|=---=|
|=----------------------=[Hijacking RDS-TMC Traffic]=------------------=|
|=----------------------=[Information signal   ]=------------------=|
|=---=|
|=---=|
|=------------------=[By Andrea "lcars" Barisani   ]=--------------=|
|=------------------=[<lcars_at_inversepath_dot_com>]=--------------=|
|=------------------=[   ]=--------------=|
|=------------------=[Daniele "danbia" Bianco   ]=--------------=|
|=------------------=[<danbia_at_inversepath_dot_com>]=--------------=|
|=---=|

--[Contents

--[1. Introduction

Modern Satellite Navigation systems use a recently developed standard called RDS-TMC (Radio Data System - Traffic Message Channel) for receiving traffic information over FM broadcast. The protocol allows communication of traffic events such as accidents and queues. If information affects the current route plotted by the user the information is used for calculating and suggesting detours and alternate routes. We are going to show how to receive and decode RDS-TMC packets using cheap homemade hardware, the goal is understanding the protocol so that eventually we may show how trivial it is to inject false information.

We also include the first release of our Simple RDS Decoder (srdsd is the lazy name) which as far as we know is the first open source tool available which tries to fully decode RDS-TMC messages. It's not restricted to RDS-TMC since it also performs basic decoding of RDS messages.

The second part of the article will cover transmission of RDS-TMC messages, satellite navigator hacking via TMC and its impact for social engineering attacks.

--[2. Motivation

RDS has primarily been used for displaying broadcasting station names on FM radios and give alternate frequencies, there has been little value other than pure research and fun in hijacking it to display custom messages.

However, with the recent introduction of RDS-TMC throughout Europe we are seeing valuable data being transmitted over FM that actively affects SatNav operations and eventually the driver's route choice. This can have very important social engineering consequences. Additionally, RDS-TMC messages can be an attack vector against SatNav parsing capabilities.

Considering the increasing importance of these system's role in car operation (which are no longer strictly limited to route plotting anymore) and their human interaction they represent an interesting target combined with the "cleartext" and un-authenticated nature of RDS/RDS-TMC messages.

We'll explore the security aspects in Part II.

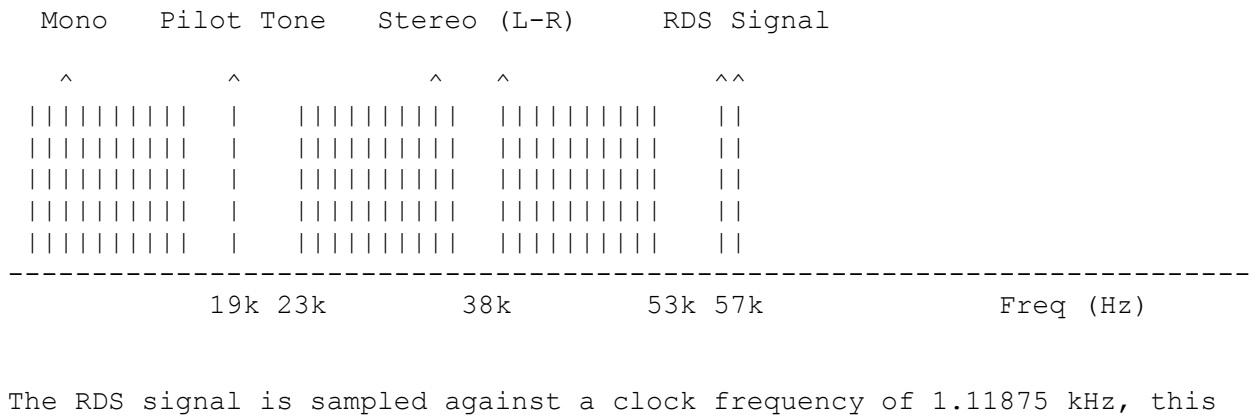
--[3. RDS

The Radio Data System standard is widely adopted on pretty much every modern FM radio, 99.9% of all car FM radio models feature RDS nowadays. The standard is used for transmitting data over FM broadcasts and RDS-TMC is a subset of the type of messages it can handle. The RDS standard is described in the European Standard 50067.

The most recognizable data transmitted over RDS is the station name which is often shown on your radio display, other information include alternate frequencies for the station (that can be tried when the signal is lost), descriptive information about the program type, traffic announcements (most radio can be set up to interrupt CD and/or tape playing and switch to radio when a traffic announcement is detected), time and date and many more including TMC messages.

In a FM transmission the RDS signal is transmitted on a 57k subcarrier in order to separate the data channel from the Mono and/or Stereo audio.

FM Spectrum:

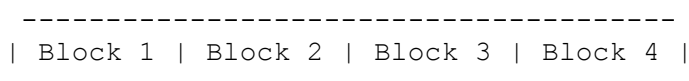


The RDS signal is sampled against a clock frequency of 1.11875 kHz, this means that the data rate is 1187.5 bit/s (with a maximum deviation of +/- 0.125 bit/s).

