EQU     0x8A
E_TX_L_val      EQU     0x14
F_TX_H_val      EQU     0xB8
F_TX_L_val      EQU     0x03
G_TX_H_val      EQU     0xD2
G_TX_L_val      EQU     0x4C
H_TX_H_val      EQU     0xE8
H_TX_L_val      EQU     0x60

A_PD_H_val      EQU     0x18
A_PD_L_val      EQU     0x2A
B_PD_H_val      EQU     0x23
B_PD_L_val      EQU     0x0B
C_PD_H_val      EQU     0x41
C_PD_L_val      EQU     0x51
D_PD_H_val      EQU     0x67
D_PD_L_val      EQU     0x71
E_PD_H_val      EQU     0x8A
E_PD_L_val      EQU     0x00
F_PD_H_val      EQU     0xB8
F_PD_L_val      EQU     0x03
G_PD_H_val      EQU     0xD2
G_PD_L_val      EQU     0x4C
H_PD_H_val      EQU     0xE8
H_PD_L_val      EQU     0x60


;*******************************************************************************
; These constants must not be changed
;*******************************************************************************

TABLE_PT_val            EQU     0x00
SAMPLE_HALF             EQU     0x03
```

```
SAMPLE_ALL              EQU     0x07


;*******************************************************************************
; These values must be changed when using different combinations of bit-rates
; and MCU clock frequencies. Sets the period of the internal TMR0 interrupt,
; used in both RX- and TX mode.
;*******************************************************************************

;*******************************************************************************
;* Fosc = 20 MHz                                                               *
;*******************************************************************************
;* Data rate * TIMING_RX  * Rate_RX * Error  * TIMING_TX  * Rate_TX * Error  *
;*******************************************************************************
;* 9.6 kbps  * 0xE5#      * 0xD0    * 0.160% * 0x84       * 0xD0    * 0.224% *
;* 4.8 kbps  * 0xC5       * 0xD0    * 0.160% * 0x02       * 0xD0    * 0.032% *
;* 2.4 kbps  * 0x84       * 0xD0    * 0.160% * 0xFF       * 0xD1    * 0.064% *
;* 1.2 kbps  * 0x02       * 0xD0    * 0.160% * 0x7F       * 0xD3    * 0.304% *
;* 600 bps   * 0xFF       * 0xD1    * 0.160% * 0x7E       * 0xD4    * 0.152% *
;* 300 bps   * 0xFD       * 0xD2    * 0.032% * 0x7E       * 0xD5    * 0.004% *
;*******************************************************************************
; # Use one NOP instruction in end_interrupt_RX, otherwise remove it.
;
; For other combinations of MCU clock frequencies and data-rates, see
; application note for values and formulas.
;
;*******************************************************************************
; Default values
;       20 MHz MCU clock, 1.2 kbps data rate
;*******************************************************************************

TIMING_RX               EQU     0x02
TIMING_TX               EQU     0x7F
Rate_RX                 EQU     0xD0
Rate_TX                 EQU     0xD3


;*******************************************************************************
; These timing values values can be changed to optimize synchronisation and
; data decision for a given application
;
;       A_LIMIT_val     - Decision limit, if A is smaller than the
;                         value, signal is 0, else signal is 1
;       A_LIMIT_val2            - Decision limit while synchronising, if
;                         A is smaller than the value, signal is
;                         0, else signal is 1 (Must be >=5)
;       COUNT_LIMIT_val         - Decides how long the synchronisation
;                         part should be. The value indicates how
;                         many 1's (A=>A_LIMIT_val2) the MCU must
;                         receive before it is synchronised (Should be >=4)
;       A_SYNC_LIMIT_hi,
;       A_SYNC_LIMIT_lo - If A>lo and A<hi for SYNC_LOSS_LIMIT bits, the
;                         receiver declares loss of synchronisation
;       BIT_ERROR_LIMIT_val
;                       - How many low-quality bits are
;                         accepted before sync is lost
;                         (Should be >=2)
;*******************************************************************************

A_LIMIT_val             EQU     0x04
A_LIMIT_val2            EQU     0x05
COUNT_LIMIT_val                 EQU     0X0A
A_SYNC_LIMIT_hi                 EQU     0x05
A_SYNC_LIMIT_lo                 EQU     0x03
BIT_ERROR_LIMIT_val     EQU     0x02



;*******************************************************************************
;*******************************************************************************
;****************** PROGRAM START **********************************************
;*******************************************************************************
;*******************************************************************************

            ORG     0x00            ; Location of reset vector
reset:      GOTO    main            ; After reset, go to main program
```

```
            ORG     0X04              ; Location of interrupt vector
interrupt:  GOTO    ServiceInterrupt  ; If interrupt->go to ServiceInterrupt
            ORG     0x50


;*******************************************************************************
; Main program
;*******************************************************************************

main:
        BCF     STATUS, RP0
        BCF     STATUS, RP1           ; Select Bank0
        CLRF    PORTC                 ; Clear output data latches
        CLRF    PORTB                 ; Clear output data latches
        BSF     STATUS, RP0           ; Select Bank1
        MOVLW   0x08                  ; PORTC: Pin 0, 1 og 2 as output
                                      ; (PDATA, CLOCK and STROBE)
        MOVWF   TRISC                 ; PORTC: Pin 3 as input (DIO)
        MOVLW   0X32                  ; PORTB: Pin 0 og 2 as output
                                      ; (CLK_OUT and DATA_OUT)
        MOVWF   TRISB                 ; PORTB: Pin 1, 4 og 5 as input
                                      ; (DATA_IN, RX/TX and PD)
        BCF     STATUS, RP0           ; Select Bank0
        GOTO    mode_decision         ; Go to mode_decision


