Tuner Studio Operating System Manual

Firmware Version 0.3

27 August 2023

Henry Wright

Contents

1 Summary

This program is intended to run on Arduino based systems programed via the Arduino IDE such as the Arduino Mega2560. It enables communication with Tuner Studio (TS), a calibration and data acquisition tool available from https://www.tunerstudio.com/

The basic functionality is to:

1. Define variables which are saved to and recalled from the internal memory (EEPROM) of the Arduino.
2. Send the variables in memory to TS and receive new data when variables in TS are changed.
3. Write (Burn) these variables to the EEPROM.
4. Send measurement variables at a defined rate to TS for display and logging.
5. Receive commands to execute certain functions on the Arduino. i.e. switch an output on or off.
6. Provide a task based timing system to execute tasks with feedback if those tasks have overrun.
7. Implement a simple CAN_BUS broadcast and message object receive implementation using the MCP2515 transceiver over spi.

This code provides the bare bones of communication and forms the basis for a functionality created by the user to be added to this code.

The Latest release can be found in the ->release sub directory.

Documentation can be found in the ->docs sub directory.

Latest version of this program can be found on github: https://github.com/HWright9/TunerStudioOS.

2 Quick Start

2.1 Getting Connected

A quick overview to get started compiling and running the program. Details are in the sections below.

1. You need an Arduino Mega2650 Uno or Nano. Just for now run it with no shields or connections other than the USB to the PC.
2. Download Tuner Studio from https://www.tunerstudio.com/. For developers the paid version will be the most useful, however the program will work with the free version.
3. Download and install latest Arduino IDE https://www.arduino.cc/.
4. Download the entire program folder from github https://github.com/HWright9/TunerStudioOS. I Recommend placing it in the Documents -> Arduino directory.
5. Either compile and download the program to the Arduino using the IDE or use the release.hex file in ->release to flash the controller.
    a. Note, if re-using an old Arduino, best to run the EEPROM_clear sketch from examples to make sure EEPROM does not contain any leftover data.
6. Open tuner studio and start a new project. When prompted browse for the release.ini file in the -> release folder.
    a. Make sure you select the size of the EEPROM in the project configuration. 1KB for UNO or Nano and 8KB for Mega.
7. Open the baseline.msq to load initial variables into Tuner Studio.

8. Connect to the controller on a serial port. The bitrate is 115200. The com port will be the same as used with the Arduino IDE. (Hint: You have to be offline with Tuner Studio to flash new code to the Arduino).
9. If prompted, write all variables to the controller.
10. Make sure the burn is complete.
11. Reset the controller.
12. You should now be connected.

Your screen should look similar to the image below:

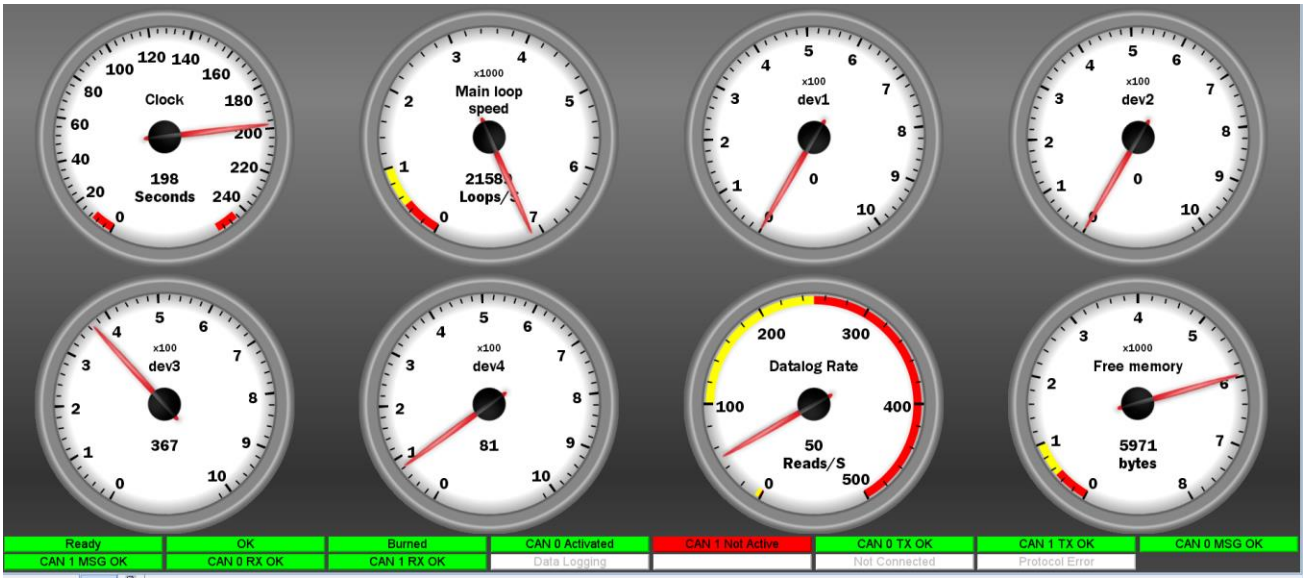

Figure 1 Main Default Gauge Cluster

2.2 Read Port Data and Control Outputs

The hardware testing button has a few options. Open them all and you will see

1. The status of all digital ports, whether they are input or output. On = green = Logical high (5V).
    a. Note1. The Mega only has 54 ports, even though 64 are shown.
    b. Note2. The port numbering matches that shown on the Arduino case.
2. The Analog inputs as shown on a live graph. They are also available as gauges on the main screen.
3. The Test ports section where you can override the status of a digital port.