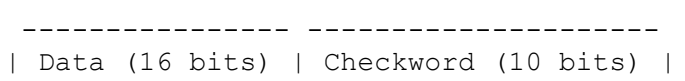
The wave amplitude is decoded in a binary representation so the actual data stream will be friendly '1' and '0'.

The RDS smallest "packet" is called a Block, 4 Blocks represent a Group. Each
Block has 26 bits of information making a Group 104 bits large.

Group structure (104 bits):



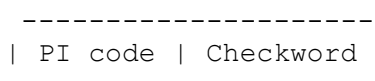
Block structure (26 bits):



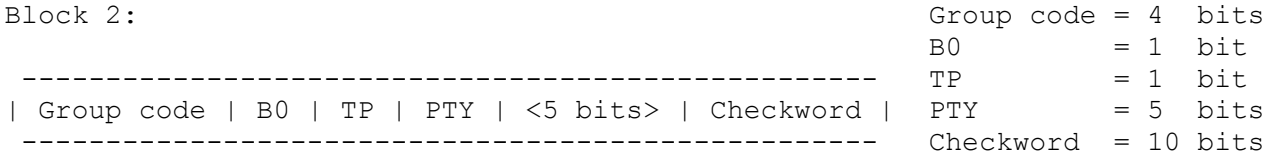
The Checkword is a checksum included in every Block computed for error protection, the very nature of analog radio transmission introduces many errors in data streams. The algorithm used is fully specified in the standard and it doesn't concern us for the moment.

Here's a representation of the most basic RDS Group:

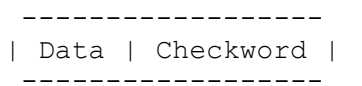
Block 1:



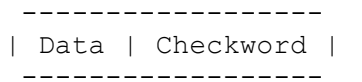
Block 2:



Block 3:



Block 4:



The PI code is the Programme Identification code, it identifies the radio station that's transmitting the message. Every broadcaster has a unique assigned code.

The Group code identifies the type of message being transmitted as RDS can be used for transmitting several different message formats. Type 0A (00000) and 0B (00001) for instance are used for tuning information. RDS-TMC messages are transmitted in 8A (10000) groups. Depending on the Group type the remaining 5 bits of Block 2 and the Data part of Block 3 and Block 4 are used according to the relevant Group specification.

The 'B0' bit is the version code, '0' stands for RDS version A, '1' stands for RDS version B.

The TP bit stands for Traffic Programme and identifies if the station is capable of sending traffic announcements (in combination with the TA code

present in 0A, 0B, 14B, 15B type messages), it has nothing to do with RDS-TMC and it refers to audio traffic announcements only.

The PTY code is used for describing the Programme Type, for instance code 1 (converted in decimal from its binary representation) is 'News' while code 4 is 'Sport'.

--[4. RDS-TMC

Traffic Message Channel packets carry information about traffic events, their location and the duration of the event. A number of lookup tables are being used to correlate event codes to their description and location codes to the GPS coordinates, those tables are expected to be present in our SatNav memory. The RDS-TMC standard is described in International Standard (ISO) 14819-1.

All the most recent SatNav systems supports RDS-TMC to some degree, some systems requires purchase of an external antenna in order to correctly receive
the signal, modern ones integrated in the car cockpit uses the existing FM antenna used by the radio system. The interface of the SatNav allows display of the list of received messages and prompts detours upon events that affect the current route.

TMC packets are transmitted as type 8A (10000) Groups and they can be divided in two categories: Single Group messages and Multi Group messages. Single Group messages have bit number 13 of Block 2 set to '1', Multi Group messages have bit number 13 of Block 2 set to '0'.

Here's a Single Group RDS-TMC message:

Block 1:

```
 --------------------              PI code   = 16 bits
| PI code | Checkword |            Checkword = 10 bits
 --------------------
```

Block 2:

```
                                                        Group code = 4  bits
                                                        B0         = 1  bit
 ----------------------------------------------------   TP         = 1  bit
| Group code | B0 | TP | PTY | T | F | DP | Checkword | PTY        = 5  bits
 ----------------------------------------------------   Checkword  = 10 bits
```

```
 T = 1 bit    DP = 3 bits
 F = 1 bit
```

Block 3:

```
                                                        D          = 1 bit
                                                        PN         = 1 bit
 ------------------------------------                   Extent     = 3 bits
| D | PN | Extent | Event | Checkword |                 Event      = 11 bits
 ------------------------------------                   Checkword  = 10 bits
```

Block 4:

```
 ----------------------             Location  = 16 bits
| Location | Checkword |           Checkword = 10 bits
 ----------------------
```

We can see the usual data which we already discussed for RDS as well as new information (the <5 bits> are now described).

We already mentioned the 'F' bit, it's bit number 13 of Block 2 and it identifies the message as a Single Group (F = 1) or Multi Group (F = 0).

The 'T', 'F' and 'D' bits are used in Multi Group messages for identifying if
this is the first group (TFD = 001) or a subsequent group (TFD = 000) in the stream.