;*******************************************************************************
; Interrupt handler
;*******************************************************************************

ServiceInterrupt:
        BTFSC   INTCON, RBIF          ; Check if external interrupt
                                      ; (PD or RX/TX)
        GOTO    mode_decision         ; If external interrupt ->
                                      ; go to mode_decision



interrupt_internal:
        BTFSC   MODE, RXTX            ; Check if system was in RX/TX-mode
                                      ; before interrupt
        GOTO    mode_RX_routine             ; If RXTX=1 ->go to mode_RX_routine
        GOTO    mode_TX_routine             ; If RXTX=0 ->go to mode_TX_routine

mode_decision:
        MOVF    PORTB, 0              ; Read PORTB
        MOVWF   TEMP                        ; Store PORTB in TEMP
        BTFSC   TEMP, 4               ; Check PD
        GOTO    mode_PD               ; If PD=1 -> go to mode_PD
        BTFSC   TEMP, 5               ; Check RX_TX
        GOTO    mode_RX               ; If RX_TX=1 ->go to mode_RX
        GOTO    mode_TX               ; If RX_TX=0 ->go to mode_TX



;*******************************************************************************
;****************** RX-MODE ****************************************************
;*******************************************************************************
; Initialisation of the RX-mode. The CC700/CC900 is configured and registers
; used in MCU are initialised. The TMR0-interrupt is scaled for the correct
; data-rate, and enabled. Waits for interrupt.
;*******************************************************************************

mode_RX:
        BSF     STATUS, RP0                 ; Select Bank1
        BSF     TRISC, DIO                  ; PORTC: Set DIO-pin as input
        BCF     STATUS, RP0           ; Select Bank0


CCX00_config_RX:                      ; Download configuration to CCX00 with
                                      ; CLOCK, PDATA og STROBE (table
                                      ; look-up: TABLE_RX)

       MOVLW   TABLE_PT_val           ; TABLE_PT initialy set to 0 ->
       MOVWF   TABLE_PT               ; Shows where in the table to look up
                                      ; (From 0 to 15).
       CLRF    REG_COUNTER            ; Counts the registers configured in
```

```
                                    ; CC700/CC900 (8 16bit registers).
                                    ; Initially 0.
        BCF     CONTROL, CONF_LOOP  ; Two loops are used.
                                    ; CONF_LOOP=0 -> loop_inner_RX_1
                                    ; CONF_LOOP=1 -> loop_inner_RX_2
        BCF     PORTC, STROBE               ; Set STROBE=0
        BSF     PORTC, CLOCK                ; Set CLOCK=1


loop_outer_RX:                      ; This loop reads values from TABLE_RX
                                    ; and stores them in CONFIG_REG.
        CLRF    BIT_COUNTER         ; Clear BIT_COUNTER. Counts the bits
                                    ; read from CONFIG_REG.
        CALL    Table_RX            ; Look-up table
        MOVWF   CONFIG_REG          ; Value read from TABLE_RX is
                                    ; stored in CONFIG_REG
        INCF    TABLE_PT, 1         ; Increment table pointer. Read next
                                    ; register next time.
        BTFSC   CONTROL, CONF_LOOP  ; Checks which loop to execute
        GOTO    loop_inner_RX_2             ; CONF_LOOP=0 -> loop_inner_RX_1.
                                    ; CONF_LOOP=1 -> loop_inner_RX_2


loop_inner_RX_1:                    ; Used for _H regisers from table.
                                    ; MSB of CC700/CC900 registers
        RLF     CONFIG_REG, 1       ; Rotate CONFIG_REG.
                                    ; Most significant bit moves to CARRY
        BTFSC   STATUS, CARRY       ; Check if PDATA should be high
        BSF     PORTC, PDATA                ; If high -> Set PDATA=1
        BTFSS   STATUS, CARRY       ; Check if PDATA should be low
        BCF     PORTC, PDATA        ; If low -> Set PDATA=0
        BCF     PORTC, CLOCK        ; Set CLOCK=0. For 50% duty cycle
                                    ; configuration clock -> Include 4 NOP
        INCF    BIT_COUNTER, 1      ; Increase BIT_COUNTER. Another bit is
                                    ; read.
        BSF     PORTC, CLOCK        ; Set CLOCK=1
        BTFSS   BIT_COUNTER, 3      ; Check if all 8 bits in register
                                    ; CONFIG_RX have been read.
        GOTO    loop_inner_RX_1             ; If not all 8 bits have been read ->
                                    ; Go to loop_inner_RX_1
        BSF     CONTROL, CONF_LOOP  ; This loop is done. Execute
                                    ; loop_inner_RX_2 next time.
        GOTO    loop_outer_RX       ; If all 8 bits have been read ->
                                    ; Go to loop_outer_RX


