ARDF Transmitter Controller

US version Firmware Description Manual

Version 8x.0

Table of Contents

Introduction

The microcontroller used in the minifox transmitter (described in a separate document, schematic in Appendix 4) has grown to its present form through many stages. Initially, the controller was implemented with an ATtiny13 which had limited pin-programming to control the ID that sent continuously. Several investigations followed to try to use the internal oscillator of various of the Atmel processors for more complex timing control, ending with the conclusion that an external crystal oscillator was required for adequate timing accuracy. The resulting implementation described here uses the ATtiny85 processor, programmed as an arduino platform. Those interested in more detail of the development processes can find it at the codppm link under www.islandnet.com/~jyoung/arduinopgm.htm

Although the controller was developed specifically for the minifox sprint and fox-or operations, it is also suitable for use in a standard high-power fox or beacon.

The peculiar name of the program to be described, codppmx8inv.ino, arose over the course of developing it. Approximately, it “means” code, pulse per minute, crystal, version 8, invertable outputs, and .ino means it's an arduino sketch. The version described here is **codppmx8US.ino** is derived from codppmx8inv.

The goals/requirements for the implementation, now realized in the software version codppmx8US.ino, are:

- low-power, low-voltage operation – two AA cells power supply.
- ARDF Sprint competition modes (12-second/48-second), Beacon (ARDF Fox-Or) mode, ARDF Standard (one-minute/four-minute) mode
- Accurate timing: keep Sprint modes' starting time on an even minute stable to one or two seconds over 6 to 8 hours operating time
- Simple operating mode selection among Sprint slow, Sprint fast, Beacon, Standard
- Minimize timing sequence interruption with sending required call sign ID. ID is sent at start, every 10 minutes thereafter for beacon and sprint modes. Standard mode 3 IDs at end of each ON time.
- Fox ID and call sign stored in EEPROM. Flash does not need to be changed.
- Indicator LED to show operation active for only a few seconds after power-on or reset.
- Sprint timing for foxes 1 – 5 automatically calculated so that every fox is started the same way—release reset on an even minute.

Fox ID and Call sign ID

The controller generates Morse code for on/off keying of the transmitter and LED via two output pins (6 and 5, respectively). The minifox transmitter in appendix 4 has the controller's pins configured so that the pin 6 Tx keying is active high, and the pin 5 output is open-drain, active low.

Conforming to IARU ARDF rules, the fox ID is one of MOE, MOI, MOS, MOH, MO5 (two dashes, three dashes, one to 5 dots, indicating fox 1 to 5 respectively) sent for the ON time of the mode. For the Sprint modes, the ON time is 12 seconds and the transmitter is silent for 48 seconds. For the Standard mode, the ON time is one minute, the OFF time is 4 minutes. The code speed is approximately 10 words per minute, except the Sprint Fast mode sends at 14 words per minute.

The call sign ID is inserted at the end of the first ON time after release of reset and it is sent at 20 WPM to minimize the disruption of the Sprint timing. The disruption will depend on the length of the call sign, but the worst case of the Slow Sprint's 12 second interval should still have at least one fox ID preceding the call sign. The call sign insertion is then repeated every 10 minutes thereafter. For the 1min/4min mode 3, the call sign is sent at the end of each 1 minute ON time.

All of the Morse code characters to be sent are stored one character per byte in the first few locations of the ATtiny85 EEPROM. The storage format is described below. These codes do not typically need to be changed after the initial decision is made about which fox number is to be sent, and who the 'control operator' is. However, if they do need to be changed, it is fairly easy to program only the eeprom with any of several Atmel-processor memory programming devices.

Code generator

The processor code generator loops through a sequence of ON/OFF intervals determined by each byte of code characters. First, the byte to be sent is scanned from the left to locate a 1 bit which is discarded. Then the remaining bits determine if the Tx ON time is a dot (bit = 0) or a dash (bit = 1). Thus, the code byte stored in EEPROM location 0 for a Morse M is 00000111. The next byte in location 1 for a Morse O is 00001111. The third character corresponds to the fox ID number by the number of dots. Fox 1, MOE will be encoded in EEPROM location 2 as 00000010. Similarly, MOI is 00000100, MOS is 00001000, MOH is 00010000, and MO5 is 00100000. Spaces are stored as a byte of zeros, the end of the call sign is indicated by a byte 0xFF which is the state of eeprom before it's been programmed. For the US version controller, a byte following the call-terminating 0xFF specifies the length of the call sign in dot times.

