FT-891 CAT Emulator Library
Version 1.1
John Price – WA2FZW

Table of Contents

Introduction

This library provides a means of adding CAT (Computer Aided Transceiver) functionality to Arduino (or other microprocessor) based applications. I have used it in an ESP32 based digital VFO and my simple interface for digital audio, which runs on an Arduino Nano.

This was originally developed for a digital VFO project by me, Glenn (VK3PE) and Jim (G3ZQC).

The CAT control language used is that used by the Yaesu FT-891 with a couple of wrinkles. Why that language instead of the more commonly used Kenwood language variations? There are a couple of reasons:

- I have an FT-891 and am thoroughly familiar with the CAT control capabilities.
- The primary CAT control program that I use is *DxCommander* and when I first got the FT-891, I discovered that there were a number of issues when using the two together. Some of these issues were misunderstandings by the *DxCommander* author, and others were due to bugs in the Yaesu software. Working with Dave (AA6YQ, the author of *DxCommander*), we either fixed all of the issues or invented work-arounds.
- The message handler software was 90% written, as I've used it in two other projects, so no need to reinvent the wheel.

Modification History

In version 1.1, the library was modified so that the action of actually keying or un-keying the transmitter upon receipt of a "TXn;" command is now optional. This was done by adding a 2nd version of the *begin* function and is explained in more detail below.

CAT Implementation

Although the CAT implementation uses the Yaesu FT-891 language, only a small sub-set of the language is implemented. Some functions are not implemented simply because the radios that I and a couple of others plan on using this with can't possibly support them; for example, software adjustable filters.

Others are omitted simply because they are; we don't provide the capability to save and transmit voice or CW messages, nor is a built-in keyer implemented.

General Architecture

There are two main components of the library; a status table, which contains the current status of the (simulated) radio (operating frequency, mode, etc.) and the message handler itself.

The software is designed to be 100% compatible with the *DxCommander* program, however should work with other CAT control capable applications such as *N1MM+* or *WSJT-X* for example provided those programs don't have some of the issues that we corrected in *DxCommander*.

Message Handler

DxCommander uses a polling scheme to request the status of the radio and also sends commands to the radio. Using this approach means that *DxCommander* can initiate changes in the status of the radio, or when changes in the status of the radio are made by the application using this library or my manual operation of the radio's controls *DxCommander* will be informed of those changes.

For example, in the VFO program for which this was developed, one can change frequency not only via CAT control, but by operating an encoder. When the encoder is moved, the application uses the *SetFA()* method in the library to update the library's internal status table and the next time *DxCommander* asks for the VFO-A frequency, it will get the updated value.

Not all of the FT-891 commands and status requests are implemented. The focus of this project has been to use the VFO to upgrade legacy radios, so many of the commands just can't be implemented as they could be in a modern software controlled radio, however, some of the messages must be processed to keep *DxCommander* happy.

The commands/status requests that are recognized (in alphabetical order), are:

AB	Copy VFO-A to VFO-B
AI	(Data 0 or 1) Turn auto-information on or off
BA	Copy VFO-B to VFO-A

BS	Band select (not yet implemented)
EX	Menu command (does nothing but needs to be here)
FA	Set or request VFO-A frequency
FB	Set or request VFO-B frequency
ID	Request radio's ID (0650 for the FT-891)
IF	Information request/answer
IS	Set or request IF shift
MD	Set or request mode (USB, LSB, CW, etc.)
NA	Set or request narrow IF shift
OI	Opposite Band Information request/answer
RIC	Alternate request for split status
RM	Read meter
SH	Set or request IF bandwidth
SM	Read S-meter
ST	(Data 0 - 2) Split mode off, on or on +5KHz up
SV	Swap VFOs
TX	Set or request transmit/receive status

The descriptions of some of the messages are a bit misleading, but the *Yaesu FT891 CAT Manual* will clarify what they actually do.

As some of the command messages cannot actually be implemented on the legacy radios that we have been playing with, such as the ones related to filter settings, however, *DxCommander* can try to set these parameters and it requests their status in its routine polling sequence, thus they have to be handled although they do nothing.

Note that if the setting of something cannot actually change something in the using application, the command to do so will be ignored. This is done by commenting out the code to effect the change in the *FT891_CAT.cpp* file. For example, the *case* statement in the *ProcessCmd* function to set the IF shift as distributed looks like this:

```
case MSG_IS:                                        // Set IF shift & status
//      strncpy ( tempBuff, &dataBuff[1], 5 );      // Extract the offset amount
//      radioStatus.IS_VALUE = atoi ( tempBuff );   // Set value in the structure
//      radioStatus.IS_STAT = dataBuff[0] - '0';    // Set on/off status break;
```

By leaving the code intact and just commenting it out allows one to easily turn it on for their application.

The basic architecture of the message handler consists of a lookup table that is used to translate the alphanumeric message into a number and a pair of “switch” statements to determine what to do with the message; one for command type messages and one for status requests.

This approach makes it fairly easy to add messages to the list that you might need for your particular application or to modify the code to use a different command language such as one of the Kenwood variations (which are very similar to the FT-891 language).

Status Table

As previously mentioned the message handler maintains a table containing the current status of the parameters that can be set and/or requested by the messages listed above.