loop_inner_RX_2:                    ; Used for _L regisers from table.
                                    ; LSB of CC700/CC900 registers
        RLF     CONFIG_REG, 1       ; Rotate CONFIG_REG.
                                    ; Most significant bit moves to CARRY
        BTFSC   STATUS, CARRY       ; Check if PDATA should be high
        BSF     PORTC, PDATA        ; If high -> Set PDATA=1
        BTFSS   STATUS, CARRY       ; Check if PDATA should be low
        BCF     PORTC, PDATA        ; If low -> Set PDATA=0
        BCF     PORTC, CLOCK        ; Set CLOCK=0.
        INCF    BIT_COUNTER, 1      ; Increase BIT_COUNTER. Another bit is
                                    ; read.
        BTFSC   BIT_COUNTER, 3      ; Check if all 8 registers have
                                    ; been read.
        GOTO    set_strobe_RX       ; If all 8 bits have been read ->
                                    ; Go to set_strobe_RX.
        BSF     PORTC, CLOCK        ; Set CLOCK=1
        GOTO    loop_inner_RX_2             ; If not all 8 bits are read ->
                                    ; Go to loop_inner_RX_2


set_strobe_RX:
        BSF     PORTC, STROBE       ; Set STROBE=1
        BSF     PORTC, CLOCK                ; Set CLOCK=1
        BCF     PORTC, STROBE       ; Set STROBE=0
        INCF    REG_COUNTER, 1      ; Increase REG_COUNTER. An entire 16
                                    ; bit register in the CC700/CC900 has
                                    ; been configured.
        BCF     CONTROL, CONF_LOOP  ; Do loop_inner_RX_1 next time
        BTFSS   REG_COUNTER, 3              ; Check if all 8 registers in
                                    ; CC700/C900 have been configured
        GOTO    loop_outer_RX       ; If not-> Go to loop_outer_RX
```

```
        BCF     PORTC, STROBE           ; Set STROBE=0
        BCF     PORTC, CLOCK            ; Set CLOCK=0


init_RX:                                        ; Give registers initial values
        CLRF    X                       ; X=00000000
        MOVLW   0x08
        MOVWF   A                       ; A=1000 (LSB)
        MOVWF   An1                     ; An1=1000 (LSB)
        MOVWF   An2                     ; An2=1000 (LSB)
        CLRF    R                       ; R=000 (LSB)
        CLRF    COUNT                   ; COUNT=000 (LSB)
        CLRF    CONTROL                         ; CONTROL=00 (LSB)
        MOVLW   A_LIMIT_val
        MOVWF   A_LIMIT                         ; A_LIMIT=A_LIMIT_val
        MOVLW   A_LIMIT_val2
        MOVWF   A_LIMIT2                ; A_LIMIT2=A_LIMIT_val2
                                        ; (>4, Must not sync on 000..)
        MOVLW   COUNT_LIMIT_val
        MOVWF   COUNT_LIMIT             ; COUNT_LIMIT=COUNT_LIMIT_val
        CLRF    INPUT                   ; INPUT=0 (LSB)
        BSF     MODE, RXTX              ; Set RXTX=1 -> shows RX-mode after
                                        ; TMR0-interrupt



enable_interrupt_RX:                    ; Enables interrupt in RX-mode
        MOVLW  0xFD
        MOVWF   TMR0                    ; Change TMR0 register value (first
                                        ; interrupt after just a short while)
        BSF     STATUS, RP0             ; Select Bank1
        MOVLW   Rate_RX
        MOVWF   OPTION_REG              ; Set TIMER0 Rate
        BCF     STATUS, RP0             ; Select Bank0
        MOVLW   0xA8
        MOVWF   INTCON                  ; Enables interrupts (external and
                                        ; internal) -> GIE=1, T0IE=1, RBIE=1,
                                        ; others=0 (incl RBIF OG T01F)


wait_for_interrupt:

                                        ; Update sync pin

        BTFSS   CONTROL,SYNC            ; If SYNC bit is set,
        BCF     PORTC, 4                ; set SYNC pin
        BTFSC   CONTROL,SYNC            ; If SYNC bit is cleared,
        BSF     PORTC, 4                ; clear SYNC pin

        GOTO wait_for_interrupt                 ; Loop. Wait for interrupt.
                                        ; Return here after TMR0-interrupt


;*******************************************************************************
;*********** TMR0-INTERRUPT IN RX-MODE ******************************************
;*******************************************************************************
;
; Each time an interrupt occurs, the DIO-pin is sampled.
; 8 interrupts (samples) per bit. Resynchronisation is done after each bit.
;*******************************************************************************

mode_RX_routine:                        ; TMR0-interrupt routine in RX-mode
        MOVLW  TIMING_RX
        MOVWF   TMR0                    ; TMR0=TIMING_TX
        BCF     INPUT, DI               ; Clear DI bit in INPUT-register
        BTFSC   PORTC, DIO              ; Check DIO-pin
        BSF     INPUT, DI               ; If DIO=1 -> DI=1



store_two_last_A:                       ; Store two last values of A
        MOVF    An1, 0
        MOVWF   An2                     ; An2=An1
        MOVF    A, 0
        MOVWF   An1                     ; An1=A
```