The 'DP' bit stands for duration and persistence, it contains information about the timeframe of the traffic event so that the client can automatically flush old ones.

The 'D' bit tells the SatNav if diversion advice needs to be prompted or not.

The 'PN' bit (Positive/Negative) indicates the direction of queue events, it's opposite to the road direction since it represent the direction of the growth of a queue (or any directional event).

The 'Extent' data shows the extension of the current event, it is measured in terms of nearby Location Table entries.

The 'Event' part contains the 11 bit Event code, which is looked up on the local Event Code table stored on the SatNav memory. The 'Location' part contains the 16 bit Location code which is looked up against the Location Table database, also stored on your SatNav memory, some countries allow a free download of the Location Table database (like Italy[1]).

Multi Group messages are a sequence of two or more 8A groups and can contain additional information such as speed limit advices and supplementary information.

--[5. Sniffing circuitry

Sniffing RDS traffic basically requires three components:

1. FM radio with MPX output
2. RDS signal demodulator
3. RDS protocol decoder

The first element is a FM radio receiver capable of giving us a signal that has not already been demodulated in its different components since we need access to the RDS subcarrier (and an audio only output would do no good). This kind of "raw" signal is called MPX (Multiplex). The easiest way to get such signal is to buy a standard PCI Video card that carries a tuner which has a MPX pin that we can hook to.

One of these tuners is Philips FM1216[2] (available in different "flavours", they all do the trick) which provides pin 25 for this purpose. It's relatively easy to identify a PCI Video card that uses this tuner, we used the WinFast DV2000. An extensive database[3] is available.

Once we get the MPX signal it can then be connect to a RDS signal demodulator which will perform the de-modulation and gives us parsable data. Our choice is ST Microelectronics TDA7330B[4], a commercially available chip used in most radio capable of RDS de-modulation. Another

possibility could be the Philips SAA6579[5], it offers the same functionality of the TDA7330, pinning might differ.

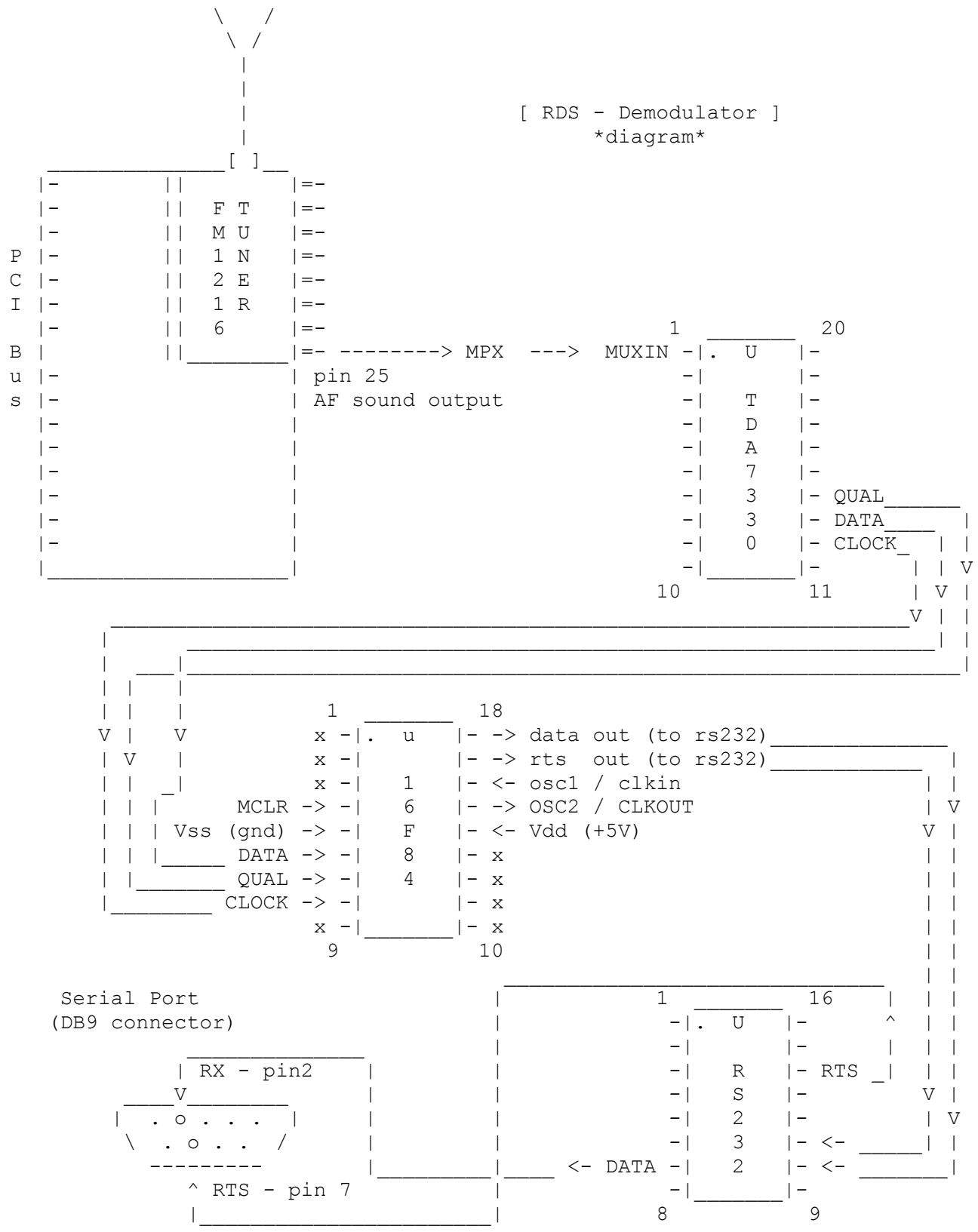
Finally we use custom PIC (Peripheral Interface Controller) for preparing and sending the information generated by the TDA7330 to something that we can understand and use, like a standard serial port.