A suggestion for a 'by hand' design of the necessary code bytes would be to use a gridded notepad, write the characters to be encoded down the page, translate the characters to Morse dots (0) and dashes (1), right-justify the pattern, add the leftmost 1 bit, and finally translate the binary to hexadecimal (which the memory programming device likely wants). At the same time, the number of dot intervals spanned by the call sign can be noted by each letter. A dash with its following dot space is 4 dots long, a dot with its space is 2 dots long, and between each letter an inter-letter space of 3 dots is made by adding two more dot intervals to the letter. A scan of such a procedure is shown in Appendix 3.

Utility program – text2codeUS

Included in the codppmUS package is an arduino program text2codeUS.ino that might be helpful for preparing the codebytes for the EEPROM. This program accepts a text string on the serial input and outputs the corresponding string of codebytes, identifies the call sign and calculates its length in dots and milliseconds at 20 WPM. It also produces an intel-hex version of the codebytes which can then be copied and pasted into a text file which a programming device can use. This program is probably only useful if several processors need to be programmed with different call signs, and an arduino board and programming IDE is readily available. A screenshot of the text2codeUS output is shown below the 'manual' encoding example in Apendix 3.

EEPROM loading instructions

The detailed procedure to follow will depend on the machinery available: the programer device, the device control program, and possibly operating system setup. The instructions shown here will assume that the programmer is one of the usbtiny devices (for example, Sparkfun's “AVR Pocket Programmer”, or “Tiny Programmer”), and that the controlling program is the open-source, non-gnu.org, avrdude which is available for Windoze, OSX, and Linux. It's hoped that these instructions are general enough that you can translate them to use whatever device/program you have available.

Loading with hexfile

Assume the code bytes shown in the examples of appendix 3 is in a hexfile called eesUS.hex. The avrdude command to load this hexfile into the eeprom is:

```
avrdude -c usbtiny -p attiny85 -U eeprom:w:eesUS.hex
```

The contents of the hex file `eesUS.hex` look like this:

```
:0C000000070F08000B1830181B09FF4CFC
:00000001FF
```

The parameters on the `avrdude` command are:

`-c usbtiny` - specifies the programmer is a usbtiny type plugged into some usb port.

`-p attiny85` - specifies that the device being programmed is an ATtiny85

`-U eeprom:w:eesUS.hex` - specifies a memory access command, memory type is eeprom, the access is a write operation (w), and the data to be written is in file eesUS.hex. Avrdude discovers by reading the file that it is in intel hex format.

Direct entry into interactive mode

avrdude has an interactive mode of operation where you type individual programming commands at the command prompt, and avrdude carries them out and echos back the command just completed (or produces an error message if there is some problem). Using this mode is convenient for making small changes, for example, if only the fox ID is to be modified, then only one location in the eeprom needs to be modified. Also, using the interactive mode means that only avrdude and the hardware programmer is needed to program the minifox—no other software is required.

The following example terminal session shows the eeprom being changed to alter the fox ID alone from MOE to MOS (in this example, the text the user enters has been coloured blue and made bold in case of black printing of this document, all the rest of the text shown comes from the operating system or from avrdude):

```
joe@newmint:~/Desktop/minifox85$ avrdude -c usbtiny -p attiny85 -t

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e930b
avrdude> dump eeprom 0 0x30
>>> dump eeprom 0 0x30
0000  07 0f 02 00 0b 18 30 18  1b 09 ff 4c ff ff ff ff  |.... .0.. .L....|
0010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude> write eeprom 0x02 0x08
>>> write eeprom 0x02 0x08

avrdude> dump eeprom 0 0x30
>>> dump eeprom 0 0x30
0000  07 0f 08 00 0b 18 30 18  1b 09 ff 4c ff ff ff ff  |.... .0.. .L....|
0010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude>
```

Note in the example above that the only eeprom location that has changed between the two 'dump' commands is location 0x02.

The 'write' command can be extended to several bytes, so you could enter all of required code bytes into a new processor's eeprom in one command, just add the bytes to follow after the address (first byte after the write). As a shorter, multi-byte example, suppose the eeprom contents above are to be changed so that the call sign ID is changed from WB6BYU to WB6UBY—that is, only 3 bytes need to change. The following avrdude write and dump shows this operation (NOTE that this example is somewhat contrived—generally a changed call sign would be a different length, and require modifying the length byte as well):

```
avrdude> write eeprom 0x07 0x09 0x18 0x1b
>>> write eeprom 0x07 0x09 0x18 0x1b

avrdude> dump eeprom 0 0x30
>>> dump eeprom 0 0x30
0000  07 0f 02 00 0b 18 30 09  18 1b ff 4c ff ff ff ff  |.... .0 ...L....|
0010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude>
```

