IBEX
25kHz RFID Reader Driver Source Code Driver Project

CONTENTS

DRIVER OVERVIEW

FEATURES

Low cost 125kHz passive RFID tags are the ideal choice for all sorts of identification applications. However many engineers view the task of designing a 125kHz RFID reader as too daunting a job involving dabbling in the black arts of radio frequency design. Instead an engineer will typically opt for the safe approach of using one of the many off the shelf but expensive RFID reader OEM modules.

Actually designing a 125KHz RFID reader is very straightforward if a good RFID reader IC is used and this driver is designed to work with the excellent NXP (formerly Philips) HTRC110 RFID Reader Chip.

This driver provides complete functionality to read EM Marin EM4102 (previously named H4001/H4102) and compatible RFID transponders. The EM4100 is one of the more common data formats for RFID transponders, so named because the microchip at the heart of compatible tags are based on a controller chip originally made by the company EM Microelectronic. The protocol is typically used by the many types of generic 125kHz read only passive RFID tags available from a huge range of suppliers. These simple RFID transponders carry 40 bits of read only memory and are typically supplied with a globally unique ID value (the 40 bits provide 1,099,511,627,776 combinations).

The driver provides the following main features:-

Automatically handles the 64, 32 or 16 period data rates and Manchester or Biphase encoding methods used by EM4102 based tags. Most tags use Manchester encoding due to many RFID readers only supporting Manchester encoded tags.

Optimised for small code footprint embedded designs.

State machine and interrupt based implementation to avoid your application stalling while tags are read and the modulated data stream decoded.

Automatically switches the HTRC110 to low power mode between reading tags.

No reliance on compiler specific libraries.

Full source code supplied for you to use and modify as required.

DRIVER TECHNICAL NOTES

The driver is designed to support the NXP HTRC110 RFID reader chip and has been designed to work with 8-bit, 16-bit and 32-bit microcontrollers.

The driver has the following hardware requirements:

In order to decode the modulated RFID tag data stream the driver requires the RFID Dout connection to be made to a microcontroller rising edge interrupt pin. The driver carries out all other communications by driving and reading the microcontroller pins directly.

When a rising edge is detected the driver function that is called needs to be passed the time x1uS since the last rising edge interrupt occured. Therefore your microcontroller needs to have a timer you can use to allow your code to pass the time value.

USING THE DRIVER WITH A RTOS OR KERNEL

The driver is implemented as a single thread so you just need to make sure it is always called from a single thread (it is not designed to be thread safe). All RFID operations are carried out using a non stalling state machine approach.

ADDING THE DRIVER TO YOUR PROJECT

NOTES ABOUT OUR SOURCE CODE FILES

There are many different ways to organise your source code and many different opinions on the best method! We have chosen the following as a very good approach that is widely used, well suited to both small and large projects and simple to follow.

Each .c source code file has a matching .h header file. All function and memory definitions are made in the header file. The .c source code file only contains functions. The header file is separated into distinct sections to make it easy to find things you are looking for. The function and data memory definition sections are split up to allow the defining of local (this source code file only) and global (all source code files that include this header file) functions and variables. To use a function or variable from another .c source code file simply include the .h header file.

Variable types BYTE, WORD, SIGNED_WORD, DWORD, SIGNED_DWORD are used to allow easy compatibility with other compilers. A WORD is 16 bits and a DWORD is 32 bits. Our projects include a 'main.h' global header file which is included in every .c source code file. This file contains the typedef statements mapping these variable types to the compiler specific types. You may prefer to use an alternative method in which case you should modify as required. Our main.h header file also includes project wide global defines.

This is much easier to see in use than to try and explain and a quick look through one of the included sample projects will show you by example.

Please also refer to the resources section of the embedded-code.com web site for additional documentation which may be useful to you.

MODIFYING OUR PROJECT FILES

We may issue new versions of our source code files from time to time due to improved functionality, bug fixes, additional device / compiler support, etc. Where possible you should try not to modify our source codes files and instead call the driver functions from other files in your application. Where you need to alter the source code it is a good idea to consider marking areas you have changed with some form of comment marker so that if you need to use an upgraded driver file its as easy as possible to upgrade and still include all of the additions and changes that you have made.