The PIC brings DATA, QUAL and CLOCK from demodulator and "creates" a stream good enough to be sent to the serial port. Our PIC uses only two pins of the serial port (RX - RTS), it prints out ascii '0' and '1' clocked at 19200 baud rate with one start bit and two stop bits, no parity bit is used.

As you can see the PIC makes our life easier, in order to see the raw stream we only have to connect the circuit and attach a terminal to the serial port, no particular driver is needed. The PIC we use is a PIC 16F84, this microcontroller is cheap and easy to work with (its assembly has only 35 instructions), furthermore a programmer for setting up the chip can be easily bought or assembled. If you want to build your own programmer a good choice would be uJDM[6], it's one of the simplest PIC programmers available (it is a variation of the famous JDM programmer).

At last we need to convert signals from the PIC to RS232 compatible signal levels. This is needed because the PIC and other integrated circuits works under TTL (Transistor to Transistor Logic - 0V/+5V), whereas serial port signal levels are -12V/+12V. The easiest approach for converting the signal is using a Maxim RS-232[7]. It is a specialized driver and receiver integrated circuit used to convert between TTL logic levels and RS-232 compatible signal levels.

Here's the diagram of the setup:



Here's the commented assembler code for our PIC:

```
;
; Copyright 2007 Andrea Barisani <lcars@inversepath.com>
;                Daniele Bianco <danbia@inversepath.com>
;
; Permission to use, copy, modify, and distribute this software for any
; purpose with or without fee is hereby granted, provided that the above
; copyright notice and this permission notice appear in all copies.
;
; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
; WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
; MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
; ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
; WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
; ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
; OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
;
; Pin diagram:
;
;                    1  _______  18
;                  x -|.  U    |- -> DATA out (to RS232)
;                  x -|        |- -> RTS  out (to RS232)
;                  x -|    1   |- <- OSC1 / CLKIN
;              MCLR -> -|    6   |- -> OSC2 / CLKOUT
;        Vss (gnd) -> -|    F   |- <- Vdd (+5V)
;             DATA -> -|    8   |- x
;             QUAL -> -|    4   |- x
;            CLOCK -> -|        |- x
;                  x -|_______ |- x
;                    9          10
;
; Connection description:
;
; pin 4 : MCLR          (it must be connected to Vdd through a resistor
;                        to prevent PIC reset - 10K is a good resistor)
; pin 5 : Vss           (directly connected to gnd)
;
; pin 6 : DATA  input   (directly connected to RDS demodulator DATA  out)
; pin 7 : QUAL  input   (directly connected to RDS demodulator QUAL  out)
; pin 8 : CLOCK input   (directly connected to RDS demodulator CLOCK out)
;
; pin 14: Vdd           (directly connected to +5V)
; pin 15: OSC2 / CLKOUT (connected to an 2.4576 MHz oscillator crystal* )
; pin 16: OSC1 / CLKIN  (connected to an 2.4576 MHz oscillator crystal* )
;
; pin 17: RTS  output   (RS232 - ''RTS'' pin 7 on DB9 connector** )
; pin 18: DATA output   (RS232 - ''RX''  pin 2 on DB9 connector** )
;
; pin 1,2,3,9,10,11,12,13: unused
;
; *)
; We can connect the oscillator crystal to the PIC using this simple
; circuit:
;
;                C1 (15-33 pF)
;              ____||____ ______ OSC1 / CLKIN
;             |    ||    |
;             |          ___
;      gnd ---|           =  XTAL (2.4576 MHz)
```

```
;              |         ---
;              |____||____|______
;                   ||             OSC2 / CLKOUT
;                 C2 (15-33 pF)
; **)
; We have to convert signals TTL <-> RS232 before we send/receive them
; to/from the serial port.
; Serial terminal configuration:
; 8-N-2 (8 data bits - No parity - 2 stop bits)
;

; HARDWARE CONF -----------------------
    PROCESSOR    16f84
    RADIX        DEC
    INCLUDE      "p16f84.inc"

    ERRORLEVEL   -302                ; suppress warnings for bank1

    __CONFIG 1111111110001b          ; Code Protection  disabled
                                     ; Power Up Timer    enabled
                                     ; WatchDog Timer   disabled
                                     ; Oscillator type        XT
; -----------------------------------

; DEFINE ------------------------------