```
calculate_A:                            ; Calculate new A
        BCF     STATUS, CARRY           ; Clear CARRY
        BTFSC   INPUT, DI               ; Check DI (DIO-pin)
        BSF     STATUS, CARRY           ; If DIO was 1 -> CARRY=1
        RRF     X, 1                    ; Shift DIO value into X (from right)
        MOVLW   0X08
        XORWF   X, 1                    ; Inverts bit nr 4 in X
        BTFSS   X, 7                    ; If X7=0 -> Increment A
        INCF    A, 1
        BTFSS   STATUS, CARRY           ; If CARRY=0 -> Decrement A
        DECF    A, 1
        BTFSC   X, 3                    ; If X3=1 -> Decrement A
        DECF    A, 1
        BTFSS   X, 3                    ; If X3=0 -> Increment A
        INCF    A, 1


sample_number:
        BTFSC   CONTROL, R_ENABLE       ; Check if R_ENABLE=1
        GOTO    resync                  ; If R_ENABLE=1 -> Go to resync
        MOVF    R, 0
        XORLW   SAMPLE_ALL              ; R xor 00000111
        BTFSC   STATUS, Z               ; Check if R=111
        GOTO    sync                    ; If R=111 -> Go to sync
        MOVF    R, 0
        XORLW   SAMPLE_HALF             ; R xor 00000011
        BTFSC   STATUS, Z               ; Check if R=011
        BSF     PORTB, CLK_OUT                  ; If R=011 -> Set CLK_OUT =1
        INCF    R, 1                    ; R=R+1
        GOTO    end_interrupt_RX


sync:
        BTFSC   CONTROL, SYNC           ; Check SYNC-bit in CONTROL-register
        GOTO    sync_ok                         ; If SYNC=1 -> Go to sync_ok
        MOVF    A_LIMIT2, 0
        SUBWF   A, 0                    ; A-A_LIMIT2
        MOVWF   TEMP                    ; Store difference in TEMP
        BTFSS   TEMP, 7                         ; Check if TEMP is pos/neg
                                        ; (A>/<A_LIMIT)
        GOTO    sync_A_upper            ; If A>=A_LIMIT2 (DIO=1) -> Go to
                                        ; sync_A_upper
        CLRF    COUNT                   ; If A<A_LIMIT -> clear COUNT
        CLRF    R                       ; Clear R
        BSF     CONTROL, R_ENABLE       ; Set R_ENABLE=1
        GOTO    end_interrupt_RX



sync_ok:                                ; Detect loss of sync
        MOVF    A, 0
        SUBLW   A_SYNC_LIMIT_hi                 ; W=A_SYNC_LIMIT_hi-A
        MOVWF   TEMP                    ; Store difference in TEMP
        BTFSC   TEMP,7                  ; Skip if result positive
        GOTO    signal_OK               ; Result is negative, signal is OK
        MOVF    A,0
        SUBLW   A_SYNC_LIMIT_lo                 ; W=A_SYNC_LIMIT_lo-A
        MOVWF   TEMP                    ; Store difference in TEMP
        BTFSS   TEMP,7                  ; Skip if result negative
        GOTO    signal_OK               ; Result is positive, signal is OK

                                        ; Signal is not OK
        INCF    BIT_ERRORS,1            ; Increase number of detected errors
        MOVF    BIT_ERRORS,0
        SUBLW   BIT_ERROR_LIMIT_val     ; W=BIT_ERROR_LIMIT_val-BIT_ERRORS
        BTFSS   STATUS,Z                ; Skip if zero
        GOTO    sync_ok2                ; Continue

                                        ; Reached error limit
        BCF     CONTROL,SYNC            ; Clear sync bit

signal_OK:
        CLRF    BIT_ERRORS              ; Clear errors

sync_ok2:
        SUBWF   A, 0                    ; A-A_LIMIT
```

```
        MOVWF   TEMP                    ; Store difference in TEMP
        BTFSS   TEMP, 7                         ; Check if TEMP is pos/neg
                                        ; (A>/<A_LIMIT)
        GOTO    data_out_high           ; If A>=A_LIMIT (DIO=1) ->
                                        ; Go to data_out_high
        BCF     PORTB, DATA_OUT                 ; If A<A_LIMIT (DIO=1) ->
                                        ; Set DATA_OUT=0
        BCF     PORTB, CLK_OUT                  ; Set CLK_OUT=0
        CLRF    R                       ; Clear R
        BSF     CONTROL, R_ENABLE       ; Sett R_ENABLE=1
        GOTO    end_interrupt_RX


data_out_high:
        BSF     PORTB, DATA_OUT                 ; Set DATA_OUT=1
        BCF     PORTB, CLK_OUT                  ; Set CLK_OUT=0
        CLRF    R                       ; Clear R
        BSF     CONTROL, R_ENABLE       ; Set R_ENABLE=1
                                        ; (resync next interrupt)
        NOP
        GOTO    end_interrupt_RX


sync_A_upper:                           ; Synchronise; bit received is a 1
        MOVF    COUNT_LIMIT, 0
        SUBWF   COUNT, 0                ; COUNT-COUNT_LIMIT
        BTFSC   STATUS, Z               ; Z=0 if COUNT=COUNT_LIMIT
        BSF     CONTROL, SYNC           ; If COUNT=COUNT_LIMIT -> Set SYNC=1
        INCF    COUNT, 1                ; COUNT=COUNT+1
        CLRF    R                       ; Clear R
        BSF     CONTROL, R_ENABLE       ; Set R_ENABLE=1
        NOP
        GOTO    end_interrupt_RX