Interactive mode is exited by entering the command '**quit**'

Controller Program

The controller program (source code is in file codppmx8US.ino, see Appendix 1) is created as an arduino sketch. Recent versions of the arduino IDE (integrated development environment) (since 1.0.5) now allow using the very flexible, open-source, gcc toolchain for compiling the programs for a considerable variety of target processors, including the ATtiny85. Consequently, if one wishes to modify the program provided in the minifox transmitter, the free, multi-platform, arduino IDE can be used. The program codppmx8US has a couple of configuration options that might be of interest, even without changing anything else—the interval between call sign transmissions, the polaritiy of the LED output signal, the polarity of the transmitter output signal, and no overlap of one transmission on its following transmission can be changed by setting #define values located near the top of the program.

On the other hand, if it is only desired to use the existing program as-is, but to load it into an as-shipped ATtiny85 (say for replicating the minifox, or using the controller in a high-power standard fox), the codppmUS package includes the compiled program in the form of an intel-hex file so that only the programmer and its support program are needed to reproduce the controller. Instructions for doing this are in the following section. There are two versions of the program included in codppmUS package, codppmx8US.hex and codppmx8USinv.hex. The first corresponds to the original minifox transmitter where both outputs were open-drain. The second is for the current version as shown in the schematic in appendix 4.

A brief outline of the structure of the controller program:

- The top section of the source defines a variety of variables and functions. Interrupt service function for the crucial one-second timer using processor timer 1 and the timer setup function are defined here. The 4 lines beginning `#define TPOLARITY_POS` define how the pin polarity is controlled for the transmitter and the LED. To change pin polarity, add/remove comment // to leave the desired definitions active. The `#define CALL_INTERVAL 600` sets the interval, in seconds, between call sign insertions
- The setup( ) function runs once after power-on or reset. The fox number, the operating mode, and code speed is determined in setup( ).
- The loop( ) function runs repeatedly at the full speed of the processor. Timing of all operations is determined by testing timing variables against values generated by interrupt service routines. The main mode intervals test against the seconds elapsed measured by timer 1, faster times (code elements) test against the milliseconds elapsed measured by the arduino timer, millis( ).
- Several 'support' functions follow loop( ), collectively they generate the Morse code.
- Some comment seems warranted about the 'magic numbers' appearing in the code: good practice would suppose that meaningful symbols should be have been used more often. However, since many of the constants are constrained by the timing and will not be changed, using those as numbers directly rather than symbol names seems to unclutter the code (just my personal preference, of course). For instance, the sprint timing will always be 12 seconds ON, 48 seconds OFF.

Program loading instructions

The program is loaded into the processor flash memory using avrdude in nearly the same way as described for loading the eeprom from a hex file, simply specifying the flash memory type instead of eeprom and the program hex file:

```
avrdude -c usbtiny -p attiny85 -U flash:w:codppmx8US.hex
```

avrdude will respond with several messages describing the process of programming and then verifying the written program.

A few cautions to keep in mind: this command will work on an as-shipped processor which will use the processor's internal oscillator; the program loading will be preceded by a chip-erase which would also clean out the eeprom; when you program the fuses to use the 1.8432 MHz crystal, the crystal must be present for the communication with avrdude to work; the hfuse can be programmed to cause eeprom to be saved when a chip-erase is performed.

The following terminal session example shows how to load all memories of a new processor with a single avrdude command line. There are four memory commands, for the flash, the eeprom, and the two fuses. The command assumes that the files codppmx8US.hex, and eesUS.hex are in the current directory. The long command line is actually all typed on one line, although it's shown below wrapped onto a second line.