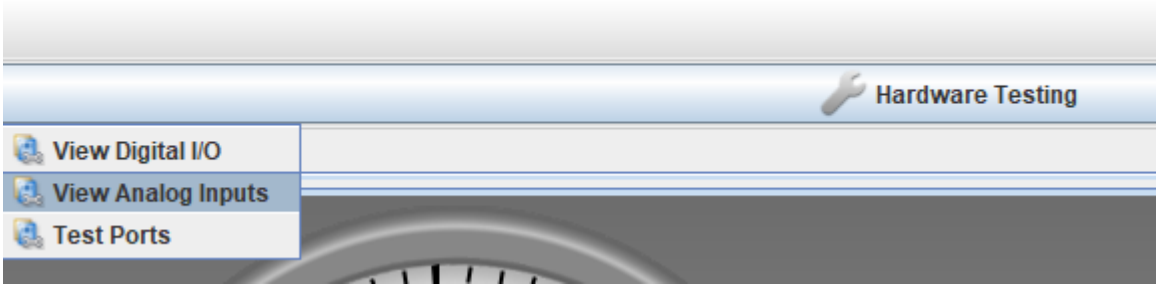

Figure 2 Menu Items in Hardware Testing

If you head to the Test Ports section. You should be able to “enable Test Mode.” If it is greyed out you first need to enable “Allow Hardware Test Commands” in system settings.

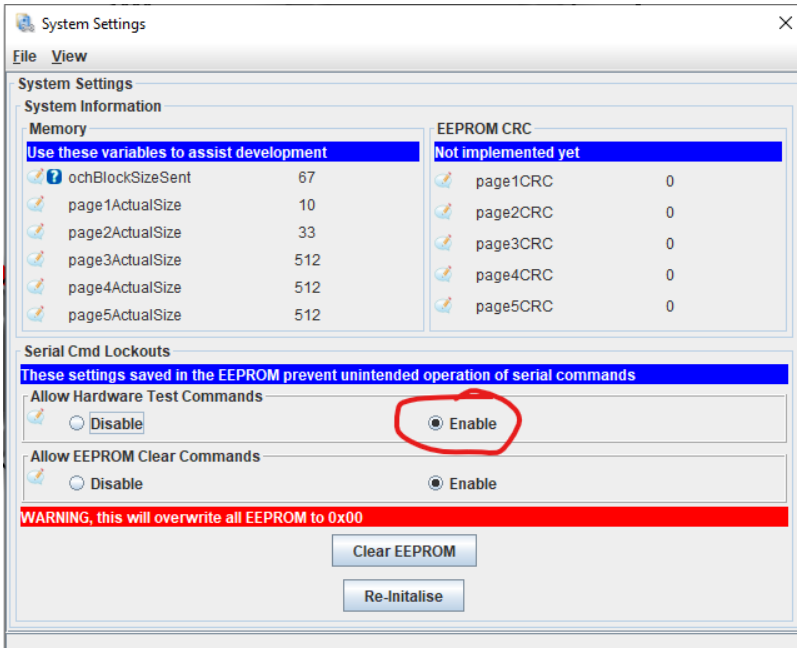

Figure 3 System settings

Once that is working, you should be able to manually override each port. Note the following:

1. Ports defined in the code as INPUT will change the value seen by the program but will not change the physical port.
2. Ports defined as OUTPUT will change the physical state of the port as well as what is seen by the program.
3. Remember that some ports are used for other functions such as serial communication and you may not be able to change them or changing them causes errors in other functions.
4. Your controller may not actually support all the ports and analogue channels shown.