#define    Bank0     bcf  STATUS, RP0  ; activates bank 0
#define    Bank1     bsf  STATUS, RP0  ; activates bank 1

#define    Send_0    bcf     PORTA, 1  ; send 0 to RS232 RX
#define    Send_1    bsf     PORTA, 1  ; send 1 to RS232 RX
#define    Skip_if_C btfss  STATUS, C  ; skip if C FLAG is set

#define    RTS               PORTA, 0  ; RTS   pin RA0
#define    RX                PORTA, 1  ; RX    pin RA1
#define    DATA              PORTB, 0  ; DATA  pin RB0
#define    QUAL              PORTB, 1  ; QUAL  pin RB1
#define    CLOCK             PORTB, 2  ; CLOCK pin RB2

RS232_data     equ               0x0C  ; char to transmit to RS232
BIT_counter    equ               0x0D  ; n. of bits to transmit to RS232
RAW_data       equ               0x0E  ; RAW data (from RDS demodulator)
dummy_counter  equ               0x0F  ; dummy counter... used for delays
; -----------------------------------

; BEGIN PROGRAM CODE ------------------

    ORG    000h

InitPort

    Bank1                            ; select bank 1

    movlw  00000000b                 ; RA0-RA4 output
    movwf  TRISA                     ;

    movlw  00000111b                 ; RB0-RB2 input / RB3-RB7 output
    movwf  TRISB                     ;

    Bank0                            ; select bank 0

    movlw  00000010b                 ; set voltage at -12V to RS232 ''RX''
```

```
    movwf  PORTA                    ;

Main

    btfsc  CLOCK                    ; wait for clock edge (high -> low)
    goto   Main                     ;

    movfw  PORTB                    ;
    andlw  00000011b                ; reads levels on PORTB and send
    movwf  RAW_data                 ; data to RS232
    call   RS232_Tx                 ;

    btfss  CLOCK                    ; wait for clock edge (low -> high)
    goto   $-1                      ;

    goto   Main

RS232_Tx                            ; RS232 (19200 baud rate) 8-N-2
                                    ; 1 start+8 data+2 stop - No parity
    btfsc  RAW_data,1
    goto   Good_qual
    goto   Bad_qual

Good_qual                           ;
    movlw  00000001b                ;
    andwf  RAW_data,w               ; good quality signal
    iorlw  '0'                      ; sends '0' or '1' to RS232
    movwf  RS232_data               ;
    goto   Char_Tx

Bad_qual                            ;
    movlw  00000001b                ;
    andwf  RAW_data,w               ; bad  quality signal
    iorlw  '*'                      ; sends '*' or '+' to RS232
    movwf  RS232_data               ;

Char_Tx

    movlw  9                        ; (8 bits to transmit)
    movwf  BIT_counter              ; BIT_counter = n. bits + 1

    call   StartBit                 ; sends start bit

Send_loop

    decfsz BIT_counter, f           ; sends all data bits contained in
    goto   Send_data_bit            ; RS232_data

    call   StopBit                  ; sends 2 stop bit and returns to
Main

    Send_1
    goto   Delay16

StartBit

    Send_0
    nop
    nop
    goto   Delay16
```

```
StopBit

    nop
    nop
    nop
    nop
    nop

    Send_1
    call   Delay8
    goto   Delay16

Send_0_
    Send_0
    goto   Delay16

Send_1_
    nop
    Send_1
    goto   Delay16

Send_data_bit
    rrf    RS232_data, f              ; result of rotation is saved in
    Skip_if_C                         ; C FLAG, so skip if FLAG is set
    goto   Send_zero
    call   Send_1_
    goto   Send_loop

Send_zero
    call   Send_0_
    goto   Send_loop
;
; 4 / clock = ''normal'' instruction period (1 machine cycle )
; 8 / clock = ''branch'' instruction period (2 machine cycles)
;
;     clock            normal instr.           branch instr.
;   2.4576 MHz           1.6276 us               3.2552 us
;
Delay16

    movlw  2                          ; dummy cycle,
    movwf  dummy_counter              ; used only to get correct delay
                                      ; for timing.
    decfsz dummy_counter,f            ;
    goto  $-1                         ; Total delay: 8 machine cycles
    nop                               ; ( 1 + 1 + 1 + 2 + 2 + 1 = 8 )

Delay8

    movlw  2                          ; dummy cycle,
    movwf  dummy_counter              ; used only to get correct delay
                                      ; for timing.
    decfsz dummy_counter,f            ;
    goto   $-1                        ; Total delay: 7 machine cycles
                                      ; ( 1 + 1 + 1 + 2 + 2 = 7 )
Delay1

    nop

    RETURN                            ; unique return point
```

```
END

; END PROGRAM CODE --------------------

</code>
```