resync:
        MOVF    An1, 0
        SUBWF   An2, 0                  ; An2-An1
        MOVWF   TEMP1                   ; Store difference in TEMP1
        BTFSS   TEMP1, 7                ; Check if TEMP1 is pos/neg
                                        ; (An2>/<An1)
        GOTO    resync_1                ; If An2>=An1 -> Go to resync_1
        MOVF    A, 0
        SUBWF   An1, 0                  ; An1-A
        MOVWF   TEMP2                   ; Store difference in TEMP2
        BTFSC   TEMP2, 7                ; Check if TEMP2 is pos/neg (An1>/<A)
        GOTO    resync_2                ; If An1<A -> Go to resync_2
        INCF    R, 1                    ; R=R+1 (no resync)
        GOTO    end_interrupt_RX2


resync_1:
        MOVF    A, 0
        SUBWF   An1, 0                  ; An1-A
        MOVWF   TEMP2                   ; Store difference in TEMP2
        BTFSS   TEMP2, 7                ; Check if TEMP2 is pos/neg (An1>/<A)
        GOTO    resync_3                ; If An1>=A -> Go to resync_3
        BTFSC   CONTROL, SYNC           ; Check SYNC-bit in CONTROL-register
        INCF    R, 1                    ; If SYNC=1 -> R=R+1 (no resync)
        GOTO    end_interrupt_RX2


resync_2:
        NOP
        BTFSS   CONTROL, SYNC           ; Check SYNC-bit in CONTROL-register
        GOTO    end_interrupt_RX2       ; If SYNC=0 -> Go to end_interrupt_RX
        MOVF    A_LIMIT, 0
        SUBWF   An1, 0                  ; A-A_LIMIT
        MOVWF   TEMP                    ; Store difference in TEMP
        BTFSC   TEMP, 7                         ; Check if TEMP is pos/neg
                                        ; (An1>/<A_LIMIT)
        GOTO    r_add_2                         ; If not An1>=A_LIMIT -> Go to r_add_2
        GOTO    end_interrupt_RX2
```

```
resync_3:
        MOVF    TEMP1, 1            ; TEMP1 written to itself
                                    ; (Z is affected)
        BTFSC   STATUS, Z           ; Check if An2=An1
        GOTO    no_resync           ; If An2=An1 -> Go_to no_resync
        MOVF    TEMP2, 1            ; TEMP2 written to itself
                                    ; (Z is affected)
        BTFSC   STATUS, Z           ; Check if An1=A
        GOTO    no_resync           ; If An1=A -> Go_to no_resync
        MOVF    A_LIMIT, 0
        SUBWF   An1, 0              ; A-A_LIMIT
        MOVWF   TEMP                ; Store difference in TEMP
        BTFSS   TEMP, 7                     ; Check if TEMP is pos/neg
                                    ; (An1>/<A_LIMIT)
        GOTO    r_add_2                     ; If An1>=A_LIMIT -> Go to r_add2
       GOTO    end_interrupt_RX2


no_resync:
        INCF    R, 1                ; R=R+1 (no resync)
       GOTO    end_interrupt_RX2

r_add_2:
        INCF    R, 1
        INCF    R, 1                ; R=R+2 (resync)
        NOP


end_interrupt_RX2:
       BCF     CONTROL, R_ENABLE    ; Set R_ENABLE=0

end_interrupt_RX:
       NOP                          ; Use only NOP when datarate= 9.6 kbps
       BCF     INTCON, T0IF         ; Set T0IF=0 -> Ready for next
                                    ; TMR0-interrupt
       RETFIE                       ; Return from interrupt


;*******************************************************************************
;****************** TX-MODE ****************************************************
;*******************************************************************************
;
; Initialisation of the TX-mode. CCX00 is configured and registers used in
; the MCU are initialised. The TMR0-interrupt is scaled for the correct data-
; rate, and enabled. Waits for interrupt.
;*******************************************************************************

mode_TX:
        BSF     STATUS, RP0         ; Select Bank1
        BCF     TRISC, DIO          ; PORTC: Set DIO-pin as output
        BCF     STATUS, RP0         ; Select Bank0


CCX00_config_TX:                    ; Download configuration to
                                    ; CC700/CC900 using CLOCK, PDATA
                                    ; and STROBE (table look-up: TABLE_TX)
       MOVLW   TABLE_PT_val         ; TABLE_PT is initially set to 0 ->
       MOVWF   TABLE_PT             ; Shows where in the table to look up
                                    ; (From 0 to 15).
       CLRF    REG_COUNTER          ; Counts the regisers configured
                                    ; in the CC700/CC900 (8 16bit
                                    ; registers). Initially 0.
       BCF     CONTROL, CONF_LOOP   ; Two loops are used.
                                    ; CONF_LOOP=0 -> loop_inner_TX_1
                                    ; CONF_LOOP=1 -> loop_inner_TX_2
       BCF     PORTC, STROBE        ; Set STROBE=0
        BSF     PORTC, CLOCK        ; Set CLOCK=1


loop_outer_TX:                      ; This loop reads values from TABLE_TX
                                    ; and stores them in CONFIG_REG.
       CLRF    BIT_COUNTER          ; Clears BIT_COUNTER. Counts the bits
                                    ; read from CONFIG_REG.
       CALL    Table_TX             ; Look-up table
```

