ARDF Transmitter Controller

Firmware Description Manual

For new versions ctrlr8 and ctrlr14

Table of Contents

Introduction

Radio direction finding has been of interest to several agencies—search and rescue, animal tracking, military operations, etc.—for some time. Amateur radio direction finding (ARDF, foxhunting, T-hunting, other names) derives from this interest and is practised as a sport in various forms. The sport for which this document is most concerned is defined by the deployment of several transmitters semi-concealed in a wooded area, which then are located by participants on foot, using hand-held directional receivers, as quickly as they are able. There have recently been introduced a couple of new versions of this sport that require development of new transmitter controllers, described below in the ARDF operating modes section.

History

The controllers described here are descendants of a somewhat lengthy development process investigating the use of small micro-controllers and their internal clock sources, finally deciding that crystal control was needed, and where new features became evident as the result of field use of the controllers. The culmination of the earlier process of development is found in the file codppmx8.ino (an arduino program) available in the .zip file package (along with various earlier versions and support materials) named codppm. Most of the development earlier than codppmx8 would only be of interest to follow along with the steps to realizing a useful working controller, and perhaps seeing how some of the early decisions came about.

There were several minor variations made in the 'final' codppmx8 to accommodate different versions of several 'mini-fox' RF sections, add features, and so forth as is normal when several folks are involved in determining what's best. Allowing some time for this dust to settle, discovering that yet more features were needed, and desiring to clean up the source code to improve readability, has resulted in this major revision renamed to ctrlr8 and a couple of new variants (ctrlr14, ctrlr914) of the controller program.

The 'mini-fox' is simply a very low-power 80 meter transmitter that consequently has a reduced range over which it can be heard using very short transmit antennas and normal receiver sensitivities. The usefulness of such a transmitter arose because the organizers of International Amateur Radio Union (IARU) foxhunts introduced two new ARDF events to accompany the 'classic' foxhunt—the Sprint, and FoxOr events. The controller timing for all of the IARU events are described in more detail below. The mini fox is suitable for the two new events because they are meant to be run over shorter distances (Sprint) or where there is a separate navigation to the approximate location after which ARDF finally locates the transmitter (FoxOr).

While this manual should be comprehensive so far as the ctrlr8, ctrlr14, and ctrlr914 variants are concerned, the previous documentation for the codppmx series may have some utility.

Features/requirements—controller variants ctrlr8, ctrlr14, ctrlr914

- Low-power, low-voltage operation – two AA cells for mini fox.
- Accurate timing: the Sprint 12 second ON time must be kept on its schedule within 1 or 2 seconds over 6 to 8 hours operating time.
- Simple operating mode selection to facilitate multiple uses for each transmitter.
- Minimize timing sequence interruption because of required sending of call sign identification.
- Likely-to-change configuration is kept in the processor EEPROM so flash code does not need to be changed.
- An indicator LED to show the circuit is working, but should shut off after a few seconds to avoid

providing a visual clue to transmitter location.

- Timing of start of ON times can be calculated automatically so that a common reset for several transmitters will simultaneously establish them all on their proper schedules.
- Ctrlr14 in addition to the above has a delayed-start operation with switch-selectable delays, low-battery operation, an optional oscillator control line (or push-to-talk control).
- Ctrlr914 in addition to above has a tone generator and output for control of foxes needing tone modulation (2 meter foxes, typically).

ARDF operating modes—normal, beacon, sprints, FoxOr

Normal mode. The 'classic' ARDF event has 5 transmitters, all operating on the same frequency, taking turns transmitting for 1 minute and then silent for 1 minute. The start times are staggered so that there is always one of the transmitters ON. A separate-frequency, continuous beacon is located near the finish of the classic course. The transmitters are usually 1 to 5 watts, spaced out so that visiting all of them will cover 5 to 10 kilometres. The course time limit is usually about 2 hours. If there is a large number of competitors, groups of up to 5 start together, and following groups start at 5 minute intervals. The 'classic' ARDF event consists of two separate days, one for 80 meters and one for 2 meters.

Sprint mode. The Sprint competition has 10 transmitters, two groups of 5 each, each group on its own frequency, transmitting the fox ID at different speeds for each group, within each group the transmitters take turns transmitting for 12 seconds, off for 48 seconds. The course is in a smaller area so that it can be completed in about 1 hour. The transmit groups are connected by a spectators corridor between the groups. There are two continuous beacons on different frequencies, one at the spectators corridor and the second at the finish.

FoxOr mode. The FoxOr event has 10 continuously transmitting transmitters, operating on very small antennas to restrict each transmitter's audible range to about 100 meters, two groups of 5 sending their fox ID at different speeds. The course is spread out to take about 2 hours to complete. To locate the foxes, a competitor is given a map with the approximate locations marked, they then navigate to the approximate location using orienteering map and compass techniques, and finally locate the transmitter with their ARDF receiver.

Beacons. The finish beacons are the same for each event, they send MO continuously for the entire event and can always be heard from anywhere on the course – hence, high power transmitters. The sprint spectator area beacon transmits S continuously. The sprint beacons operate on separate frequencies.