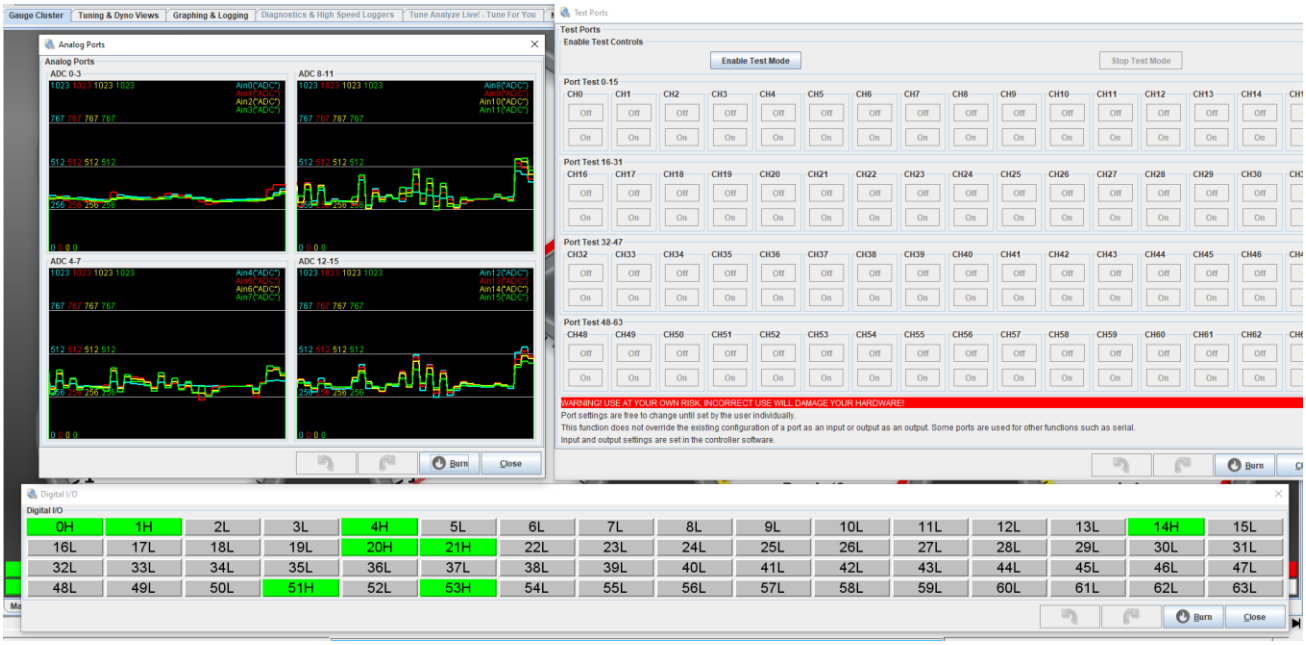

Figure 4 Hardware Testing and Override View

2.3 Inserting your own code

The code inherits all the normal Arduino libraries. When starting out, it's best to open the whole project in a reasonable C editor such as notepad++. Open the whole folder as a workspace so you can quickly navigate between files.

Open userfunctions.ino

Write a simple function, such as

```
/* USER_blinkCEL
* Run Rate: 500ms
* Toggles the state of the builtin LED on the Arduino.
*/
void USER_blinkCEL(void)
{
  if (readDigitalPort(LED_BUILTIN) == HIGH)
  {
    setDigitalPort(LED_BUILTIN, LOW, OUTPUT_NORMAL);
  }
  else
  {
    setDigitalPort(LED_BUILTIN, HIGH, OUTPUT_NORMAL);
  }
}
```

Make sure you add the prototype of the function to the userfunctions.h file. Then you head over to the TunerStudio_OS_Dev.ino file and find the correct task rate to call the function.

```
/*
* Function: 500msTask() everything in this function runs once every 500ms, 2Hz
* Returns: none
*/
void FUNC_500msTask(void)
{
  canBroadcast_500ms();
  USER_blinkCEL(); // blink the build in LED for heartbeat.
} //END 500ms Task
```

Notes:

1. Avoid writing a lot of code and if statements in the task schedulers in TunerStudio_OS_Dev.ino, it gets messy fast. Much better to create a function then just call it from the appopriate FUNC_XXXmsTask.
2. Note how we are using setDigitalPort and readDigitalPort instead of digitalRead and digitalWrite. This wrapper is what allows the Test port commands to work.

When you are done. Save your work and compile the code in the Arduino. Then download and run the new program.

2.4 Adding a Measure Variable (Sent to Tuner Studio)

To add a new measure variable, you need to add a signal to the structure Out_TS. You do this by editing the Out_TS_t typedef in globals.h

Valid measure variable types are:

BitField (8 bytes minimum), U08, S08, U16, S16, U16, S32 or F32.

Once you add it here you can reference it in code using the prefix Out_TS.Vars.new_variable_name

It's important to make sure that the order, and unit types specified here exactly match what is specified in the TunerStudioOS.ini file otherwise you will have communication problems.

```
/* The global serial transmit status object.
* All variables in this list will be transmitted to Tuner Studio in the order presented here.
* Its critical that the order of variables here must match exactly what tuner studio .ini file is set up to recieve in the [OutputChannels] section.
* The total size of this variable is captured in ochBlockSizeSent which can be read in Tuner Studio on page 1.
* Its highly reccomended to keep the variable names the same between the .ini file and this code.
*/
typedef struct Out_TS_t
{
  uint8_t secl; // counter of seconds 0-255 looping, required for TS comms.
  uint8_t systembits; //system status bits
  uint8_t LoopDlyWarnBits; //indicator that a l
  uint8_t canstatus;    //canstatus bitfield
  uint16_t canRXmsg_dflt; //check if CAN RX messages are defaulted due to RX timeout
  uint16_t loopsPerSecond;
  uint16_t readsPerSecond; // how many datalog reads in the last sec
  uint16_t UTIL_freeRam;
  uint8_t testIO_hardware;//testIO_hardware
  uint16_t digitalPorts0_15_out;
  uint16_t digitalPorts16_31_out;
  uint16_t digitalPorts32_47_out;
  uint16_t digitalPorts48_63_out;
  uint16_t Analog[16];    // 16bit analog value data array for local analog(0-15)
```

Figure 5 Out_TS_t Typedef Example

In the TunerStudioOS.ini file find the [OutputChannels] section.