Note that the fuse programming commands have a qualifier, :m, appended which means that the data for the command is in the command, not in a file.

```
joe@newmint:~/Desktop/minifox85$ avrdude -c usbtiny -p attiny85 -U
flash:w:codppmx8US.hex -U eeprom:w:eesUS.hex -U hfuse:w:0xd7:m -U lfuse:w:0xeb:m

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e930b
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "codppmx8US.hex"
avrdude: input file codppmx8US.hex auto detected as Intel Hex
avrdude: writing flash (2858 bytes):

Writing | ################################################## | 100% 7.17s

avrdude: 2858 bytes of flash written
avrdude: verifying flash memory against codppmx8US.hex:
avrdude: load data flash data from input file codppmx8US.hex:
avrdude: input file codppmx8US.hex auto detected as Intel Hex
avrdude: input file codppmx8US.hex contains 2858 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 8.05s

avrdude: verifying ...
avrdude: 2858 bytes of flash verified
avrdude: reading input file "eesUS.hex"
avrdude: input file eesUS.hex auto detected as Intel Hex
avrdude: writing eeprom (12 bytes):

Writing |                                                    | 0% 0.00s
avrdude: error: usbtiny_send: error sending control message: Connection timed out
```

```
(expected 128, got -110)
Writing | ################################################## | 100% 1.20s

avrdude: 12 bytes of eeprom written
avrdude: verifying eeprom memory against eesUS.hex:
avrdude: load data eeprom data from input file eesUS.hex:
avrdude: input file eesUS.hex auto detected as Intel Hex
avrdude: input file eesUS.hex contains 12 bytes
avrdude: reading on-chip eeprom data:

Reading | ################################################## | 100% 0.54s

avrdude: verifying ...
avrdude: 12 bytes of eeprom verified
avrdude: reading input file "0xd7"
avrdude: writing hfuse (1 bytes):

Writing | ################################################## | 100% 0.02s

avrdude: 1 bytes of hfuse written
avrdude: verifying hfuse memory against 0xd7:
avrdude: load data hfuse data from input file 0xd7:
avrdude: input file 0xdf contains 1 bytes
avrdude: reading on-chip hfuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of hfuse verified
avrdude: reading input file "0xeb"
avrdude: writing lfuse (1 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 1 bytes of lfuse written
avrdude: verifying lfuse memory against 0xeb:
avrdude: load data lfuse data from input file 0xeb:
avrdude: input file 0xeb contains 1 bytes
avrdude: reading on-chip lfuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of lfuse verified

avrdude: safemode: Fuses OK (H:FF, E:D7, L:EB)

avrdude done.  Thank you.

joe@newmint:~/Desktop/minifox85$
```