To summarize, if the controller is required to operate a transmitter in any of these modes, it needs to be able to select among 7 modes, each one having their unique timing requirements: Normal, Sprint slow, Sprint fast, FoxOr fast, FoxOr slow, Beacon MO, Beacon S.

Related Documents

Codppmx8... manuals available on website www.islandnet.com/~jyoung/arduinopgm.htm

codebytes.txt also on above website, among the .zip files

ATtiny85, ATtiny84 datasheets www.atmel.com

TCA Mar/Apr/17, TCA May/Jun/17 issues www.rac.ca

Program Structure

The controller programs use the normal arduino arrangement—a number of definitions of variables and functions, followed by a setup( ) section where the controller's operating mode and consequent timing parameters are established, then dropping into an infinite loop which has several timer testing statements that determine the sequence of activities.

In the setup section for ctrlr8, an interrupt service routine is established for the hardware timer1. This setup uses the CTC mode of timer1 to generate an interrupt on the CPU clock following when the timer reaches a value of 224, thus dividing the clock by 225. The clock being divided has already been reduced by the prescaler using the setting 0x0e which divides the crystal frequency by 8192. The total division is by 1843200 resulting in a once per second interrupt precisely determined by the CPU crystal frequency. The interrupt service routine simply increments a counter `ovflcnt` at each second. All of the precise interval timing is realized by tests against this `ovflcnt`. The ctrlr14 and ctrlr914 interrupt service routines are somewhat different because the ATtiny84 has 16-bit timer1 registers, and because ctrlr914 also uses faster interrupts for tone generation, but in each case the 1.8432 MHz CPU clock divides to one second ticks of `ovflcnt`.

In addition to the seconds timer, faster operations such as the Morse code sending are timed using the arduino millisecond timer, `msec( )`. `msec( )` returns a count value that is advancing at a rate 1.8432 times faster than 1 millisecond because in determining the period of the CPU clock, the arduino makes an integer division of the specified frequency by one million. That is, 1843200/1000000 ==> 1.

Controller program configuration information (fox ID, call sign, other variables) are stored in the microprocessor's EEPROM memory as described in the following section for each of the controllers. Thus, changing this configuration data only needs EEPROM modification. Re-compiling of the program stored in flash memory is not needed unless some of the more major structures need modification—see the 'compile-time options' sections following.

Header file ctrlr8.h

This file defines a number of macros to handle several of the compile-time options so that the source file is not cluttered with sequences of `#ifdef` *`option`* `....` `#endif` statements. For example, whichever of the three transmit pin control logic configurations are chosen the initial pin setup is invoked with the macro `setTxpin( )`. Then the transmit line is turned ON/OFF by invoking the macros `TxpinON( )/TxpinOFF( )`. Similar macro definitions are provided for the LED, PTT, and TONE pins.

Another function of the definitions in the header file are to re-assign the operating modes according to the number of bits used for mode selection. So, when there are 3 bits or 8 modes, all of the operating modes are defined. When there are only two bits only four modes are available. Different sets of 4 modes may be chosen, and they may be assigned to the binary mode number in various ways. These assignments are handled by the preprocessor directives in the header file.

EEPROM contents

The following example contents from a sample programmed controller IC are obtained with avrdude in interactive mode using the command `avrdude -c usbtiny -p attiny85 -t`, or `avrdude -c usbtiny -p attiny84 -t`. The `-p` parameter is changed according to which processor is used. Then, the outputs shown are then generated with the command `avrdude> dump eeprom 0x00 0x40`. The session is ended with the command `avrdude> quit`.

ctrlr8

```
0000  07 0f 04 00 11 02 38 18  12 0d ff ff ff ff ff ff  |......8.. ......|
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  |................|
0030  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

ctrlr14

```
0000  07 0f 04 00 11 05 38 0f  07 ff ff ff ff ff ff ff  |......8.........|
0010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0020  05 00 0a 00 0f 00 14 00  ff ff ff ff ff ff 05 00  |.. ..............|
0030  da 02 ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

ctrlr914

```
0000  07 0f 08 00 0c 02 00 11  02 38 18 12 0d ff ff ff  |.... ....8.. ...|
0010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
0020  05 00 0a 00 0f 00 14 00  ff ff ff ff ff ff ff ff  |.. ..............|
0030  ff 01 8e 23 57 02 ff ff  ff ff ff ff ff ff ff ff  |...#W...........|
```

In each case, the first 32 locations (hex addresses 0x00 to 0x1f) are used for the fox ID and the call sign ID stored as Morse codebytes.

The codebytes contain the dots and dashes of each character where a dot is stored as a 0 bit, dashes as a 1 bit, and the start of the code is indicated by a leading 1. A codebyte is read from left to right. For example, the codebyte at eeprom location 0x00 in each case above is hex 07, or as binary, 0000 0111. Reading from left to right, zeros are discarded until the first 1 is found, also discarded, and then the remaining two bits are sent as dash dash.

The first 3 bytes are the fox ID characters, M O I (ctrlr8, ctrlr14), M O S (ctrlr914) in the examples above. For a different fox number, only the third location needs to be altered. Location 0x03 is always 00 (a codebyte space).