Make sure you insert your variable in exactly the same order as in the Out_TS_t structure. Use the nextOffset variable to automatically set the next corresponding byte address. If defining a bitfield you have to use lastOffset to prevent incrementing the address for each bit inside a byte address.

It is also possible to exactly specify each byte address, however this gets tedious to renumber all of them if during development you want to insert a new variable between existing variables.

Follow the examples in the source code.

Tuner Studio also needs to know how many bytes to expect with the ochBlockSize variable. The size is the size of the Out_TS_t structure, if you know how many bytes you added you can increment this number by that many bytes. If you are unsure, then this number is transmitted to Tuner studio with the ochBlockSizeSent parameter, however it has to be manually updated in the OutputChannels section.

If you have comms issues after an update to the OutputChannels the ochBlockSize being incorrect is usually the cause.

```
[OutputChannels]
   ; The number of bytes MegaTune or TunerStudio should expect as a result
   ; of sending the "A" command to Speeduino is determined
   ; by the value of ochBlockSize, so be very careful when
   ; you change it.

#if FAST_COMMS
   ochGetCommand    = "r\$tsCanId\x3C%2o%2c"
#else
   ochGetCommand = "A"
#endif
   ; this size must match the total byte size of the channels to be recieved. Check ochBlockSizeSent
   ochBlockSize     =  70

   secl             = scalar, U08,  0, "sec",    1.000, 0.000

   system           = scalar, U08,  nextOffset, "bits",   1.000, 0.000
    sys_ready       = bits,    U08,    lastOffset, [0:0]
    sys_warn        = bits,    U08,    lastOffset, [1:1]
    sparel_1c       = bits,    U08,    lastOffset, [2:2]
    sparel_1d       = bits,    U08,    lastOffset, [3:3]
    sparel_1e       = bits,    U08,    lastOffset, [4:4]
    sparel_1f       = bits,    U08,    lastOffset, [5:5]
    sparel_1g       = bits,    U08,    lastOffset, [6:6]
    burn_good       = bits,    U08,    lastOffset, [7:7]

   TIMR_LoopDlyWarnBits bits = scalar, U08,  nextOffset, "bits",   1.000, 0.000
    TimerDelayWarn_1000MS      = bits,    U08,    lastOffset, [0:0]
    TimerDelayWarn_500MS       = bits,    U08,    lastOffset, [1:1]
    TimerDelayWarn_250MS       = bits,    U08,    lastOffset, [2:2]
    TimerDelayWarn_100MS       = bits,    U08,    lastOffset, [3:3]
    TimerDelayWarn_75MS        = bits,    U08,    lastOffset, [4:4]
    TimerDelayWarn_50MS        = bits,    U08,    lastOffset, [5:5]
    TimerDelayWarn_20MS        = bits,    U08,    lastOffset, [6:6]
    TimerDelayWarn_5MS         = bits,    U08,    lastOffset, [7:7]

   canstatus        = scalar, U08,  nextOffset, "bits",   1.000, 0.000
    can0Activated   = bits,    U08,    lastOffset, [0:0]
    can0Failed      = bits,    U08,    lastOffset, [1:1]
    can1Activated   = bits,    U08,    lastOffset, [2:2]
```

Figure 6. Example of OutputChannels

The System Settings window contains the correct size of ochBlockSize Variable sent from the Arduino.

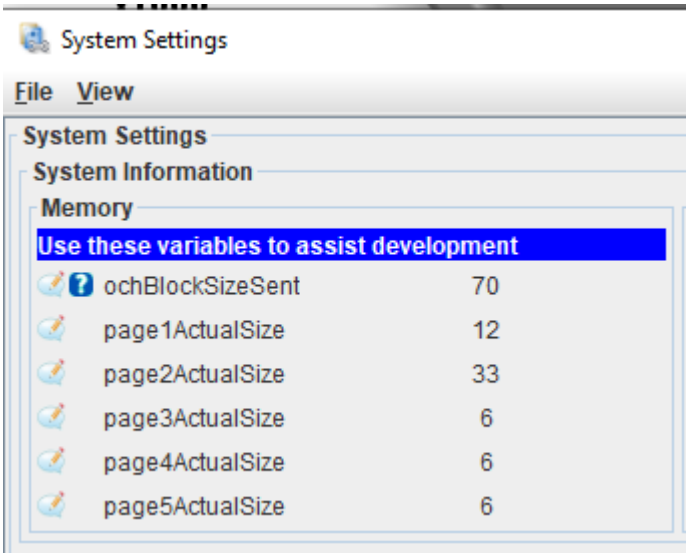

Figure 7. System Settings showing ochBlockSizeSent and page sizes. The values here are calculated and available when connected to the Arduino

Once the .ini file and code are updated, upload the code to the Arduino and then reload the TunerStudio.ini file. TS should automatically connect.