Using the circuit we assembled we can "sniff" RDS traffic directly on the serial port using screen, minicom or whatever terminal app you like.
You should configure your terminal before attaching it to the serial port, the settings are 19200 baud rate, 8 data bits, 2 stop bits, no parity.

```
# stty -F /dev/ttyS0 19200 cs8 cstopb -parenb
speed 19200 baud; rows 0; columns 0; line = 0; intr = ^C; quit = ^\;
erase = ^?; kill = ^H; eof = ^D; eol = <undef>; eol2 = <undef>;
swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 100; time = 2; -parenb -parodd
cs8 -hupcl cstopb cread clocal crtscts -ignbrk brkint ignpar -parmrk -inpck
-istrip -inlcr -igncr -icrnl -ixon -ixoff -iuclc -ixany -imaxbel -iutf8
-opost -olcuc -ocrnl -onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0 -isig -icanon iexten -echo echoe echok -echonl -noflsh -xcase
-tostop -echoprt echoctl echoke

# screen /dev/ttyS0 19200
10101001000011000000000101000*000101001+11101111011111111110000001011011100
10101001++000001100101100*11010010100100001100000011101000010010100111111
00111011000100110001000000+000000000 ... <and so on>
```

As you can see we get '0' and '1' as well as '*' and '+', this is because the circuit estimates the quality of the signal. '*' and '+' are bad quality '0' and '1' data. We ignore bad data and only accept good quality. Bad quality data should be ignored, and if you see a relevant amount of '*' and '+' in your stream verify the tuner settings.

In order to identify the beginning of an RDS message and find the right offset we "lock" against the PI code, which is present at the beginning of every RDS group. PI codes for every FM radio station are publicly available on the Internet, if you know the frequency you are listening to then you can figure out the PI code and look for it. If you have no clue about what the PI code might be a way for finding it out is seeking the most recurring 16 bit string, which is likely to be the PI code.

Here's a single raw RDS Group with PI 5401 (hexadecimal conversion of 101010000000001):

```
0101010000000001111101100100000100001010001100101100000000100001010000001100
1001010010010000010001101110
```

Let's separate the different sections:

```
0101010000000001 1111011001 0000  01 0  0001 01000    1100101100
0000001000010100 0000110010 0101001001000001 0001101110
PI code          Checkword  Group B0 TP PTY  <5 bits> Checkword  Data
Checkword  Data             Checkword
```

So we can isolate and identify RDS messages, now you can either parse them visually by reading the specs (not a very scalable way we might say) or use a tool like our Simple RDS Decoder.

--[6. Simple RDS Decoder 0.1

The tool parses basic RDS messages and 0A Group (more Group decoding will be implemented in future versions) and performs full decoding of Single group RDS-TMC messages (Multi Group support is also planned for future releases).

Here's the basic usage:

```
# ./srdsd -h

Simple RDS-TMC Decoder 0.1     || http://dev.inversepath.com/rds
Copyright 2007 Andrea Barisani || <andrea@inversepath.com>
Usage: ./srdsd.pl [-h|-H|-P|-t] [-d <location db path>] [-p <PI number>]
<input file>
   -t display only tmc packets
   -H HTML output (outputs to /tmp/rds-*.html)
   -p PI number
   -P PI search
   -d location db path
   -h this help

Note: -d option expects a DAT Location Table code according to TMCF-LT-EF-
MFF-v06
      standard (2005/05/11)
```

As we mentioned the first step is finding the PI for your RDS stream, if you don't know it already you can use '-P' option:

```
# ./srdsd -P rds_dump.raw | tail

0010000110000000: 4140 (2180)
1000011000000001: 4146 (8601)
0001100000000101: 4158 (1805)
1001000011000000: 4160 (90c0)
0000110000000010: 4163 (0c02)
0110000000010100: 4163 (6014)
0011000000001010: 4164 (300a)
0100100001100000: 4167 (4860)
1010010000110000: 4172 (a430)
0101001000011000: 4185 (5218)
```

Here 5218 looks like a reasonable candidate being the most recurrent string. Let's try it:

```
# ./srdsd -p 5218 -d ~/loc_db/ rds_dump.raw

Reading TMC Location Table at ~/loc_db/:
       parsing NAMES: 13135 entries
       parsing ROADS: 1011 entries
       parsing SEGMENTS: 15 entries
       parsing POINTS: 12501 entries
done.

Got RDS message (frame 1)
      Programme Identification: 0101001000011000 (5218)
      Group type code/version: 0000/0 (0A  - Tuning)
      Traffic Program: 1
      Programme Type: 01001 (9  - Varied Speech)
      Block 2: 01110
```

```
      Block 3: 1111100000010110
      Block 4: 0011000000110010
      Decoded 0A group:
            Traffic Announcement: 0
            Music Speech switch: 0
            Decoder Identification control: 110 (Artificial Head / PS char
5,6)
            Alternative Frequencies: 11111000, 00010110 (112.3, 89.7)
            Programme Service name: 0011000000110010 (02)
            Collected PSN: 02

...

Got RDS message (frame 76)
      Programme Identification: 0101001000011000 (5218)
      Group type code/version: 1000/0 (8A  - TMC)
      Traffic Program: 1
      Programme Type: 01001 (9  - Varied Speech)
      Block 2: 01000
      Block 3: 0101100001110011
      Block 4: 0000110000001100
      Decoded 8A group:
            Bit X4: 0 (User message)
            Bit X3: 1 (Single-group message)
            Duration and Persistence: 000 (no explicit duration given)
            Diversion advice: 0
            Direction: 1 (-)
            Extent: 011 (3)
            Event: 00001110011 (115 - slow traffic (with average speeds Q))
            Location: 0000110000001100 (3084)
            Decoded Location:
                  Location code type: POINT
                  Name ID: 11013 (Sv. Grande Raccordo Anulare)
                  Road code: 266 (Roma-Ss16)
                  GPS: 41.98449 N 12.49321 E
                  Link:
http://maps.google.com/maps?ll=41.98449,12.49321&spn=0.3,0.3&q=41.98449,12.4
9321


...and so on.
```