The remaining bytes are reserved for the call sign ID. The examples show three variations: ctrlr8 has ve7bfk, ctrlr14 has va7om, and ctrlr914 has de ve7bfk.

The ctrlr14 eeprom also has data stored in the next 16 locations (0x20 to 0x2f). These are the two byte, little-endian(lsb first), integer delay times for the selectable durations of the delayed start. There may be up to 3 binary bits of delay selection, or 8 possible selections. Selection 0 is reserved for no startup delay, leaving 7 possible values to be stored. The last pair of locations (0x2e, 0x2f) holds the time interval at the end of the selected delay during which the transmitter will send a short identifying burst, earlier times the transmitter is silent. The example values shown in the dump are a testing set of 5 minutes (0x05 0x00), 10 minutes (0x0a, 0x00), 15 minutes (0x0f, 0x00), and 20 minutes (0x14, 0x00), three unused pairs, and the end interval set to 5 minutes (0x05, 0x00).

The ctrlr14 eeprom locations 0x30 and 0x31 hold the two byte integer value of the a/d count which is the threshold of the low-battery detection. The example is 02da hex, or 730 decimal.

The ctrlr914 eeprom in addition to all of the items above, has two more integers at locations 0x32 and 0x34. The value at 0x32 is the increment value for the tone generator which sets the tone frequency. The relationship between this value and tone frequency is 256*256*Fout/7200=increment. For example, 800 Hz uses an increment of 7282, 1000 Hz uses 9102. 9102 decimal is 238E hex, which is the value stored in the ctrlr914 contents above.

The value at 0x34 is the count for the `msec( )` timer to give the time for the PTT output to go active in

advance of the first code output. The example in the listing above is 0257 hex, or 599 decimal, corresponding to 599/1.8432 = 325 milliseconds. This time is about right for allowing a 2m HT to settle before sending tones to modulate the carrier.

Morse Code Generator

The Morse code generator is contained in a section of the program source at the end of the `loop( )`. It is set up with a pointer to the characters to send, a few control flags, and the dot duration. This setup changes according to whether the controller is in the delayed start, low battery detected, or ARDF mode of operation. The ctrlr8 does not have the delayed start or low battery detection capability. The sequence of operations is something like the following:

- lookup the character to send, setup pointer to first Morse element
- turn on the Tx, start timing a dot duration
- if the element is a dash, increase ON time by 2 dot times
- at end of element start timing a dot length space
- get next element
- if previous ON was the last element of a character, get next character and set up pointer to its elements, increase length of space being sent by 2 dot times (inter-letter space)
- if next character is a space, increase space being sent by 4 more dot times (inter word space)

This sequence repeats until all characters are sent, then restarts at beginning of the characters. Meanwhile, seconds timers will be checked to determine when to stop sending—12 seconds for sprints, 1 minute for normal mode—and a flag is set to indicate stop after completing current group. Similarly, a timer determines when to insert a call sign ID which gets set up to go following the end of currently sending group.

The apparent simultaneous occurrence is managed by starting timers, then exiting that section of code to carry on around the `loop( )` where each of the running timers are checked and appropriate flags are set to indicate which operation is happening, being waited for, etc.

Compiling the program

The three different programs—ctrlr8, ctrlr14, ctrlr914—are all compiled using the arduino programming environment. A little bit of preparation to modify the standard integrated development environment (IDE) is required to permit the IDE to produce machine language files to be uploaded into the ATtiny85 or ATtiny84 micro controllers. The extent of preparation may depend on the version of the arduino IDE you are using. In general terms, the preparation requires that the IDE be equipped with processor 'cores' for these micro controllers, and that the crystal frequency we are using be among the choices available.

To prepare for compiling (assuming the IDE is set up to know about the ATtiny85 ATtiny14), place the folder for the controller version desired (ctrlr8, ctrlr14, or ctrlr914) into your 'sketchbook' folder. Then start the arduino IDE, select 'file', 'sketchbook', locate the program name and click on that. Finally under the 'tools' menu, select the processor being used, and under 'loader' select the device programmer you're using (all the examples here use the usbtiny programmer). You can compile the program without the programmer connected using the 'verify' tab. If that goes well, you can connect the programmer with a blank microprocessor and install the flash code with 'upload'.

With recent versions of the IDE (1.5.8 and newer) the 'cores' requirement is fairly easily satisfied using the 'board manager' that's part of newer IDEs. The remaining complication arises from which one of several

cores packages to use. The programs here have been tested with the package available from Mellis, but any of them should be OK, and some of the more recent ones offer more choices than just the tiny '45, '85, and '84 processors.

The crystal frequency matter is slightly more awkward since our choice of 1.8432 MHz is not among the standard choices, and also because the arduino timer0 handlers determine the number of CPU cycles in a microsecond by integer dividing the specified clock frequency by 1000000. This determination means that our chosen frequency is regarded equivalent to 1.000000 MHz, and further means that the `msec( )` function is wrong by a factor of 1.8432. The ctrlrxx programs compensate for this frequency misunderstanding by adjusting msec times by 1.8432. For example, the dot duration for a 10 wpm code speed is conventionally calculated as 1200/10 = 120 msec. To measure a 10 wpm dot, our `msec( )` comparing timer will be set to 120 * 1.8432 = 221 (counts/120msec). There are two ways to handle the crystal frequency matter in the IDE. The simplest is to simply choose the 1 MHz option already available and remember that the frequency is actually 1.8432 MHz, and DO NOT USE the 'burn bootloader' option to set the ATtiny fuses—program the fuses separately with avrdude. The somewhat more elegant option is to modify the boards.txt file to add an entry for the 1.8432 MHz crystal and the proper values for the fuses. This option is described in an appendix.