If there are coms issues after an update, you can use the comms debug log in TS to get more information. Most times it is a problem with the ochBlockSize not being correct or the variable order not matching.

2.5 Adding a Calibration Variable (Saved to EEPROM)

A calibration variable exists in the RAM and also is synchronized with the EEPROM and Tuner Studio. The process is:

When the Arduino powers up the EEPROM is copied to the RAM.

When Tuner Studio powers up it will request the RAM copy to be sent via the serial link. Tuner studio then compares this data with its own copy of the calibration parameters.

If a variable requires updating Tuner Studio sends the variable to Arduino RAM via the serial link but the data is not actually saved to EEPROM until a burn command is issued. Then the Arduino writes any changes in the RAM variables to EEPROM.

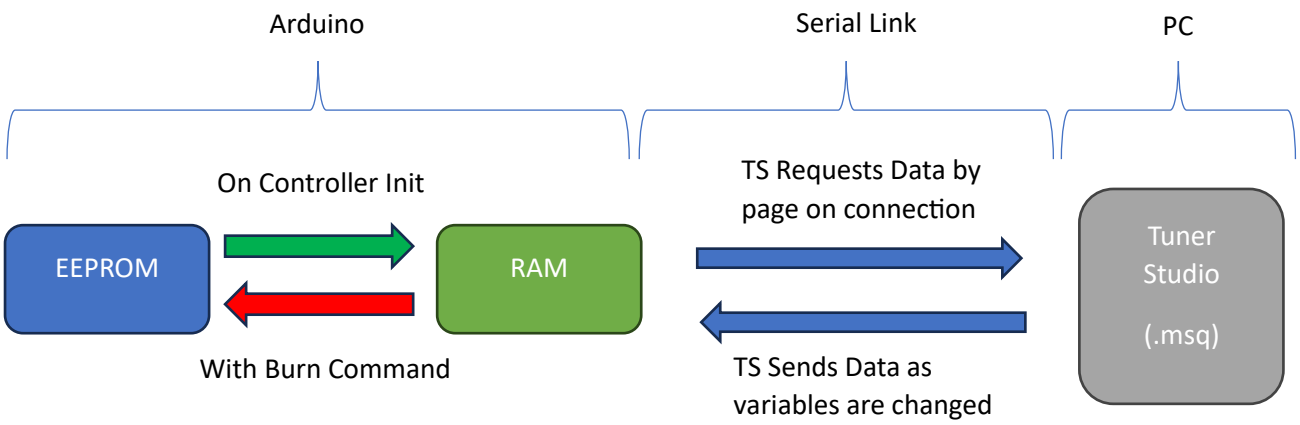

Figure 8 Diagram of Calibration Parameter Transfer

It's important to note that the EEPROM write can take 3.6ms per byte, so 8Kb can take up to 30 seconds. If the communications with TS are interrupted TS will reset the Arduino in an attempt to reconnect. So, the EEPROM updates must be performed asynchronously. Also, Tuner Studio breaks the EEPROM into pages to make it more manageable. Pages can be any size, but larger pages take longer to synchronize and send data.

If you need to re-define the page sizing start with the storage.h file. Remember the actual page size should never be greater than the defined page size or the variables will overwrite each other and TS will throw errors.

Variables in EEPROM can be single bits, scalars, or multidimensional arrays.

To define a new calibration variable search for the correct page definition in globals.h.

Add your variable with the correct data type. Note that if using a variable following some bits the bits are always part of a whole byte. The next scalar or array variable will be the nextOffset.

Once defined you can reference your new variable in the code with configPageX. new_variable_name.

```
//------------------------------------------------------------------------------
//Page 1 of the config - See the ini file for further reference
struct __attribute__ ( ( packed ) ) config1
{
  uint16_t page1ActualSize;  //TS READ ONLY: to check page size in development. This should never exceed the defined page sizes.
  uint32_t page1CRC;         //TS READ ONLY: Future expansion EEPROM CRC for error checking.
  uint8_t ochBlockSizeSent;  //TS READ ONLY: to check the serial data size for development. Match this with "ochBlockSize" parameter in the TS .ini file

  // 8 Bits in a byte for CAN config
  uint8_t can0Enable: 1;                //bit flags for canmodule configuration
  uint8_t can0Baud: 4;             // CAN0 Baud Rate defined in MCP_CAN library v1.5 mcp_can_dfs.h
  uint8_t can0XTalFreq: 2;          // MCP2515 board chip frequency defined in MCP_CAN library v1.5 mcp_can_dfs.h
  uint8_t can0Unusedbits: 1;
  uint8_t can0RXIntPin: 6;
  uint8_t unused1_8_bits: 2;

  uint8_t analogSelectorPin: 4;
  uint8_t analogSelectorEn: 1;
  uint8_t allowHWTestMode: 1;  // EEPROM based lockout of the hardware test mode. Prevents inadvertent serial data from accidently enabling this mode.
  uint8_t allowEEPROMClear: 1;  // EEPROM based lockout of the EEPROM wipe function. Prevents inadvertent serial data from accidently enabling this mode.
  uint8_t unused1_9_bits: 1;

  uint16_t canRXmsg_MotecPLM; // can Address for motec PLM message.


//#if defined(CORE_AVR)
};
```

Figure 9 Example of Definition of Page 1 in code.