```
        MOVWF   CONFIG_REG              ; Value read from TABLE_TX is
                                        ; stored in CONFIG_REG
        INCF    TABLE_PT, 1             ; Increment table pointer.
                                        ; Read next register next time.
        BTFSC   CONTROL, CONF_LOOP      ; Checks which loop to execute
        GOTO    loop_inner_TX_2                 ; CONF_LOOP=0 -> loop_inner_TX_1
                                        ; CONF_LOOP=1 -> loop_inner_TX_2


loop_inner_TX_1:                        ; Used for _H regisers from table.
                                        ; MSB of CC700/CC900 registers
        RLF     CONFIG_REG, 1           ; Rotate CONFIG_REG.
                                        ; Most significant bit moves to CARRY
        BTFSC    STATUS, CARRY          ; Check if PDATA should be high
        BSF      PORTC, PDATA           ; If high -> Set PDATA=1
        BTFSS    STATUS, CARRY          ; Check if PDATA should be low
        BCF      PORTC, PDATA           ; If low -> Set PDATA=0
        BCF      PORTC, CLOCK           ; Set CLOCK=0. For 50% duty-cycle
                                        ; configuration clock -> Include 4 NOP
        INCF    BIT_COUNTER, 1          ; Increase BIT_COUNTER. Another bit
                                        ; is read.
        BSF      PORTC, CLOCK           ; Set CLOCK=1
        BTFSS   BIT_COUNTER, 3          ; Check if all 8 bits in register
                                        ; CONFIG_TX have been read.
        GOTO    loop_inner_TX_1                 ; If not all 8 bits have been read ->
                                        ; Go to loop_inner_TX_1
        BSF     CONTROL, CONF_LOOP      ; This loop is done. Execute
                                        ; loop_inner_RX_2 next time.
        GOTO    loop_outer_TX           ; If all 8 bits have been read ->
                                        ; Go to loop_outer_TX


loop_inner_TX_2:                        ; Used for _L registers from table.
                                        ; LSB of CC700/CC900 registers
        RLF     CONFIG_REG, 1           ; Rotate CONFIG_REG.
                                        ; Most significant bit moves to CARRY
        BTFSC   STATUS, CARRY           ; Check if PDATA should be high
        BSF     PORTC, PDATA            ; If high -> Set PDATA=1
        BTFSS   STATUS, CARRY           ; Check if PDATA should be low
        BCF     PORTC, PDATA            ; If low -> Set PDATA=0
        BCF     PORTC, CLOCK            ; Set CLOCK=0.
        INCF    BIT_COUNTER, 1          ; Increase BIT_COUNTER. Another bit
                                        ; is read.
        BTFSC   BIT_COUNTER, 3          ; Check if all 8 registers have been
                                        ; read.
        GOTO    set_strobe_TX           ; If all 8 bits have been read ->
                                        ; Go to set_strobe_TX.
        BSF     PORTC, CLOCK            ; Set CLOCK=1
        GOTO    loop_inner_TX_2                 ; If not all 8 bits have been read ->
                                        ; Go to loop_inner_TX_2


set_strobe_TX:
        BSF      PORTC, STROBE          ; Set STROBE=1
        BSF      PORTC, CLOCK           ; Set CLOCK=1
        BCF      PORTC, STROBE          ; Set STROBE=0
        INCF    REG_COUNTER, 1          ; Increase REG_COUNTER, an entire
                                        ; 16 bit register in the CC700/CC900
                                        ; has been configured.
        BCF     CONTROL, CONF_LOOP      ; Do loop_inner_TX_1 next time
        BTFSS   REG_COUNTER, 3          ; Check if all 8 registers in the
                                        ; CC700/CC900 has been configured
        GOTO    loop_outer_TX           ; If not-> Go to loop_outer_TX
        BCF      PORTC, STROBE          ; Set STROBE=0
         BCF      PORTC, CLOCK          ; Set CLOCK=0


init_TX:                                ; Gives registers initial values
         BCF      MODE, RXTX            ; Set RXTX=0 -> shows TX-mode
                                        ; after TMR0-interrupt
         BCF      MODE, NUMBER          ; Set NUMBER=0 (Decides which (of 2)
                                        ; interrupt to execute in TX-mode)
        CLRF    PORTB                   ; Clear PORTB


enable_interrupt_TX:                    ; Enables interrupt in TX-mode
        MOVLW   0xFC
```

```
        MOVWF   TMR0
        BSF     STATUS, RP0           ; Select Bank1
        MOVLW   Rate_TX
        MOVWF   OPTION_REG            ; Set TIMER0 Rate
        BCF     STATUS, RP0           ; Select Bank0
        MOVLW   0xA8
        MOVWF   INTCON                ; Enables interrupts (external and
                                      ; internal) -> GIE=1, T0IE=1, RBIE=1,
                                      ; Others=0 (incl RBIF and T01F)
        GOTO    wait_for_interrupt