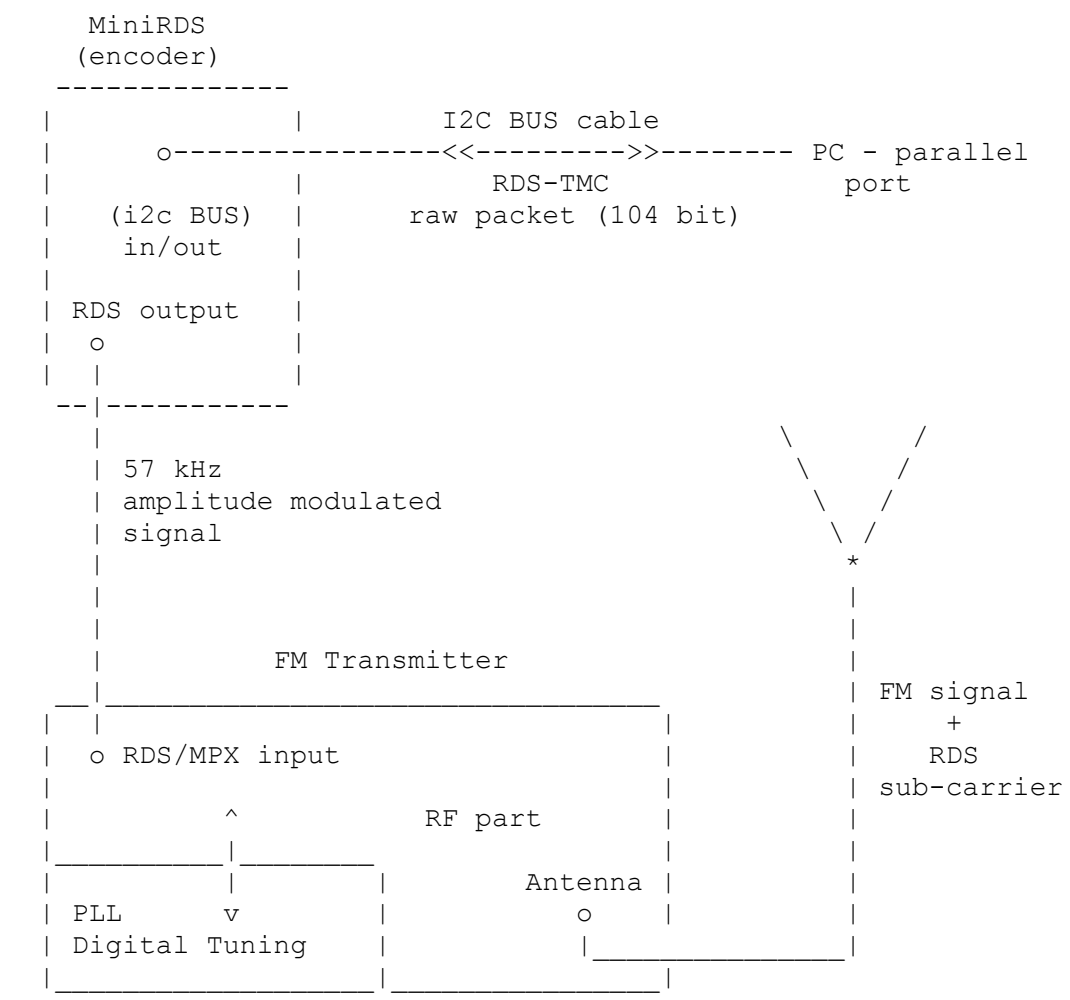
The 'Collected PSN' variable holds all the character of Programme Service name seen so far, this way we can track (just like RDS FM Radio do) the name of the station:

```
# ./srdsd -p 5201 rds_dump.raw | grep "Collected PSN" | head

            Collected PSN: DI
            Collected PSN: DIO1
            Collected PSN: DIO1
            Collected PSN: RADIO1
            Collected PSN: RADIO1
```

Check out '-H' switch for html'ized output in /tmp (which can be useful for directly following the Google Map links). We also have a version that plots all the traffic on Google Map using their API, if you are interested in it just email us.