STEP BY STEP INSTRUCTIONS

Move The Main Driver Files To Your Project Directory

The following files are the main driver files which you need to copy to your main project directory:

rfid.c – The RFID driver functions
rfid.h

Move The Generic Global Defines File To Your Project Directory

The generic global file is located in the driver sample project directory. Copy the following file to your main project directory:

main.h – The embedded-code.com generic global file:

Check Driver Definitions

Check the definitions in the following file to see if any need to be adjusted for the microcontroller / processor you are using, and your hardware connections:-

rfid.h

Check the definitions in the following file and adjust if necessary for your compiler:-

main.h

Interrupts

In order to decode the modulated RFID tag data stream the driver requires the RFID Dout connection to be made to a microcontroller rising edge interrupt pin (see example circuit schematic). In your application initialisation you should configure the interrupt ready for operation but leave it disabled.

The driver reduces interrupt demands to an absolute minimum, with the interrupt occurring at a maximum rate of once every 128uS when reading a valid RFID tag using the fastest bitrate option (most tags tend to use the slower and therefore longer range bitrate of 512uS).

In the interrupt handler you should provide code to determin the number of uS since the last rising edge interrupt (see rfid.h file for an example) and then call the rfid_sampling_rising_edge() function.

Timer

When readi
ng of a tag is active every time a rising edge interrupt occurs the driver function that is called needs to be passed the time x1uS since the last rising edge interrupt occured. Therefore in your application initiialisation your should configure a suitable timer so that you can use it to obtain this time value.

You will need to provide some form of timer for the driver. Typically this can be done in your applications general heartbeat timer if you have one. Do the following every 1mS:-

```
//----- RFID READER TIMER -----
if (rfid_1ms_timer)
      rfid_1ms_timer--;
```

If you do not have a matching timer then using a time base that is slightly greater than 1mS is fine.

APPLICATION REQUIREMENTS

In each .c file of your application that will use the driver functions include the ‘rfid.h’ file.

You will need to periodically call the drivers background processing function. Typically this can be done as part of your applications main loop. Add the following call:-

```
//----- PROCESS RFID DRIVER -----
rfid_process();
```

USING THE SAMPLE PROJECT

SAMPLE PROJECTS INCLUDED

A sample project is included with the driver for a specific devices and compilers. The example schematic at the end of this manual detail the circuit the sample project is designed to work with. You may use the sample projects with the circuit shown or use them as a starting block for your own project with a different device of compiler. To use them copy all of the files in the sample project directory into the same directory as the driver files and then open and run using the development environment / compiler the project was designed with.

Microchip C18 Compiler

Compiler: Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers

Device: PIC18F2420

SAMPLE PROJECT FUNCTIONS

When run the sample project will attempt to read a RFID tag every 250mS. If no tag is read the red LED is lit. If a tag is successfully read the green LED is lit.

USING THE DRIVER IN YOUR PROJECT

The driver is implemented as a single thread so you just need to make sure functions are always called from a single thread (it is not designed to be thread safe). All RFID operations are carried out using a non stalling state machine approach.

FUNCTIONS

The driver is implemented as a single thread so you just need to make sure functions are always called from a single thread (it is not designed to be thread safe). All RFID operations are carried out using a non stalling state machine approach.

void rfid_process (void)

This function should be called regularly as part of your applications main loop.

void rfid_read_tag (void)

Call this function to start the read tag process

BYTE rfid_get_read_tag_result (void)

Call this function to check for the end of the read tag process. It returns 0 if still reading, 1 if completed. Bit 7 will be set if the read failed.

After a sucessful read the ASCII tag ID will be in rfid_data_nibbles[0] | rfid_data_nibbles[9]

void rfid_sampling_rising_edge (WORD us_since_last_rising_edge)

Call with us_since_last_rising_edge = The time in uS since the last edge interrupt. You application interrupt hander must create this value from an available microcontroller timer. The maximum time between interrupts will be approximately 1024uS, so your timer calculation only needs to deal with time values of up to say 1280uS (to give a reasonable margin of worst case tollerance).

Call with the timer value captured as close as possible to the rising edge occuring (reduce latency as much as possible).

This fucntion will ignore the value for the first 2 calls allowing for initial timer error.

void rfid_force_initialise (void)

In typical use this function is not requried as the driver will automatically initialise the HTRC110 IC on powerup. However if you are using this driver with multiple HTRC110 devices (for instance by switching the SCLK, DIN and DOUT lines by using one or more 74HC4053 IC's) then this function can be called during your applications initialisation after selecting each HTRC110. To read each HTRC110 call rfid_read_tag() and wait for rfid_get_read_tag_result() to return a completed result before switching to the next HTRC.