The compile-time options described in the following sections would normally only require choosing infrequently, possibly only once when the purpose of the controller and the actual circuit configuration of its RF section is decided.

Compile-time options for ctrlr8

Specifying the compile-time options is handled by various `#define` preprocessor statements located near the top of the program source file. For the ctrlr8 program, a typical arrangement is shown below:

```
// ***** compile-time options
//#define INDEXMODENOR         // calculate start time from fox number in normal mode

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

#define CALL_INTERVAL 1800  // time interval between sending call sign

#define NRMODEBITS 3       // bits used for mode selection: 3, 2, or 1
//#define ALTMODES1BIT2      // enable alternative assignment of 4 modes

//#define TEST               // produce test o/p on select pin

#define ADJ_OVERLAP 1500  // divisor for calcn of seconds to shorten on time

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

#include "ctrlr8.h"
```

The interpretation of this particular arrangement of options and alternative options follows:

`INDEXMODENOR` in this case is disabled (commented out) and consequently the automatic calculation of the time that each normal mode transmitter starts does not occur. As a result, when the reset is released, the transmitter will start sending its fox ID immediately regardless of which fox it is. This operation might be

desirable for the case when normal mode foxes are deployed by different individuals. To start the fox, the person deploying the fox would set up the antenna, then wait until the time for that fox comes around. For example, if it's fox 2, the reset would be released when the even 1 minute happens. Other deployers would start fox 1 on the even 5 minute, fox 3 at 2 minutes after the even 5, etc.

If the INDEXMODENOR option is enabled, the program calculates the start times for each of the transmitters according to its fox number—fox 1 at even 5 minute, fox 2 at 6, fox 3 at 7, etc. This option would be desirable when all of the foxes are to be started simultaneously by connecting all their reset inputs together and a single reset button release occurs at an even 5 minute time.

The set of 5 options of the form TPOLARITY_xx and LPOLARITY_xx determine the output characteristics of the transmitter control line and the LED control line, respectively. The _xx determine which kind of output each of the pins has. _OD means 'open drain'–the line is either open for OFF or driven LOW for ON. The example simple transmitter shown in an appendix uses this mode for both Tx and LED. Note that only one of the options for each output must be selected, the others disabled. The _POS means 'active, positive true'–the output will be ON when driven HIGH, OFF when driven LOW. _NEG mean 'active, negative true' –the output will be ON when driven LOW, OFF when driven HIGH.

The CALL_INTERVAL definition takes an argument which is number of seconds between each sending of the call sign identification. The example here is 1800 which means that the call sign will be sent once each 30 minutes (Canadian minimum requirement) for all modes of operation.

The NRMODEBITS definition determines how many modes will be selectable with the mode select input. If the number is 3, that means that there will be 8 possible modes. 2 means 4 possible modes, 1 means 2 modes are possible. Choosing other than 3 (8 modes) might be useful to make the controller behave the same way as the previous codppmx8 where there were only 4 modes. The choice of 2 in that case would make a controller pin- and select resistors-compatible with the earlier version. The binary mode number is determined by the voltage present on the mode select input pin for ctrlr8. The definition of which modes are assigned to which binary mode number is determined in the header file ctrlr8.h. The definitions could conceivably be modified to make alternative assignments. One such alternative assignment is selected by choosing the next option ALTMODES1BIT2 which in the present version assigns the 4 selectable modes to FoxOr fast and slow and Sprint fast and slow. The main reason for the NRMODEBITS choice is to simplify the number of switch inputs needed in the ctrlr14 versions while also recognizing that not all fox transmitters need all the modes. For example, FoxOr transmitters would never use high-power transmitters, and so it might be useful/simpler to make special-purpose, very low power transmitters only needing the FoxOr modes—NRMODEBITS=1, and the slow and fast modes could be selected with high and low on the one select input.

The TEST option will cause the mode select pin to become an output after initial setup and this output will toggle on each of the 1 second interrupts. This signal can be measured with an external period counter with at least 6 digit resolution and used as the measure of the processor clock frequency accuracy. While measuring this 2 second interval (positive edge to positive edge), the processor crystal feedback capacitors can be selected to trim the time interval to 2.000000 +/- .000003 or less to achieve the necessary timing accuracy. The time accuracy required is +/- 2 seconds over the duration of an event—say 8 hours, which is 2 in 8*60*60, or about 7 in 10e6. That is +/- 3 is about twice as good as the IARU specification. For the normal select resistor inputs, the test output will have no trouble driving those resistors. However, if the select input is connected directly high or low, the output will source several 10's of mA, so a caution is to always use some reasonable resistance between the select pin and high and low if the test signal is used.