Appendix 1 – Source Code Listing

File: /home/joe/arduinoSketchbook/codppmx8US/codppmx8US.ino *Page 1 of 5*

```
// codppmx8inv - ATtiny85 timing using timer1 interrupt

// Started: Feb 27, 2015  G. D. (Joe) Young <jyoung@islandnet.com>
//
// Revised: Mar/Apr/15 - see codppmx7 for earlier version(s)
//          Sep 13/15 - codppmx8 adds callsign identification to all modes. From condev2
//          Sep 16/15 - set callspeed to 40 wpm, same format all modes, detect no callsign
//          Jan 11/16 - invert Tx output for FET keying of Tx
//          Feb 27/16 - fix LED-on error, add invert possibility for LED as well as for Tx
//          Aug 15/16 - callsign at end of transmission codppmx8US
//          Aug 28/16 - correct offset interval calcn for trailing callsign
//                    - reconfig trailing calsign logic
//          Aug 29/16 - add no-overlap adjustment, fixup call insertion all modes
//          Aug 31/16 - fixup call suppression option
//
// To permit re-programming the callsign without otherwise modifying the controller
// code, the fox ID (MOE..MO5) and the call are stored in processor's eeprom at
// location 0, in the form of morse codewords (see condev2).

// An example eeprom entry:
// >>> dump eeprom
// 0000  07 0f 02 00 0b 18 3f 1d  1b 1d ff 5e ff ff ff ff  |.... .?....^....|
// which corresponds to the text: MOE WB0QYQ followed by call terminator FF and by
// call length in dots 0x5e ==> 94.

// The codeword storage format is used to avoid storing the table of ASCII-to-morse
// An arduino utility, text2code, can be used to produce an intel-hex file for loading
// into eeprom.



// There are 4 modes of operation selected by analogue value on pin 7:
//  fmode = 0  Sprint slow
//          1  Sprint fast
//          2  continuous MO
//          3  normal 1min/4min timing



#include <avr/io.h>
#include <avr/interrupt.h>
#include <EEPROM.h>

//#define TPOLARITY_POS      // Tx output active, positive true
//#define LPOLARITY_POS      // LED output active, positive true
#define TPOLARITY_OD       // Tx output open-drain negative true
#define LPOLARITY_OD       // LED output open-drain negative true

#define CALL_INTERVAL 600  // time in seconds between sending callsign
//#define THISFOX 1   // now lookup fox number from eeprom

//#define NO_OVERLAP    // to shorten ON time by one fox ID

// interrupt service variables
volatile word ovflcnt = 0;
word loopovfl;

const byte ledPin = 0;
const byte txPin = 1;
const byte ainPin = A1;

byte fmode, thisfox;
word onTime, offTime, ledtime, calltime;
boolean code = false;              // true during code sending
boolean sendm, sendo, msent, sendid, sendsp, cend, ledon;
boolean sendlsp, sendwsp, eltOn;   // loop control of morse output
unsigned long timer;
unsigned long dotLen = 221;        // dot length in millis( ) ticks (1200/wpm)1.8432
//unsigned long cdotLen = 55;  // dot length while sending call sign
```

```
unsigned long cdotLen = 111;  // dot length while sending call sign 20 wpm US
unsigned long savedotLen;

// change interrupt service to compare match interrupt
ISR( TIM1_COMPA_vect ) {
  ovflcnt += 1;
}

void setupTimer1( ) {
  TCCR1 = bit(CTC1) | 0x0e; // do not use pwm mode
  OCR1C = 224;    // 225 div with 1.8432 MHz xtal
  OCR1A = 224;    // using compare match int
  TIMSK |= bit(OCIE1A);  //add timer1 overflow interrupt
} // setupTimer1 for interrupt generation on overflow

// new functions and variables for callsign insertion
void setupNext( byte );       // functions to setup pointers to code bytes
void setupCall( );
byte setGetBit( );    // code byte parsing, now returns fence bit position 7..0
bool getBit( );
byte codeChar, nextelt, nextBit, bitCnt, nextChar, thisLength;
byte callend;
bool endCode, callsend, firstcall;
word calldurn, callintctr;



void setup( ) {

  int temp = analogRead( ainPin );      // get operating mode
  fmode = temp>>8;

  onTime = 12;
  offTime = 48;
  if( fmode == 1 ) dotLen -= 64;        // fast sprint
  savedotLen = dotLen;
  if( fmode == 3 ) {
    onTime = 60;
    offTime = 240;
  } // if normal timing
  noInterrupts( );    // attempt to sync millis( ) with timer1
  setupTimer1( );
  interrupts( );
#ifdef LPOLARITY_OD
  pinMode( ledPin, INPUT );  // setup for open drain o/p
  digitalWrite( ledPin, LOW ); // will setup no pullups, 0 in reg.
#endif
#ifdef LPOLARITY_POS
  digitalWrite( ledPin, LOW );  // setup for active high o/p
  pinMode( ledPin, OUTPUT );
#endif
#ifdef TPOLARITY_OD
  pinMode( txPin, INPUT );    // setup for open drain o/p, Tx ON pin pulls low
  digitalWrite( txPin, LOW );
#endif
#ifdef TPOLARITY_POS
  digitalWrite( txPin, LOW );
  pinMode( txPin, OUTPUT );     //active high for Tx ON with inverted version
#endif
  eltOn = false;
  cend = false;
  ledon = true;

  codeChar = EEPROM.read( 2 );  //determine fox number from EE code character
  thisfox = setGetBit( );
  callend = 4;                  //locate end of callsign
  while( EEPROM.read( callend ) != 0xff ) callend++;

  thisLength = 3;
```

```
  if( fmode == 2 ) {
    thisfox = 0;              // no fox id for continuous
    thisLength = 2;
    ledtime = ovflcnt + 30;   // 30 sec for led
  }

// Setup timers
  if( fmode == 0 || fmode == 1 ) {