TIMING REQUIREMENTS

The decoding of the modulated tag datastream requries the Dout rising edge interrupt to be handled with reasonably low latency so that the time x 1uS since the last rising edge can be passed to the drivers rfid_sampling_rising_edge() function with as much accuracy as possible. Depending on the RFID tags data rate (different tags are permitted to use different rates) the quickest interrupting will be a rising edge every 512uS, 256uS or 128uS. Many tags use the slowest 512uS bit rate as it helps increase range.

In the Microchip PIC18 sample project using an 8 bit microcontroller running with a 125nS instruction clock (8MHz crystal with 4 x PLL) the rfid_sampling_rising_edge() function takes a maximum of 95uS to complete (typcially significantly less – the most time is taken after parity bits are receveid). This is with no C18 compiler optimisations enabled.

The above should allow you to decide on a suitable interrupt arrangement for your microcontoller based on the other demands of your applicaiton. Remember that the rising edge interrupt is only active when reading a RFID tag. When the driver is idle the interrupt is disabled.

MEMORY SPACE REQUIREMENTS

Using the PIC18 Sample Project the driver uses approximately 2534 program memory bytes (1276 x 16 bit instructions) compiling just the driver functions with the Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers (with no compiler optimisations enabled).

The driver requries approximately 40 bytes of static RAM memory space. It requires only a small amount of temporary variable storage space from the stack.

TROUBLESHOOTING

A few things to check if you run into problems implementing the driver in your application:

Are the communicaitons between the HTRC110 and your microcontroller working correctly – does the HTRC110 reply? Use the RFID_SPI_SAMPLE_CS_PIN and a logic analyser to capture these if necessary.

Is the time value being passed to the rfid_sampling_rising_edge() function correct – check with a scope.

In the function rfid_sampling_rising_edge() breakpoint at the end of the section "//—— CALCULATE THE CYCLE TIMES ——". Is the rfid_1us_bit_rate correct? If you use a logic analyser to capture Dout from the HTRC110 during tag reading you will be able to determin the bit rate by looking for the shortest low high pulse sequence (high edge to high edge) – does rfid_1us_bit_rate match?

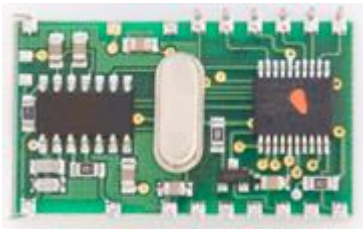
HARDWARE DESIGN NOTES

GENERAL DESIGN NOTES

The HTRC110 datasheet provides a detailed overview of the HTRC110 specifications. The HTRC110 Read/Write Devices Application Note provides very thorough information on how to implement a HTRC110 based circuit.

The application note can get very technical in places and its easy to become quite offput by the sections covering antennas etc if you are not one of the rare bread of engineers who is fully conversant with the black art that is RF design. However there is no need to worry if parts of it go over your head – the application note was of course written by RF engineers for RF engineers. The 125kHz being dealt with is not at all complex from a RF point of view and designs will generally work fine using the default component values and just by taking care with your antennas PCB routing to give it reasonably thick tracks and keep it away from sources of noise. At the end of the day you can make a fully functioning RFID reader using the HTRC110, this driver and a few loops of PCB copper as the antenna!

A great source of design help is to look at one of the many 125kHz RFID reader modules that use the HTRC110, available from many different suppliers (the HTRC110 is a very well known and commonly used RFID read solution). Looking at their antenna implementation notes will often provide the basic information you need to feel comfortable with your design. For example the Micro RWD QT (Quad Tag) manufactured by www.ibtechnology.co.uk uses the HTRC110 with a PIC16F87 microcontroller:



RFID Circuit & PCB Design Notes

It's important that the voltage supply to the HTRC110 is protected from ripple and interference while a read is being carried out.

Where you have noisy electronics in the proximity that are active while reading a tag:

A simple RC filter of 10R and 1000uF will go a long way to preventing power supply interference. Better still is using a separate power supply from a dedicated voltage regulator if you have the option.

Use a star point for the earthing. Typically this will ideally be the negative pin of a 100 – 1000uF smoothing capacitor, with all the grounds taken back to this single point.

Use thick tracks to high power devices and areas to reduce airborne interference.