The ADJ_OVERLAP option reduces the allowed ON time by an amount, calculated from the fox number, which will ensure that one fox transmitting its fox id sequence (MOx) will finish in time to avoid overlapping the assigned time for the next fox. The consequences of this option are most significant with the very short Sprint modes where reducing the time by even a couple of seconds means that the transmission becomes very brief. If the #define ADJ_OVERLAP line is commented out, then no adjustment

is done and a fox will start sending its last MOx as long as the specified ON time has not ended, and consequently may overlap the next transmission by as much as the total time it takes to send its fox ID.

The parameter specified with the ADJ_OVERLAP option is used in the calculation of the time to subtract from the ON time. It is the divisor in the conversion of the MOx sending time from milliseconds to seconds. So, if it is set to 1000, the total MOx time is subtracted, which will result in no overlap, but may have silence at the end of a transmission by as much as the total time of sending its fox ID. If the parameter is larger, then the time subtracted is less than the no overlap case, but more than the case where overlap is allowed. In this example, the divisor is 1500, so the time subtracted is 2/3 of the time to transmit the MOx.

The definitions of the pins to be used for the LED output, the transmitter output and the select input are only slightly variable in ctrlr8. The only analog input available for the select resistors is A1. The LED line and the Tx line could be swapped, but that's about all. There is more scope for variation with the 14-pin processors.

Finally, the header file ctrlr8.h defines the assignment of binary input mode numbers to the controller's operating modes. These definitions could be changed if some more useful arrangement was found by altering the `#define` statements in the file ctrlr8.h

Compile-time options for ctrlr14

The '14' in this program version name comes from using the 14-pin ATtiny84 processor. This choice was made to be able to have enough inputs to allow for switch selecting several values of times to delay the start of transmission.

The purpose of the delayed-start capability is so that all of the transmitters to be used in an event can be connected together and started simultaneously with one button. Then the running transmitters are carried to the site of the event, set up before the official start, and they all start together at the event start time.

Once use of switches to select delay times has been decided, then the simplest mode selection method is to also use selection switches on the processor input pins. The configuration options for ctrlr14 use switch inputs for mode selection.

The complete set of compile-time options for ctrlr14 follows:

```
#define INDEXMODENOR         // calculate start time from fox number in normal mode

#define TPOLARITY_POS      // Tx output active, positive true
//#define TPOLARITY_NEG      // Tx output active, negative true
//#define TPOLARITY_OD       // Tx output open-drain negative true
//#define LPOLARITY_POS      // LED output active, positive true
#define LPOLARITY_OD        // LED output open-drain negative true
#define PTPOLARITY_POS      // PTT line active pos true
//#define PTPOLARITY_NEG      // PTT line active neg true
//#define PTPOLARITY_OD       // PTT line open-drain negative true

#define CALL_INTERVAL 1800    // time interval between sending call sign

//#define TEST                // output waveform on OC1A pin 7 - arduino D6

#define NRMODEBITS 2        // may be 3, 2, or 1 - see ctrlr.h
char modselectpins[ ] = { 0x08, 0x07, 0x06 };  // pins to use for mode selection

#define NRDELAYBITS 2
char dlyselectpins[ ] = { 0x02, 0x03, 0x01 };  // pins to use for delay selection
```

```
#define DLYINT 5    // interval between burst Tx during delayed start (multiple of 5)

//#define DLYSENDCALL   // include call sending during delay bursts

#define PTTADJ 1800      // msec*1.8432 to delay ptt turn on from second before code
#define ADJ_OVERLAP 1500  // adjust ON/OFF times by length of MOx. 1000 no overlap

const byte ledPin = 5;
const byte txPin = 4;
const byte ainPin = A0;
const byte pttPin = 1;
const byte vbattPin = A0;
```

The first few options—INDEXMODENOR, Tx and LED polarities, have the same meaning as for ctrlr8. An additional polarity group selects the logic level of the push-to-talk (PTT) (or oscillator control) line.

The TEST option is slightly different since it uses a specific output pin associated with the timer1 operation—enabling TEST has the 2 second waveform on IC pin 7.

The NRMODEBITS is similar to ctrlr8, but now the pins to be used for the select inputs may be specified on the next line by entering the sequence of arduino digital input names in the three byte array specification. In the example here, only two bits are needed, they will be on inputs 8 and 7 (IC pins 5 and 6). The named inputs are read in on setup most-significant-bit-first. That is, input 8 is MSB, 7 is LSB.

The NRDELAYBITS option and following array specify the inputs for selecting the delays. In this example, only two inputs are needed, they will be on digital inputs 2 and 3 (IC pins 11 and 10). The named inputs are read in on setup most-significant-bit-first. That is, input 2 is MSB, 3 is LSB.

For both of the mode bits and delay bits inputs, the processor is configured for internal pullups on the named pins, so only a single SPST switch on each bit is needed to make the selections. The sense of the switch inputs is such that OPEN yields a 0 input, CLOSED yields 1 input. So, for example, if the switch on input 7 is closed and input 8 is open, the binary input will be 01 and operating mode 1 will be selected—fast sprint in the current arrangement.