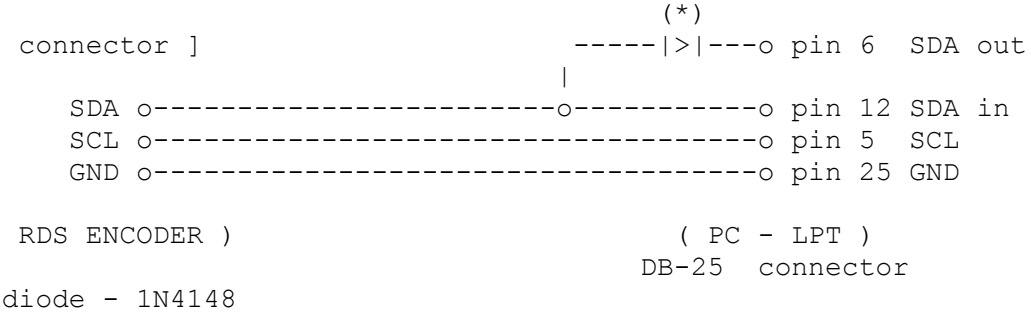
--[7. Injection circuitry



The hardware injection setup is composed by a PC, a RDS encoder and a FM transmitter.

We performed the RDS encoding using a single chip encoder[9] available from Piratske Radio[10], the core of this encoder is a programmed microcontroller type 18F12[11]. The chip holds a RAM memory and an EEPROM memory for data storage during power-off, both memory areas are accessible using the serial I2C protocol since the PIC fully implements an I2C BUS specifications. Commands and control byte sequences are described in the product data-sheet.

Here we show the schema for the cable needed to connect I2C BUS in/out pins on the encoder to the PC parallel port, the pinning here is consistent with the our driver code[12].



(*) switching diode - 1N4148

Our *CRUDE* (Code Rushed and Ugly due to unexpected DEadline) driver code allows easy write and read access of the encoder memory and command sending to the device. You can use the information gathered from our decoder application for crafting whatever RDS-TMC packet you might need.

The output of the encoder module is an amplitude modulated signal centered on 57 kHz, this signal is ready to be attached as MPX/RDS input to the FM transmitter.

Almost every FM transmitter is suitable for TMC injection, the only important requirement is frequency stability. An unstable transmitter doesn't allow proper broadcasting of the RDS data stream, this is because a sensible frequency spreading of the RDS sub-carrier will result in a great number of corrupted data blocks on the receiving demodulator.

Thus, if you plan to build your own transmitter we advice you to integrate in your device a PLL circuit to properly lock on the selected frequency.

The transmitter we built includes a digital tuner based on the SAA1057[13] Radio tuning PLL frequency synthesizer. Our transmitter also offers an audio part suitable to hook external MICs or other audio peripherals, that's very useful to testing purposes.

Moreover the RF part holds an MPX/RDS input suitable to attach directly the RDS encoder module signal, it's also possible to mix together an extra audio component with the RDS signal using a simple mixer circuit.

Detailed scheme for a sample FM transmitter is available from the Piratske Radio website. In the site you can also find useful information if you plan to build your own TX antenna. This is not the only FM transmitter you can use, there are many resource available on the Internet that can easily allow whatever FM transmitter you might need for any purpose.

--[I. References

[1] - Italian RDS-TMC Location Table Database
https://www2.ilportaledellautomobilista.it/info/infofree?idUser=1&idBody=14

[2] - Philips FM1216 DataSheet
http://pvr.sourceforge.net/FM1216.pdf

[3] - PVR Hardware Database
http://pvrhw.goldfish.org

[4] - SGS-Thompson Microelectronics TDA7330
http://www.datasheetcatalog.com/datasheets_pdf/T/D/A/7/TDA7330.shtml

[5] - Philips SAA6579
http://www.datasheetcatalog.com/datasheets_pdf/S/A/A/6/SAA6579.shtml

[6] - uJDM PIC Programmer
http://www.semis.demon.co.uk/uJDM/uJDMmain.htm

[7] - Maxim RS-232
http://www.maxim-ic.com/getds.cfm?qv_pk=1798&ln=en

[8] - Xcircuit
http://xcircuit.ece.jhu.edu

[9] - MiniRDS encoder chip (MRDS192)
      http://www.pira.cz/rds/mrds192.pdf

[10] - Piratske Radio website - RDS section
      http://www.pira.cz/rds/index.htm

[11] - Microchip PIC 18F1220
       http://ww1.microchip.com/downloads/en/DeviceDoc/39605F.pdf

[12] - MiniRDS driver code
       http://dev.inversepath.com/rds/i2c_minirds.tar.gz

[13] - Philips SAA1057 - Radio tuning PLL frequency synthesizer
       http://www.datasheetcatalog.net/de/datasheets_pdf/S/A/A/1/SAA1057.shtml

--[II. Links

- Project directory
  http://dev.inversepath.com/rds

|=[EOF]=---=|