    loopovfl = ovflcnt+1+(thisfox-1)*onTime;  //indexed start for sprint
  } else {
    loopovfl = ovflcnt + 1;    // no indexing for normal
  } // if sprint modes

//  calltime = ovflcnt+2;
  calltime = ovflcnt+1;
  callsend = false;
  firstcall = true;
// calculate rounded offset for trailing call insertion in seconds
  if( callend > 4 ) {         // allow for suppression of call sign
    calldurn = ( EEPROM.read( callend+1 )*60 + 500 )/1000;
  } else {
    calldurn = 0;
  }
  callintctr = CALL_INTERVAL+onTime+offTime;

#ifdef NO_OVERLAP       //adjust timing to prevent overlap due to ID completion
  if( fmode == 0 || fmode == 3 ) {
    word adjsec = ( ( 26+(thisfox<<1) )*120 + 500 )/1000;
    onTime -= adjsec;
    offTime += adjsec;
  }
  if( fmode == 1 ) {
    word adjsec = ( ( 26+(thisfox<<1) )*87 + 500 )/1000;
    onTime -= adjsec;
    offTime += adjsec;
  }
#endif

} // setup( )


void loop( ) {

  if( !code && loopovfl == ovflcnt ) {  // start sending
    loopovfl += onTime;
    dotLen = savedotLen;
    nextChar = 0;
    setupNext( nextChar );      // get first letter of fox code
    code = true;
    cend = false;
    calltime = ovflcnt;       // set call time test to start of ontime
    callsend = false;
    callintctr = callintctr-onTime-offTime;
  }

  if( code && loopovfl == ovflcnt && fmode != 2 ) {    // end sending
    loopovfl += offTime;
    cend = true;    // delay end until id complete
  }

  if( fmode==2 && ledtime == ovflcnt ) ledon = false;

  if( (ovflcnt - calltime) >= (onTime - calldurn) && fmode == 3 ) {
    callsend = true;
  }
  if( code && firstcall && fmode <= 1 && (ovflcnt-calltime) >= (onTime-calldurn) ) {
    firstcall = false;
    callsend = true;
  }
```

```
  if( !firstcall && fmode <= 1 && (ovflcnt - calltime) >= (callintctr + (onTime-calldurn)) ) {
    callintctr = CALL_INTERVAL;
    callsend = true;
  }
  if( fmode == 2 && (ovflcnt - calltime) >= CALL_INTERVAL ) {
    calltime = ovflcnt;
    callsend = true;
  }


  if( code && !eltOn ) {
    timer = millis( ) + dotLen;            // setup for dot length
    if( getBit( ) ) timer += 2*dotLen;      // sending a dash
#ifdef TPOLARITY_OD
    pinMode( txPin, OUTPUT );
#endif
#ifdef TPOLARITY_POS
    digitalWrite( txPin, HIGH );           //active high version
#endif
#ifdef LPOLARITY_OD
    if( ledon ) pinMode( ledPin, OUTPUT );
#endif
#ifdef LPOLARITY_POS
    if( ledon ) digitalWrite( ledPin, HIGH );
#endif
    eltOn = true;
  } // if not sending, start first element

  if( code && eltOn && !sendsp && ( timer < millis( ) ) ) {
#ifdef TPOLARITY_OD
    pinMode( txPin, INPUT );
#endif
#ifdef TPOLARITY_POS
    digitalWrite( txPin, LOW );           //active high version
#endif
#ifdef LPOLARITY_OD
    pinMode( ledPin, INPUT );     //Feb 17/16
#endif
#ifdef LPOLARITY_POS
    digitalWrite( ledPin, LOW );  //Feb 17/16
#endif
    timer += dotLen;    // timing space following element
    sendsp = true;
    if( endCode ) {
      endCode = false;
      sendlsp = true;
      timer += 2*dotLen;
      nextChar++;
      if( (nextChar == thisLength) || (nextChar == callend) ) {
        dotLen = savedotLen;
        sendwsp = true;
        timer += 4*dotLen;
        nextChar = 0;          // reset pointer to characters
      } // if last character of this group was just sent
      setupNext( nextChar );
      if( codeChar == 0 ) {
        sendwsp = true;
        timer += 4*dotLen;
        nextChar++;
        setupNext( nextChar );    // if space charcter to be sent
      }
    } // extend space to letter space or word space
  } // if element on time is up

  if( code && sendsp && ( timer < millis( ) ) ) {
    eltOn = false;
    sendsp = false;
    if( sendlsp ) {
      sendlsp = false;
```

```
    } // if letter space just ended
    if( sendwsp ) {
      sendwsp = false;
      if( cend ) {
        code = false;  // turn off sending
        ledon = false;   // and led after first burst
      } // if last word space, end sending
//      if( fmode == 2 && callsend ) {
//      if( fmode != 3 && callsend ) {
      if( callsend ) {
        callsend = false;
        setupCall( );
      } // if time to send a callsign
    } // if word space ended, setup to begin again
  } // if sending space

} // loop( )


// support function implementations

void setupNext( byte nextChar ) {
  codeChar = EEPROM.read( nextChar );
  setGetBit( );
} // setup to first code byte (usually for M)

void setupCall( ) {
  nextChar =  4;    // start of callsign in EE
  codeChar = EEPROM.read( nextChar );
  dotLen = cdotLen;
  if( codeChar == 0xff ) {              // allow for suppressing call
    nextChar = 0;
    codeChar = EEPROM.read( nextChar );
    dotLen = savedotLen;
  }
  setGetBit( );
} // setup to first code byte of callsign ID

byte setGetBit( ) {
  bitCnt = 7;
  nextBit = 0b10000000;
  while( (bitCnt > 0) && ((nextBit & codeChar) == 0) ) {
    nextBit = nextBit>>1;
    bitCnt--;
  } // while leading zeros before fence bit
  return bitCnt;
} // setup the code word parsing

bool getBit( ) {
  nextBit = nextBit>>1;
  bitCnt--;
  if( bitCnt == 0 ) {
    endCode = true;
  } // if last bit has just been selected
  return( (nextBit & codeChar) == nextBit );
} // get next code element to send
```