The DELAYINT option specifies the way the delay time is handled. The parameter 5 specifies that the delay is checked every 5 minutes, and if burst transmission is happening a burst ID is output. This parameter must be a multiple of 5 so that the timing of the normal mode will be right after the delay is counted down.

DLYSENDCALL if enabled will cause a call sign to be included with each burst transmission sent during the delay time. This option would substantially increase the duration of a burst, and the idea of burst transmissions is to be as short as possible so that clandestine ARDF activity is unlikely before the official event start. For this reason, it is usually disabled.

PTTADJ is a 'fine-tune' adjustment of the time the PTT line is active. The PTT line comes ON before the first element of the code starts. This feature is implemented by asserting a flag one second before the code start, and on that flag, starting a `msec( )` timer with the time interval count specified in the parameter. For the example here, the time interval count of 1800 corresponds to 1800/1.8432 = 976 msec, and so the PTT line will go ON about 24 msec before the first code element starts.

ADJ_OVERLAP works the same way as for ctrlr8. The parameter 1500 will reduce ON time by 2/3 of the duration of MOx

Compile-time options ctrlr914

All of the options for ctrlr14 except PTTADJ are included in ctrlr914. Option PTTADJ is implemented with

the adjustment interval stored in EEPROM at location 0x34. There are three additional options:

```
#define TONE_PIN 6    // specify tone on pin 5 or 6, freq by incr value in addr 0x32
//#define TONE_PTT      // tone on/off with ptt line
//#define TONE_CONT     // tone on at setup, runs continuously
```

The ATtiny84 PWM generator associated with timer1 only allows its outputs on either digital 5 or 6 (IC pins 8 or 7). Consequently, when a tone output is desired one of these two pins will have to be reserved for that purpose. In the 'Testing Breadboard' schematic included in the appendix, IC pin 8 is used for the LED and pin 7 a select input, so one or the other would need to be reassigned when tone output is enabled.

Option `TONE_PTT` will cause the tone to be turned on when the PTT line goes true, and off when PTT returns false.

Option `TONE_CONT` has the tone output on continuously.

When `TONE_PTT` and `TONE_CONT` are both disabled as in this example, the tone is keyed on/off with the Morse code output.

Leaving `TONE_PIN` undefined will disable the tone output.

Pre-compiled versions .hex files

For those who would simply like to reproduce the controller, compiled firmware files have been prepared which anticipate likely definitions of the compile-time options. If these choices are satisfactory, then the user would not need to set up the arduino environment, but can just load the flash, eeprom, and fuses using commands to the AVR programmer—examples using avrdude and the usbtiny programmer are given in the following section 'Programming the micro controllers'.

Fox ID .hex files

The files eeenocall.hex, eeinocall.hex, eesnocall.hex, eehnocall.hex, and ee5nocall.hex are for loading the first bytes of the processor eeprom with only the fox ID (MOE, MOI, MOS, MOH, or MO5). All of the controller firmwares with just these few bytes in their eeprom will be minimally functional: they will be able to start with all modes and sending only the fox ID. Other features such as delayed start, call sign insertion, low battery, etc. will be disabled because the unprogrammed eeprom locations controlling these features will be detected as invalid and defaults or disabling will be defined.

ctrlr8x .hex files

File ctrlr8E.hex is the version compiled with the options discussed above in the section 'Compile Time Options for ctrlr8'. This version might be considered as the 'upgrade' for the original mini fox shown in Appendix 2 where the extra modes and overlap control are implemented.

File ctrlr8F.hex has same options as ctrlr8E.hex except that `ADJ_OVERLAP` is disabled.

File ctrlr8Ftest.hex same options at ctrlr8F.hex with `TEST` enabled.

File ctrlr8G.hex options as for 8F, but Tx output is positive true. This output option would be used when the transmitter keying line uses a separate switching transistor—as shown, for example, in the 'ctrlr14 breadboard' circuit diagram in the appendix.

ctrlr14x .hex files

File ctrlr14N.hex is the version compiled with the options discussed above in the section 'Compile Time Options ctrlr14'.

File ctrlr14Ntest.hex is the same as version ctrlr14N, but where the `TEST` option is also enabled, which will thus have the 2 second interval signal output on IC pin 7.

ctrlr914x.hex files

File ctrlr914F.hex is the version compiled with the options discussed above.

Programming the micro controllers

ctrlr8 using ATtiny85

The following terminal session example shows how to load all memories of a new processor with a single `avrdude` command line. The text input you type is shown in bold blue, following the command are messages from avrdude reporting its progress with the command. There are four memory commands, for the flash, the eeprom, and the two fuses. The command assumes that the files ctrlr8E.hex, and eesnocall.hex are in the current directory. The long command line is actually all typed on one line, although it's shown below wrapped onto following lines.