;*******************************************************************************
;*********** TMR0-INTERRUPT IN TX-MODE ******************************************
;*******************************************************************************
; Two interrups occur per bit, one for each baud of the outgoing Manchester
; coded signal at the DIO-pin.
;*******************************************************************************

mode_TX_routine:                      ; TMR0-interrupt routine in TX-mode
        MOVLW   TIMING_TX
        MOVWF   TMR0                  ; TMR0=TIMING_TX
        BTFSC   MODE, NUMBER          ; Check which interrupt to execute
        GOTO    second_TX             ; If NUMBER=1 -> Go to second_TX
        GOTO    first_TX              ; If NUMBER=0 -> Go to first_TX


first_TX:                             ; First baud
        BSF     PORTB, CLK_OUT                ; Set CLK_OUT=1
        BCF     MODE, READ            ; Set READ=0 in MODE-register
        BTFSC   PORTB, DATA_IN                ; Check DATA_IN-pin
        BSF     MODE, READ            ; If DATA_IN=1 -> Set READ=1
        BTFSC   MODE, READ            ; Check READ
        BSF     PORTC, DIO            ; If READ=1 (DATA_IN=1) ->
                                      ; Set DIO=1 (First baud)
        BTFSS   MODE, READ            ; Check READ
        BCF     PORTC, DIO            ; If READ=0 (DATA_IN=0) ->
                                      ; Set DIO=0 (First baud)
        BSF     MODE, NUMBER          ; Set NUMBER=1
        GOTO    end_interrupt_TX


second_TX:                            ; Second baud
        BCF     PORTB, CLK_OUT                ; Set CLK_OUT=0
        BTFSC   MODE, READ            ; Check READ
        BCF     PORTC, DIO            ; If READ=1 (DATA_IN=1) ->
                                      ; Set DIO=0 (Second baud)
        BTFSS   MODE, READ            ; Check READ
        BSF     PORTC, DIO            ; If READ=0 (DATA_IN=0) ->
                                      ; Set DIO=1 (Second baud)
        BCF     MODE, NUMBER          ; Set NUMBER=0


end_interrupt_TX:
        NOP
        BCF     INTCON, T0IF          ; Set T0IF=0 ->
                                      ; Ready for next TMR0-interrupt
        RETFIE                        ; Return from interrupt


;*******************************************************************************
;******************** PD-MODE **************************************************
;*******************************************************************************
; Initialisation of the PD-mode. The CC700/CC900 is configured and the MCU
; enters sleep mode. Only the external interrupt is enabled (mode change)
;*******************************************************************************

mode_PD:


CCX00_config_PD:                      ; Download configuration to CCX00
                                      ; using CLOCK, PDATA og STROBE
                                      ; (table look-up: TABLE_PD)
        MOVLW   TABLE_PT_val          ; TABLE_PT initially set to 0 ->
        MOVWF   TABLE_PT              ; Shows where in the table to look up
```

```
                                        ; (From 0 to 15).
        CLRF    REG_COUNTER             ; Counts the registers configured in
                                        ; the CC700/CC900 (8 16bit registers)
                                        ; Initially 0.
        BCF     CONTROL, CONF_LOOP      ; Two loops are used.
                                        ; CONF_LOOP=0 -> loop_inner_PD_1
                                        ; CONF_LOOP=1 -> loop_inner_PD_2
        BCF     PORTC, STROBE           ; Set STROBE=0
         BSF    PORTC, CLOCK            ; Set CLOCK=1


loop_outer_PD:                          ; This loop reads values from TABLE_PD
                                        ; and stores them in CONFIG_REG.
        CLRF    BIT_COUNTER             ; Clears BIT_COUNTER. Counts the bits
                                        ; read from CONFIG_REG.
        CALL    Table_PD                ; Look-up-table
        MOVWF   CONFIG_REG              ; Value read from TABLE_RX is stored
                                        ; in CONFIG_REG
        INCF    TABLE_PT, 1             ; Increment table pointer.
                                        ; Read next register next time.
        BTFSC   CONTROL, CONF_LOOP      ; Checks which loop to execute
        GOTO    loop_inner_PD_2                 ; CONF_LOOP=0 -> loop_inner_PD_1
                                        ; CONF_LOOP=1 -> loop_inner_PD_2


loop_inner_PD_1:                        ; Used for _H registers from table.
                                        ; MSB of CC700/CC900 register
        RLF     CONFIG_REG, 1           ; Rotate CONFIG_REG.
                                        ; Most significant bit moves to CARRY
        BTFSC    STATUS, CARRY          ; Check if PDATA should be high
        BSF      PORTC, PDATA           ; If high -> Set PDATA=1
        BTFSS    STATUS, CARRY          ; Check if PDATA should be low
        BCF      PORTC, PDATA           ; If low -> Set PDATA=0
        BCF      PORTC, CLOCK           ; Set CLOCK=0. For 50% duty cycle
                                        ; configuration clock -> Include 4 NOP
        INCF    BIT_COUNTER, 1          ; Increase BIT_COUNTER. Another bit
                                        ; is read.
        BSF      PORTC, CLOCK           ; Set CLOCK=1
        BTFSS   BIT_COUNTER, 3          ; Check if all 8 bits in register
                                        ; CONFIG_PD have been read.
        GOTO    loop_inner_PD_1                 ; If not all 8 bits have been read ->
                                        ; Go to loop_inner_PD_1
        BSF     CONTROL, CONF_LOOP      ; This loop is done. Execute
                                        ; loop_inner_PD_2 next time.
        GOTO    loop_outer_PD           ; If all 8 bits have been read ->
                                        ; Go to loop_outer_PD