In the TunerStudioOS.ini file find the page = X section. With X corresponding to the page you added your variable. In the same way as [OutputChannels] section you must add your variable in the exact same order and with the exact same units.

nextOffset and lastOffset also work here as they do in OutputChannels.

```
; PAGES of EEPROM are defined below. The definition must match exactly what is defined in the Arduino code for each page
; Reccomended practice is to keep the variable names the same. Use nextOffset and lastOffset, then comparing the variable list

 ;Page 1 is main settings. 128 bytes
page = 1
  ;  name                 = bits,   type,    offset, bits  ;comments
  ;  name                 = array,  type,    offset, shape, units,     scale, translate,    lo,      hi,       digits   ;comme
  ;  name                 = scalar, type,    offset,        units,     scale, translate,    lo,      hi,       digits   ;comme

  page1ActualSize        = scalar,   U16,    0,             "",        1.0,   0.0,          0.0,     65535.00,     0  ;TS READ
  page1CRC               = scalar,   U32,    2,             "",        1.0,   0.0,          0.0,     65535.00,     0  ;TS READ
  ochBlockSizeSent       = scalar,   U08,    6,  "",   1.00000,   0.00000,   0.00,    255.00,     0
  can0enable             = bits,   U08,    7,     [0:0],   "Disable","Enable"
  can0Baud               = bits,   U08,    7,     [1:4],   $CAN_Baud_list
  can0XTalFreq           = bits,   U08,    7,     [5:6],   $CAN_Freq_list
  unused1_7_bits         = bits,   U08,    7,     [7:7]
  can0RXIntPin           = bits,   U08,    8,     [0:5],   $Digital_Pins_List
  unused1_8_bits         = bits,   U08,    8,     [6:7]

  analogSelectorPin      = bits,   U08,    9,     [0:3],   $Analog_Pins_List
  analogSelectorEn       = bits,   U08,    9,     [4:4],   "Disable","Enable"

  allowHWTestMode        = bits,   U08,    9,     [5:5],   "Disable","Enable"
  allowEEPROMClear       = bits,   U08,    9,     [6:6],   "Disable","Enable"
  unused1_9_bits         = bits,   U08,    9,     [7:7]

  canRXmsg_MotecPLM      = bits,   U16,    10,    [0:10]   $CAN_ADDRESS_HEX
```

Figure 10 Example of Definition of Page 1 in TunerStudio.ini

3 Function Documentation

The code provides a number of functions to help with common control tasks and measurement. These can be removed if not required.

3.1 Task Execution and warning bits

3.1.1 Task Timer

The code implements a simple task scheduler. Every 1ms an interrupt timer is executed on timer 2 (timers.ino). This interrupt timer sets flags which are then checked in the main loop. This keeps the interrupt processing overhead down to a minimum.

Don't put extra code in the Timer.ino section as this will slow down the interrupt. Your user code should be called from the FUNC_XXXmsTask() schedulers.

3.1.2 Warning Bits and Dealing with Overflow

Make sure your code doesn't take too long to execute. Any complete loop can't be longer than 5ms otherwise the 5ms task will miss an execution step. So just because there is a 100ms task, does not mean that task can take 100ms. On a single core processor like the AVR, all tasks must run sequentially. If you have a very computational intensive process (like updating a display) then perhaps consider.

1. Updating the display over a few loops of the task

2. Moving your code to slower tasks to prevent overflow. (you may not need to run everything at 5ms). Then you can ignore the faster tasks overflowing if you don't have any code in them.

To help development each task is checked that it actually completed before it is set to run again. If it has not completed a flag will be set in TIMR_LoopDlyWarnBits. The TIMR_LoopDlyWarnBits variable is reset every second to help with visualization and logging in Tuner Studio. If a flag is set then you know it overflowed at least once in the last second. The TIMR_LoopDlyWarnBits are visible in Tuner Studio as indicators.