If you are able to have +V and 0V power tracks to high power devices run above and below each other on top and bottom layers this will help cancel out the field they generate.

When using long PCB tracks for a connection to an antenna ideally use a decent thickness and route along the edge of the PCB if possible and away from noisy devices / circuitry. Placing ground plane around and below tracking can be useful to protect the signals.

Remember, the enemy of any RFID electronics design is noise! Is there anything else you can think of to reduce it?

Solving Interference Problems

If you have +V and 0V cables running in proximity to the RFID reader twisting them together will help cancel out the field they generate. Also make them as short as possible and if possible angle them away from your reader.

Experiment by turning off the RFID reader and looking at the coil voltage with a sensitive oscilloscope. Rotate the coil for minimum interference. If possible move the coil to a minimum interference position. Even if it is impossible to eventually change the angle or possition of the coil, this experiment may identify 'Hot Spots' that will help you zero in on the cause of a problem.

Many RFID readers have interference rejection but the interference may be so great that it is saturating the output of the reader detector. It may be that a low Q coil will give a lower output and the reader will not saturate so it may actually read better. A 1k resistor across the coil is extreme but may still be a good starting point. You can plot the range against several values of resistance from 1k to 10k.

The effects of metalwork are often not predictable. If the RFID reader coil is surrounded by metal in such a way that the steel effectively becomes a shorted turn, then this will appear as negative inductance and you could try to compensate by adding 50pF in parallel with the coil. If it helps try other values. A simple way to see if the metal is having an effect is to isolate the reader and measure its supply both in situ and out of situ. If there is a large difference then you may have a problem.

Multiple Aerials

If your project requires the use of multiple RFID antennas you may consider using some form of aerial switching circuitry. Whilst entirely possible this solution is typically most suited to designs that are making use of expensive off the shelf OEM RFID reader modules. The HTRC110 and its associated components are so cheap that it is usually much more cost effective to provide a HTRC110 for each separate antenna and simply switch the 3 serial connections to your microcontroller using an IC such as the 74HC4053. This also removes the issues of non perfect resistance values for typically used opto isolated MosFet switches (such as the AQY210EHA) which will affect antenna performance.

Writing Tags

Although this driver does not support it the HTRC110 does provide the posibility of writing data to some specific alternative RFID tag types. The datasheet and application note sections covering this can be safely ignored.

HTRC110

The HTRC110 IC was selected as it is a low cost but advanced RFID reader IC which incorporates many features to remove tuning and calibration requirements often found with other RFID reader IC's. It is widely used in a large number of commerical RFID readers. The HTRC110 may also be used with other RFID protocols and this driver provides an excellent starting point for such development if this is your aim.

OSCILLATOR

The HTRC110 and your microcontroller can be run off a single oscillator if desired. See the HTRC110 datasheet for details.

PURCHASING PARTS

HTRC110

HTRC110 may be purchased from DigiKey, part number: 568-2206-5-ND

Aerials

The following are a selection of close range 680uH Inductor examples which will work as aerials with the HTRC110. Please note that this list is meant to be an example only, and pretty much any general 680uH (or thereabouts) inductor will work. The larger the coil and the more wire turns its has the greater the range your reader will have.

Epcos B82111E0000C028 (Farnell part number 164-4286)

Murata 1468420C (Farnell part number 107-7026)

Murata 22R684C (Farnell part number 107-7035)

Wuerth Elektronik 74456268 (Farnell part number 163-6141)

Wuerth Elektronik 74457268 (Farnell part number 163-6177)

How The Driver Works

How the HTRC110 works

Overview

The HTRC110 is a great RFID reader IC and it takes care of all aspects of the radio frequency transmittion and reception. It incorporates features to remove tuning and calibration requirements which can be a clasic pain of RFID reader deisgn. The communication interface to the microcontroller is similar to SPI. However a special start bit sequence is used and there is no chip select pin making it not directly compatible with SPI. The driver incudes an option to allow a spare microcontroller pin to be used as a dummy chip select pin, allowing you to use a logic analyser to capture the communications between the HTRC110 and your microcontroller during development if you wish – see the rfid.h file for details of enabling this.