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/ctrlr8$ avrdude -c usbtiny -p attiny85 -U
flash:w:ctrlr8E.hex -U eeprom:w:eesnocall.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 "ctrlr8E.hex"
avrdude: input file ctrlr8E.hex auto detected as Intel Hex
avrdude: writing flash (2962 bytes):

Writing | ################################################## | 100% 7.52s

avrdude: 2962 bytes of flash written
avrdude: verifying flash memory against ctrlr8E.hex:
avrdude: load data flash data from input file ctrlr8E.hex:
avrdude: input file ctrlr8E.hex auto detected as Intel Hex
avrdude: input file ctrlr8E.hex contains 2962 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 8.41s

avrdude: verifying ...
avrdude: 2962 bytes of flash verified
avrdude: reading input file "eesnocall.hex"
avrdude: input file eesnocall.hex auto detected as Intel Hex
avrdude: writing eeprom (5 bytes):

Writing |                                                    | 0% 0.00s
avrdude: error: usbtiny_send: error sending control message: Connection timed out
(expected 128, got -110)
Writing | ################################################## | 100% 1.34s
```

```
avrdude: 5 bytes of eeprom written
avrdude: verifying eeprom memory against eesnocall.hex:
avrdude: load data eeprom data from input file eesnocall.hex:
avrdude: input file eesnocall.hex auto detected as Intel Hex
avrdude: input file eesnocall.hex contains 5 bytes
avrdude: reading on-chip eeprom data:

Reading | ################################################## | 100% 0.36s

avrdude: verifying ...
avrdude: 5 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 0xd7 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% 0.02s

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 (E:FF, H:D7, L:EB)

avrdude done.  Thank you.

joe@newmint:~/Desktop/ctrlr8$
```

ctrlr14, ctrlr914 using ATtiny84

The command for loading the ctrlr14N.hex file plus the eeprom minimum entry .hex file plus the fuse settings is very similar to that used for the ctrlr8E.hex example above:

```
joe@newmint:~/Desktop/ctrlr8$ avrdude -c usbtiny -p attiny84 -U
```

```
flash:w:ctrlr14N.hex -U eeprom:w:eesnocall.hex -U hfuse:w:0xd7:m -U
lfuse:w:0xdb:m
```

The output from `avrdude` would be practically identical to that shown above and is not repeated here. However, note that the `lfuse` value is not the same as for the ctrlr8 command.

Programming the EEPROM for ctrlr8, ctrlr14, and ctrlr914

Most of what you need to know about the micro processors eeprom programming was covered above in the descriptions in the section 'EEPROM contents' above. The most straightforward way to get the few bytes needed for configuring the controller is to use the programmer's 'interactive mode' which is invoked with the command:

```
avrdude -c usbtiny -p attiny84 -t
```

Then, the bytes to be stored in the eeprom can be written with commands of the form:

```
avrdude> write eeprom address databyte databyte ...
```

where the bold (blue) is text the user types, the black is echoed from avrdude. *address* is the hexadecimal starting address to place the following hexadecimal data bytes into consecutive locations.

As a simple example, suppose you wish to change the controller's fox ID from whatever it is now to be MO5. The only byte that needs to change is at the location where the single codebyte for the character 5 will go. To make this change while in the interactive mode of avrdude the command is:

```
avrdude> write eeprom 0x02 0x20
```

As a more extensive example, suppose you wish to enter the call sign ve7ajt which is to be inserted at call ID intervals. The codebytes needed are:

v	10001	0x11
e	10	0x02
7	111000	0x38
a	101	0x05
j	10111	0x17
t	11	0x03

and then the interactive avrdude command would be:

```
avrdude> write eeprom 0x04 0x11 0x02 0x38 0x05 0x17 0x03
```

The call sign storage starts at address 0x04 in the eeprom.

If you do have the arduino environment running and have a standard arduino board there is a little utility program called text2code which will generate a .hex file from text entered into the serial monitor which may be used to create a .hex file of codebytes to load into eeprom following the example of the eesnocall.hex file above.

To enter the numerical parameters for the startup delay, the battery threshold, the PTT offset, and the tone increment, simply convert the decimal numbers to hexadecimal, then enter them into the eeprom where they belong, low-byte first. For example, for the first selected delay to be 30 minutes, the hexadecimal value is 0x1e, the avrdude command would be:

```
avrdude> write eeprom 0x20 0x0e 0x01
```

Remember to exit from the avrdude interactive mode with the command:

```
avrdude> quit
```

Appendix 1 – adding an entry in boards.txt

The boards.txt file is added along with several other folders and files to a hidden directory (linux) .arduino15 when the ATtiny package is loaded with the boards manager. The file is fairly deeply tucked inside, the complete path is something like:

~/.arduino15/packages/attiny/hardware/avr/1.0.2/boards.txt

Just searching for boards.txt doesn't necessarily find the one you need because there may be several—at least one other one is always present in the arduino folder. The one you want is the one associated with the add-on ATtiny package. The location has moved around a bit with versions of arduino, and the format of the file has changed as well. This example is for arduino 1.8.1.

I suggest locating a block of definitions looking like the second block below (beginning with the line `ATtinyX5.menu.clock.external8=External 8 MHz`), copying that block into the space above it in the file, and then modifying as shown in the first block below.