Also the loop timers are not all called at the same interval. You will notice how the timers remainder are compared to ==0 or ==1 etc. This means that the 20ms loop timer will attempt to start 1ms AFTER the 5ms loop starts. This helps to maintain accurate loop timing by not attempting to run all the tasks sequentially on the same 1ms loop. As a counter example, if we didn't have this feature whatever was in the 5ms and 20ms and 50ms task would delay the start of 100ms task, but only when it all the timers line up. This would show up as jitter if for example the 100ms task was switching outputs on and off.

```
//200Hz loop, 5ms
if ((loop1ms % 5) == 0)
{
  if (BIT_CHECK(TIMR_LoopTmrsBits, BIT_TIMER_5MS)) { BIT_SET(TIMR_LoopDlyWarnBits, BIT_TIMER_5MS); } // The lo
  BIT_SET(TIMR_LoopTmrsBits, BIT_TIMER_5MS);
}

//50Hz loop, 20ms
if ((loop1ms % 20) == 1)
{
  if (BIT_CHECK(TIMR_LoopTmrsBits, BIT_TIMER_20MS)) { BIT_SET(TIMR_LoopDlyWarnBits, BIT_TIMER_20MS); } // The
  BIT_SET(TIMR_LoopTmrsBits, BIT_TIMER_20MS);
}
```

Problems with overflow:

If one of those flags gets set the code will still execute however if you are using a software timer i.e. myUserTimer1_100ms incrementing in the 100ms loop and it overflows then the value of that timer will be incorrect (longer). The severity of this will depend on your application, but at least the code will show you that it happened and a default action could be written to mitigate the problem.

Another common problem with is that control functions that rely on regular execution such as PID feedback loops may go unstable.

3.1.3 Master Caution

Watching and datalogging all the various warning bits can be tedious. So, there is the implementation of a "Master Caution" or warning bit in the software. This just groups together all the warning flags and gives an overall status of the system. If you see it change in Tuner Studio you know there is a problem in some area that requires further investigation.

You can add any additional warning signals into this flag

The code is in the function SYS_setWarningBit(); and is called once every 100ms.

```
if ((TIMR_LoopDlyWarnBits > 0) || // any of the loop timer overflow warnings are set
    (Out_TS.Vars.UTIL_freeRam < 500) || // running out of RAM
    (Out_TS.Vars.loopsPerSecond < 200)) // slow loops
{
  BIT_SET(Out_TS.Vars.systembits, BIT_SYSTEM_WARNING); //set system warning flag
  Ve_t_WarningTimeoutTmr_100ms = 0;
}
```

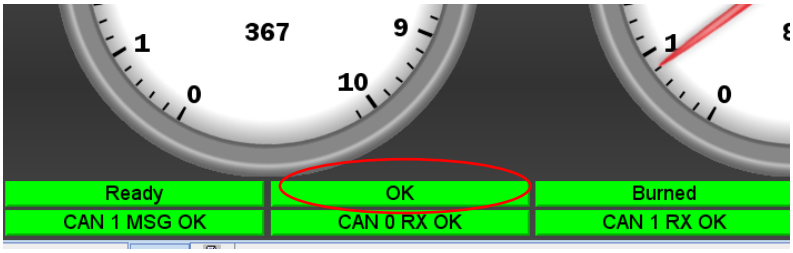

Figure 11. Location of Warning Bit in Tuner Studio

3.2 Low Pass Filter

uint16_t lowPassFilter_u16(uint32_t input, uint8_t alpha, uint32_t prior);

The function implements a low pass filter or lag filter using the new input, the previous input value and an "alpha value" from 1-255 which sets the filtering on the output. High values of alpha increase filtering with 128 being 50% filtering. The code ensures that the output will converge to the input even with very high filtering, the heaviest filter will have the output move closer to the input at a rate of one bit per loop.

The equation is:

Output = Prior * (1-alpha) + input * alpha

An example of the filtering is shown below.

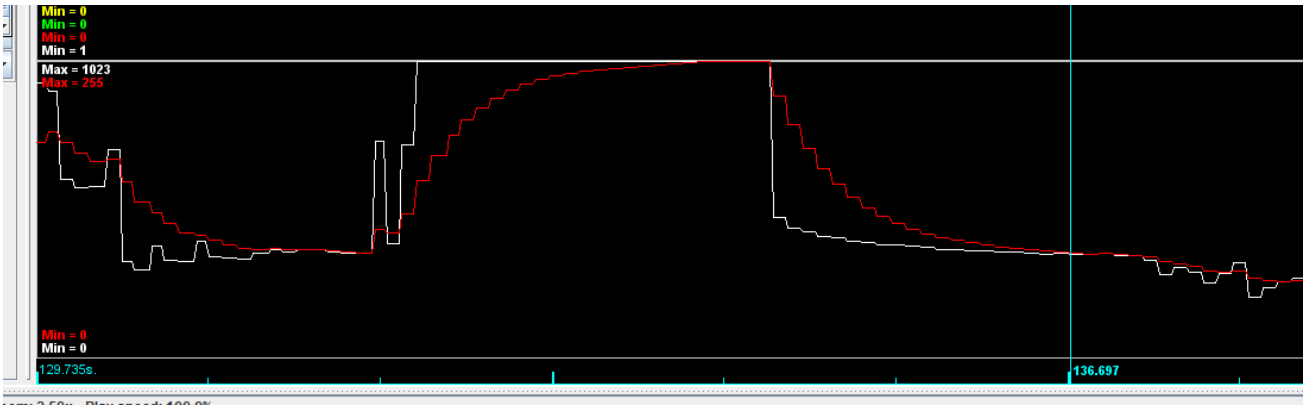

Figure 12 Example of Low Pass filter. Red signal is filtered, white is the input. Alpha of approximately 200

3.3 Table Interpolation

Tuner Studio supports defining calibration variables as tables (or maps). These can describe a 2 dimensional surface (single lookup) or 3 dimensional surface (dual lookup).