Appendix 2 – setup arduino environment

The basic arduino setup is best done by following the instructions in the 'getting started' tutorial on the www.arduino.cc web site. Then, adding the ability to use the ATtiny85 is handled from the arduino > tools > board > board manager menu. There will be an entry for ATtiny processors, just select 'install' to add the needed core files.

There is also an excellent tutorial on the Sparkfun website associated with their tiny programmer product:

https://learn.sparkfun.com/tutorials/tiny-avr-programmer-hookup-guide

An aspect of the supplied ATtiny85 core not covered in these guides concerns the minifox controller using a 1.8432 MHz crystal. None of the supplied variations includes provision for this external frequency. Consequently, you cannot directly use the 'burn bootloader' feature of the arduino IDE to program the processor's fuses (the fuses are all that's 'burned' with the 'burn bootloader' command on the ATtiny85).

The arduino IDE also does not let you pre-load the eeprom. Although, in principle, you could write an arduino program using the EEPROM library functions to run first to program the eeprom this would require the hfuse value to have already been changed. Otherwise, uploading the controller program to the flash will erase the eeprom during the chip-erase step!

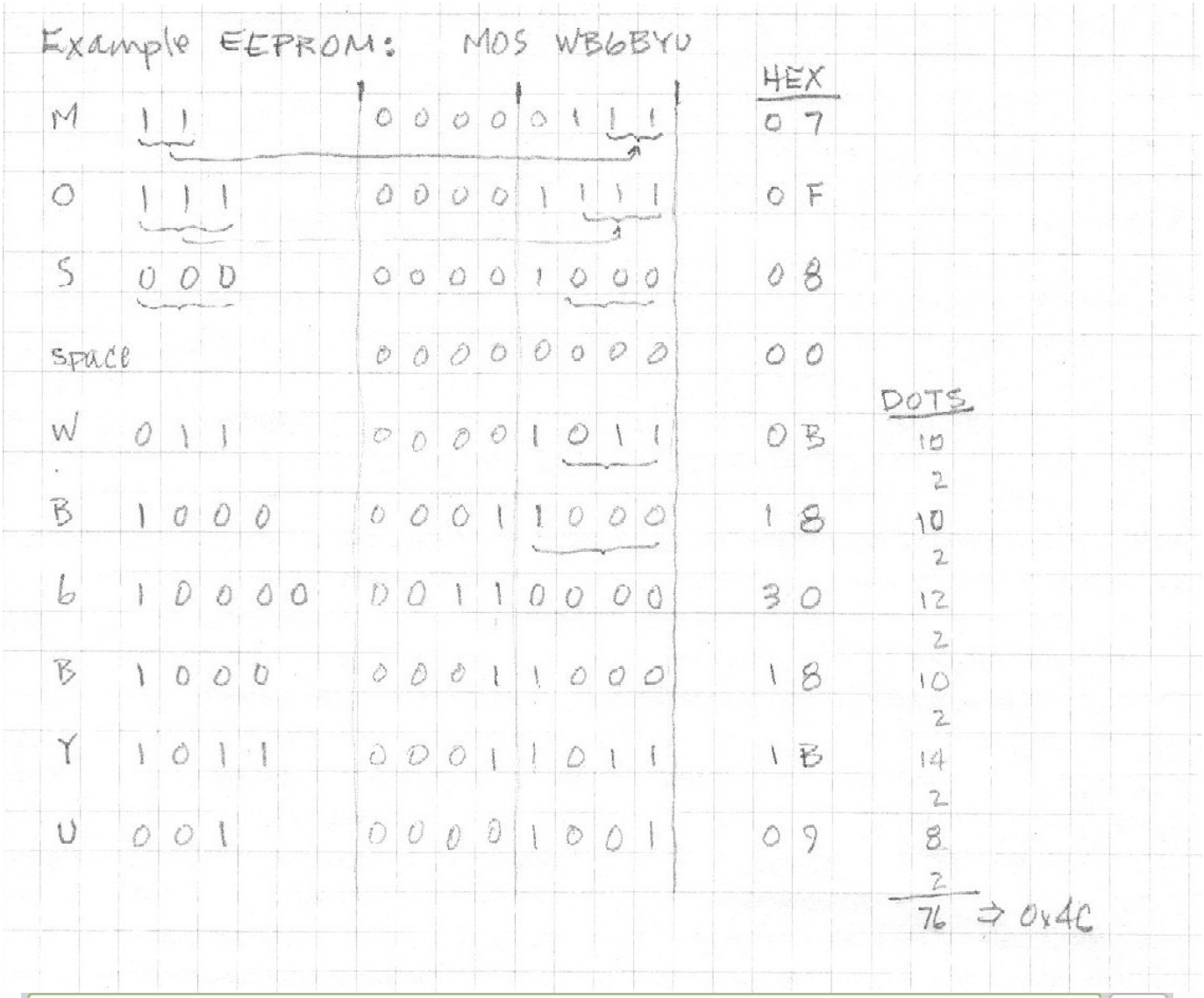
Both of these problems can be corrected to some extent by making a modification to a configuration file called boards.txt. The following group of lines needs to be inserted into the boards.txt file found in a configuration file in a folder (possibly hidden) called .arduino15/packages/attiny/hardware/avr/1.0.1/boards.txt (this is the linux path to the boards.txt file—other systems might be different). It is reasonable to place this group after the existing group for a 1.0 MHz external clock (first line).