loop_inner_PD_2:                        ; Used for _L registers from table.
                                        ; LSB of CC700/CC900 register
        RLF     CONFIG_REG, 1           ; Rotate CONFIG_REG.
                                        ; Most significant bit moves to CARRY
        BTFSC    STATUS, CARRY          ; Check if PDATA should be high
        BSF      PORTC, PDATA           ; If high -> Set PDATA=1
        BTFSS    STATUS, CARRY          ; Check if PDATA should be low
        BCF      PORTC, PDATA           ; If low -> Set PDATA=0
        BCF      PORTC, CLOCK           ; Set CLOCK=0.
        INCF    BIT_COUNTER, 1          ; Increase BIT_COUNTER. Another bit
                                        ; is read.
        BTFSC   BIT_COUNTER, 3          ; Check if all 8 registers have
                                        ; been read.
        GOTO    set_strobe_PD           ; If all 8 bits have been read ->
                                        ; Go to set_strobe_PD.
        BSF      PORTC, CLOCK           ; Set CLOCK=1
        GOTO    loop_inner_PD_2                 ; If not all 8 bits have been read ->
                                        ; Go to loop_inner_PD_2


set_strobe_PD:
        BSF      PORTC, STROBE          ; Set STROBE=1
        BSF      PORTC, CLOCK           ; Set CLOCK=1
        BCF      PORTC, STROBE          ; Set STROBE=0
        INCF    REG_COUNTER, 1          ; Increase REG_COUNTER. An entire
                                        ; 16 bit register in the CC700/CC900
                                        ; has been configured.
        BCF     CONTROL, CONF_LOOP      ; Do loop_inner_PD_1 next time
        BTFSS   REG_COUNTER, 3          ; Check if all 8 registers in the
```

```
                                    ; CC700/CC900 are configured
        GOTO    loop_outer_PD       ; If not-> Go to loop_outer_PD
        BCF     PORTC, STROBE       ; Set STROBE=0
        BCF     PORTC, CLOCK        ; Set CLOCK=0


init_PD:                            ; Clear all output pins
        CLRF    PORTC               ; Clear output data latches
        BCF     PORTB, CLK_OUT      ; Set CLK_OUT=0
        BCF     PORTB, DATA_OUT             ; Set DATA_OUT=0


enable_interrupt_PD:
        MOVLW   0x88
        MOVWF   INTCON              ; Enables interrupt (external) ->
                                    ; Sets GIE=1, RBIE=1,
                                    ; the rest are 0 (including RBIF)


wait_for_interrupt_PD:              ; Wait for External interrupt on PORTB
        SLEEP                       ; MCU in sleep mode
        NOP                         ; Does nothing before an interrupt
                                    ; occurs
        GOTO    wait_for_interrupt_PD

;*************************************************************************
; Look-up tables used for configuration of CC700/CC900
; 3 tables, one for each possible mode: PD, RX and TX
;*************************************************************************

        ORG     0x06
Table_RX:
        MOVFW           TABLE_PT
        ADDWF           PCL, 1
        RETLW           A_RX_H_val
        RETLW           A_RX_L_val
        RETLW           B_RX_H_val
        RETLW           B_RX_L_val
        RETLW           C_RX_H_val
        RETLW           C_RX_L_val
        RETLW           D_RX_H_val
        RETLW           D_RX_L_val
        RETLW           E_RX_H_val
        RETLW           E_RX_L_val
        RETLW           F_RX_H_val
        RETLW           F_RX_L_val
        RETLW           G_RX_H_val
        RETLW           G_RX_L_val
        RETLW           H_RX_H_val
        RETLW           H_RX_L_val


        ORG     0x18
Table_TX:
        MOVFW           TABLE_PT
        ADDWF           PCL, 1
        RETLW           A_TX_H_val
        RETLW           A_TX_L_val
        RETLW           B_TX_H_val
        RETLW           B_TX_L_val
        RETLW           C_TX_H_val
        RETLW           C_TX_L_val
        RETLW           D_TX_H_val
        RETLW           D_TX_L_val
        RETLW           E_TX_H_val
        RETLW           E_TX_L_val
        RETLW           F_TX_H_val
        RETLW           F_TX_L_val
        RETLW           G_TX_H_val
        RETLW           G_TX_L_val
        RETLW           H_TX_H_val
        RETLW           H_TX_L_val


        ORG     0x2A
Table_PD:
```

```
        MOVFW           TABLE_PT
        ADDWF           PCL, 1
        RETLW           A_PD_H_val
        RETLW           A_PD_L_val
        RETLW           B_PD_H_val
        RETLW           B_PD_L_val
        RETLW           C_PD_H_val
        RETLW           C_PD_L_val
        RETLW           D_PD_H_val
        RETLW           D_PD_L_val
        RETLW           E_PD_H_val
        RETLW           E_PD_L_val
        RETLW           F_PD_H_val
        RETLW           F_PD_L_val
        RETLW           G_PD_H_val
        RETLW           G_PD_L_val
        RETLW           H_PD_H_val
        RETLW           H_PD_L_val

;*******************************************************************************
        END
;*******************************************************************************
```

This application note is written by the staff of Chipcon to the courtesy of our customers. Chipcon is a world-wide distributor of integrated radio transceiver chips. For further information on the products from Chipcon Components please contact us or visit our web site.

Contact Information

Address:
Chipcon AS
Gaustadalléen 21
N-0349 Oslo,
NORWAY

Telephone	:	(+47) 22 95 85 44
Fax	:	(+47) 22 95 85 46
E-mail	:	wireless@chipcon.com
Web site	:	http://www.chipcon.com

Disclaimer