Anytime *DxCommander* changes the value of one of the parameters (only those that can actually be changed in the radio), the table is updated. When *DxCommander* requests the status of something, the answer is provided based on the contents of the table.

For those parameters that cannot actually be changed, the table contains values that will keep *DxCommander* happy when it asks for them.

The application using this library can also change many of the parameters through function calls to the CAT control library. For example, when one moves the encoder to change the VFO-A frequency, the VFO program calls *CAT.SetFA()* to change the VFO-A value stored in the library.

The using application will need to routinely check the status of things that might be changed as a result of a received command. For example, it calls the *CAT.GetFA()* function to see if the VFO-A frequency has been changed via a CAT command.

There are similar get and set methods for mode, VFO-B frequency, split status, etc.

Main Software Functions

There are only a few functions associated with the processing of the commands:

FT891_CAT	The constructor function which creates the CAT control module object.
Begin	This function is overloaded (there are a number of ways it can be called) and will be explained in greater detail below.
CheckCAT	The application must call this function periodically (in the *loop* function) to see if there is a new command to be processed. It returns a *true* or *false* indication if a new command has been processed.
GetMessage	This function checks for incoming data on the USB port and if there is incoming data, it simply reads it.
FindMsg	This function looks for the ASCII message in the lookup table and converts it to the internal numerical representation.
ParseMsg	If the message came with data attached, this function separates the data from the message.
ProcessCmd	If the message is a command, a “switch” statement takes care of updating the status table and performing any other actions that might be required (e.g., turning the transmitter on or off).
ProcessStatus	If the message is a status request, this function formats a proper answer to the request based on the information in the status table and sends it to *DxCommander*.
Xtoi	Works like the standard *atoi* C function, except it converts hexadecimal numbers (in various formats) to integers.
Init	This function handles the common processing for the variations of the *begin* function.

Except for the constructor the *begin* function and *CheckCAT*, all those listed above are private functions that cannot be accessed from the using application.

There are also a number of public functions that the application can use to update the status of things in the status table. These currently include:

```
void SetFA ( uint32_t freq )     // Set VFO-A frequency
void SetFB ( uint32_t freq )     // Set VFO-B frequency
void SetMDA ( uint8_t mode )     // Set VFO-A mode
void SetMDB ( uint8_t mode )     // Set VFO-B mode
void SetTX ( uint8_t tx )        // Set Transmit/receive status
void SetST ( uint8_t tx )        // Set split mode on or off
```

There are also a number of public functions that the application can use to get the current status of things. These include:

```
uint32_t GetFA ();               // Get VFO-A frequency
uint32_t GetFB ();               // Get VFO-B frequency
uint8_t  GetMDA ();              // Get VFO-A mode
uint8_t  GetMDB ();              // Get VFO-B mode
bool     GetTX ();               // Get transmit/receive status
bool     GetST ();               // Get split mode status
```

The *begin* Function

In Version 1.1 of the library, I added a second version of the *begin* function. The two function calls available are:

```
begin ( int txPin, bool debug );
begin ( bool debug );
```

In the first version, the application can specify a GPIO pin that will be used to key the transmitter when a “TX1;” or “TX2;” command is received from the CAT control program.

If the second form of the function call is used, the library will not operate the transmitter when a transmit command is received, rather that will be up to the application.

In either case, the *debug* argument is optional and defaults to *‘false’*.

But what does the *debug* argument do when set to '*true*' one might ask.

The FT-891 commands are not terminated with either a newline or carriage return character; the semicolon is the terminator. However, if one is trying to debug an application using the Arduino IDE's serial monitor function, the responses to commands and status requests sent to the application via the serial monitor are much easier to read if they are terminated by a newline.

Setting the *debug* argument to '*true*' will cause a newline to be added to the responses. Note however that the newline will confuse *DxCommander* so the argument should be set to '*false*' when the application is in actual use.

Transmitter Keying

As mentioned in the previous section, if a GPIO pin is specified to be used to key or un-key the radio that will be done by the library. In the header file for the library, there are two definitions that you might need to modify:

```
#define    XMIT_ON    HIGH
#define    XMIT_OFF   LOW
```

If your hardware implementation keys the transmitter by setting a LOW condition on the specified pin, simply flip-flop the definitions.

Required Definitions in the Application

There are only one or two things that the application must provide to the library; the bitrate to be used on the serial port and (optionally) the pin number to be used to key the transmitter.

These can be defined as symbols or variables or simply hard coded into the calls to the *Serial.begin()* and *CAT.begin()* functions (a very poor coding practice)

Message Table

The *msgTable* is an array with an entry for each message that we handle. Each entry in that array is a *msg* structure that contains the ASCII message text (*Name*), the internal numerical representation (*ID*) and a byte indicating whether the message is a command, a status request, or if it could be either depending on whether it came with attached data or not (*Type*). The *Type* indicators are individual bits defined as *MSG_STS* for a status only message, *MSG_CMD* for a command only message or *MSG_BOTH* indicating that it could be either (note, *MSG_BOTH = MSG_STS | MSG_CMD*).

Suggestion Box

I welcome any suggestions for further improvements. Please feel free to email me at WA2FZW@ARRL.net.