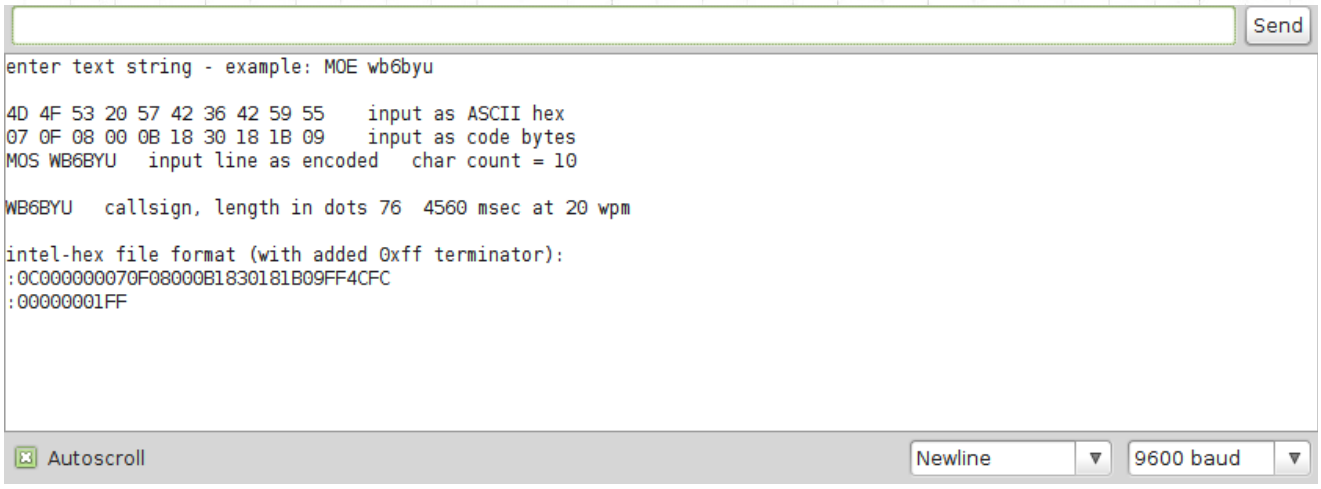
```
attiny.menu.clock.external2=1.8 MHz (external)

attiny.menu.clock.external2.bootloader.low_fuses=0xeb
attiny.menu.clock.external2.bootloader.high_fuses=0xd7
attiny.menu.clock.external2.bootloader.extended_fuses=0xff
attiny.menu.clock.external2.build.f_cpu=1843200L
```

Note that this will only permit the 'burn bootloader' operation to write the proper fuses. The actual clock frequency will not be correct at 1.8432 MHz because the arduino IDE determines the cpu clock frequency in megahertz from this parameter specification by doing an integer divide of 1843200/1000000 which results in 1. This fact is accounted for in codppmx8US by scaling desired millisecond times by 1.8432 (approximately).

Modifying this file is 'dangerous' in the sense that if a later version of the ATtiny cores is uploaded at some future time, the file will be overwritten and the change would disappear. As an alternative course, if the ATtiny cores package includes a 1.0 MHz (external) clock specification entry (not all versions of the cores pacakage seem to have this option), then that could be used without modifying the boards.txt file as the program already assumes that the clock rate calculated by the IDE will be 1 MHz, when it is actually 1.8432 MHz.

Appendix 3 – EEPROM code byte preparation example





Appendix 4 – Transmitter schematic