```
ATtinyX5.menu.clock.external2=External 1.8432 MHz
ATtinyX5.menu.clock.external2.bootloader.low_fuses=0xeb
ATtinyX5.menu.clock.external2.bootloader.high_fuses=0xd7
ATtinyX5.menu.clock.external2.bootloader.extended_fuses=0xff
ATtinyX5.menu.clock.external2.build.f_cpu=1843200L

ATtinyX5.menu.clock.external8=External 8 MHz
ATtinyX5.menu.clock.external8.bootloader.low_fuses=0xfe
ATtinyX5.menu.clock.external8.bootloader.high_fuses=0xdf
ATtinyX5.menu.clock.external8.bootloader.extended_fuses=0xff
AttinyX5.menu.clock.external8.build.f_cpu=8000000L
```

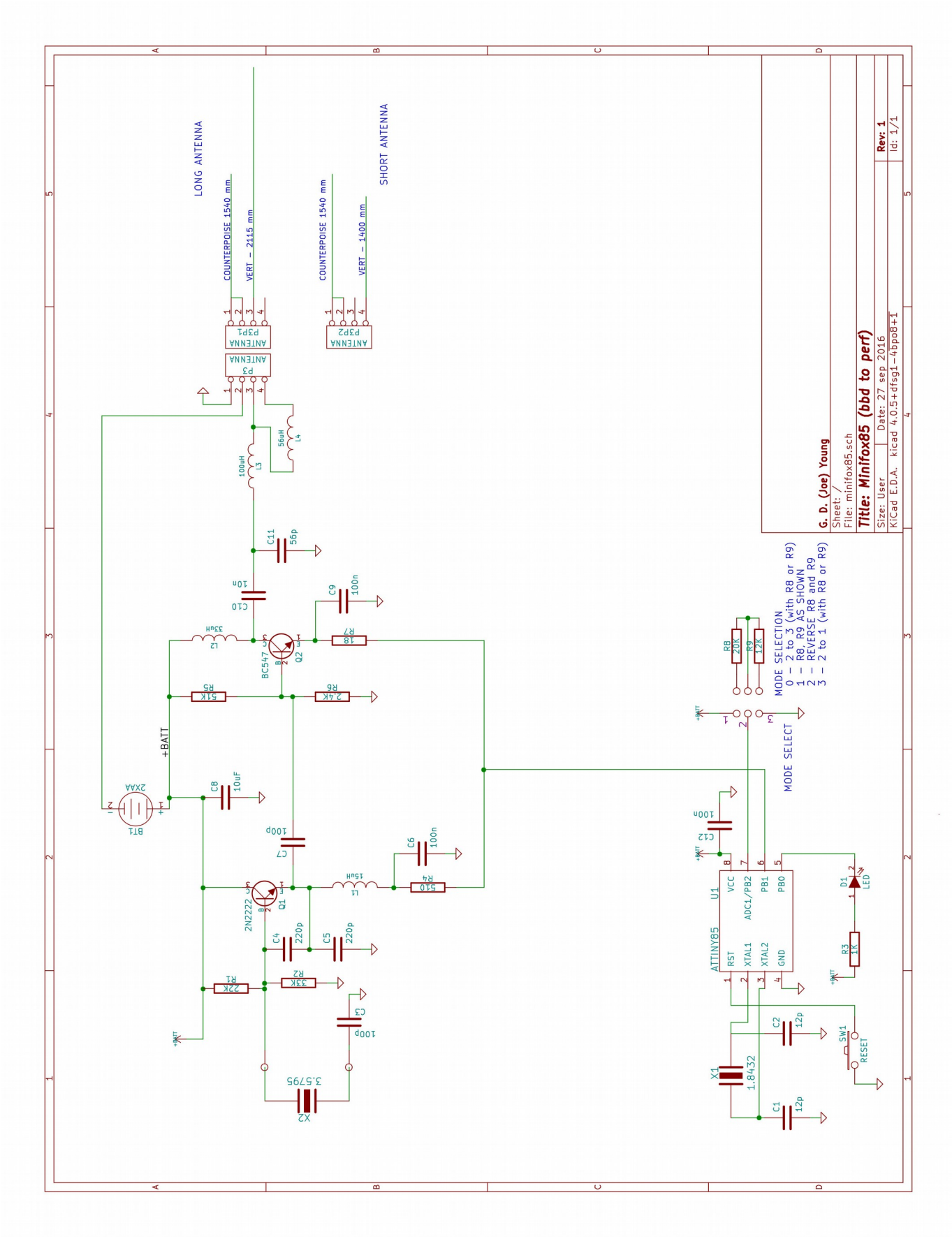
A similar modification will be needed in the section of the file defining things for the ATtiny84 (note different lfuse value):

```
ATtinyX4.menu.clock.external2=External 1.8432 MHz
ATtinyX4.menu.clock.external2.bootloader.low_fuses=0xdb
ATtinyX4.menu.clock.external2.bootloader.high_fuses=0xd7
ATtinyX4.menu.clock.external2.bootloader.extended_fuses=0xff
ATtinyX4.menu.clock.external2.build.f_cpu=1843200L
```

Note also that these additions will disappear if you later update the package with the boards manager because the boards manager will overwrite your modified file with the update package's boards.txt.

The main benefits of making this change is that you can now program the fuses in a fresh-from-the-store micro controller using the arduino 'burn bootloader' command, and it is a reminder that you're using an unusual clock frequency.

Appendix 2 – example mini fox transmitter



Appendix 3 – Test bench breadboard for ctrlr14 and ctrlr914