Reading Tags

The HTRC will take care of all aspects of the radio frequency side of things for you, but its output when reading a tag is the raw modulated data output from the tag. This 64bit continuous looping data stream needs to be decoded by your microcontoller to determin if the data is from a valid tag or if it is just RF noise (if no tag is present the datastream will contain junk data). Once the HTRC110 has been put in read tag mode in will pass the data stream until the microcontroller stops it. It is during this phase that the HTRC uses the most power.

How The Driver Functions

On powerup the driver will initialise the HTRC110 and then place it in a low power standby state.

When your applicaiton calls the rfid_read_tag() function the driver then enables the HTRC110, carries out several configuration functions and RF value setting operations before placing the HTRC110 in read mode. In this mode the Dout pin of the HTRC110 simply passes the received modulated data from the RFID tag, or if no tag is present it will contain trash data. The 64bits of data the RFID tag will be continuously transmitting in a cycle containing 9 header bits, 40 data bits and 14 parity bits. The header bits allow the start of the data sequence to be identified and the 14 parity check bits provide sufficient error detection to ensure that only a valid RFID tag will be read. The EM4102 datasheet provides a lot of detail about the data bits and the possible modulation methods.

Once in read mode the HTRC110 outputs a 125kHz sine wave. The tag (if present) will interfear with the RF sine wave signal in order to pass its data stream back which the HTRC110 detects for us. When reading is active the Dout pin of the HTRC110 contains the raw data output of the tag after being filtered into a digital signal by the HTRC110.

Although the tag data length is 64 bits the driver needs to receive an initial 22 bits to determin the bit rate and then between 64 to 127 bits need to be received before we have a complete sequence.

Therefore a read can take 149 bits x 512uS maximum bit rate = 76mS.

Tags will often be set to use the slowest 512uS bit rate as slow transitions on the air interface are easier to detect, which means you get additional range and more reliable communications.

The EM4102 protocol permits tags to respond with Manchester, BiPhase or PSK (Phase Shift Keying) modulation. Although any of the 3 methods can be used tags basically always use Manchester encoding as there are many RFID readers out there that only support manchaester encoding. This driver supports Manchester and BiPhase encoding, as BiPhase is very similar to Manchester and there is minimal code footprint required to also support it. Although supporting PSK is not an particuarly complex task, it does requrie specific microcontroller hardware to be done efficiently and would cause unessary complexity so as PSK is not used in practice this driver does not support it.

For Manchester or BiPhase modulation the tag will be responding with 1 bit every 64, 32 or 16 periods of the carrier frequency, and will change state once or twice during this (the tag is allowed to use any of these three as its data rate). Using a clever detection method the driver only needs to detect each rising edge to determine the preceeding bit states, reducing the interrupt burden on the main application. Also only detecting each rising edge removes issues of dealing with non perfect high and low widths within each cycle – being radio they are often not perfectly 50/50, but the overall bit frequency will be relatively accurate.

So for either of these modulations the bit rate will be approximately 128uS, 256uS or 512uS. If the tag was using PSK modulation it would be changing state every carrier frequency period (every 8uS)

Once the bit rate has been determined by the driver it calculates the 125% and 175% bit times which it will then use to decode the data stream.

The driver now decodes the receied data bits on the fly as they are received. This is a much more efficient and minimal memory requriement approach rather than storing a large chunk of data and decoding it later.

To find the header the driver looks for the bit sequence 0111111111 which is the stop bit followed by header. This sequence cannot occur anywhere else in the tags 64 bit stream.

From the header bitstream the driver is able to determin if Manchester of BiPhase modulation is being used by the RFID tag. The driver then decodes the remaining bits of the 64 bit sequence, checking for a parity error as each parity bit is received.

If all of the parity checks pass and the final stop bit is received the RFID read is a sucess and a valid tag has been detected. The driver stores the 10 character ASCII hex tag ID ready for the main application to read and switches the HTRC110 back into low power mode. When a tag is present and in range there is typcially no need for a retry – the read sequence will typically work first time.

However the driver will retry for up to 150mS in case it helps where the signal is week. As this driver checks for parity errors as nibbles are received (rather than at the end) failures can occur early on in the bit sequence, so doing this will often mean the driver is ready to try again at the start of the very next sequence from the tag.

If no valid tag bitstream is detected the driver switches the HTRC110 back into low power mode and declares the read a failure.

Designed by:

IBEX UK Limited
32A Station Road West
Oxted
Surrey
RH8 9EU
England
Tel: +44 (0)1883 716 726
E-mail: info@ibexuk.com
Web: www.ibexuk.com