3.3.1 2D Lookup

A 2D lookup function is very powerful. Some usages are:

1. Scaling a non-linear input (such as a ADC count) into another unit such as temperature.
2. Mapping control parameters.
3. Storing calculated math routines.

To define a 2D lookup you need an array X axis parameters and Y value parameters. It's worth pointing out that both do not need to be saved as calibration parameters. To save EEPROM the x axis for example may be

fixed in code. The axis and values do have to be the same length. Also the X axis values must be all increasing or all decreasing (equal values are ok, but not very useful). Y values can be any value.

To define a new table. Create the x axis and y axis as arrays of variables. There is no special structure and this can be helpful as it allows the user to re-use the X-axis for example with different Y Values. Then call the lookup function.

Example:

```
Out_TS.Vars.dev4 = u8_table2DLookup_u8(configPage2.exampleTable_Xaxis, - The X axis array
                                       configPage2.exampleTable_Ydata, - The Y data array
                                       sizeof(configPage2.exampleTable_Xaxis), - Number of array bins.
                                       configPage2.exampleLookupValue); - The X lookup value
```

Currently there are routines for unsigned 8 and 16bit integer math. But other data types can easily be supported including float lookup by adding the appropriate functions.

Currently supported functions:

- u16_table2DLookup_u16 – 16 bit unsigned output with 16 bit unsigned input variables.
- u8_table2DLookup_u8 – 8 bit unsigned output with 8 bit unsigned input variables.

If you need a signed lookup, its more efficient to use an offset on the signal (so for example 127 becomes "0") when you define the measure variable in Tuner Studio you can specify the same offset so the variable is correctly displayed as a negative number when less than 127.

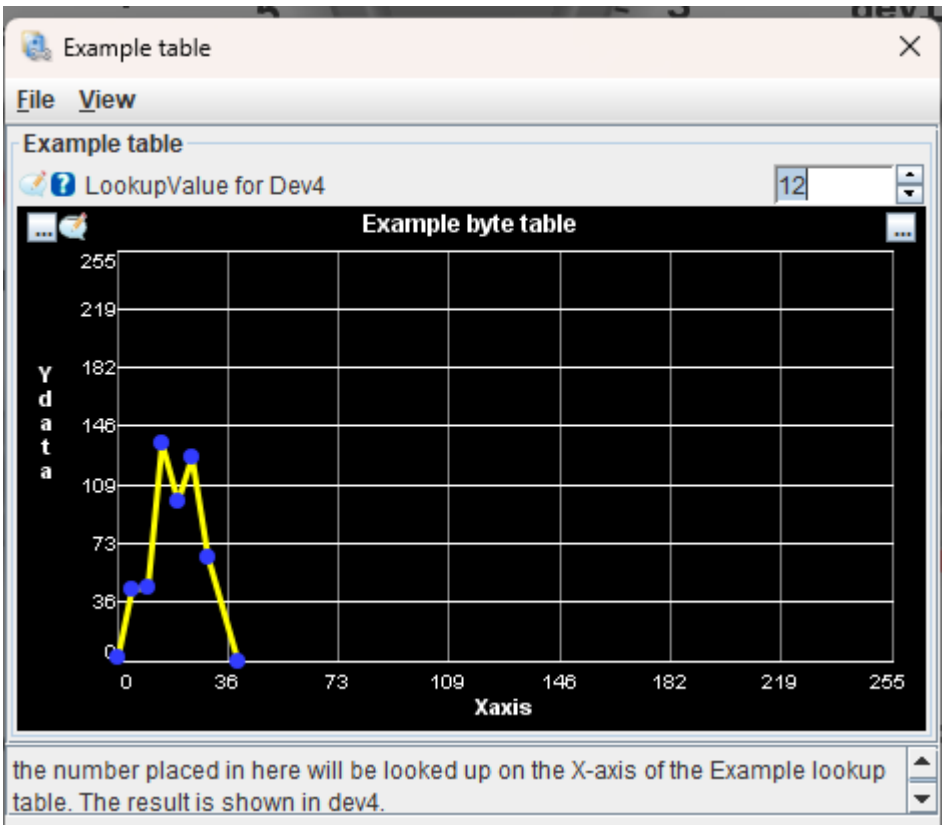

Figure 13. Example of a 2D lookup table in Tuner Studio

In the TunerStudioOS.ini file you need to define the table in the [CurveEditor] section.

Example:

```
; ExampleTable curve
  curve = ExampleTableCurve, "Example byte table"
    columnLabel = "Xaxis", "Ydata"
    xAxis       = 0, 255, 8
    yAxis       = 0, 255, 8
    xBins       = exampleTable_Xaxis
    yBins       = exampleTable_Ydata
```

The Xaxis and Ydata also needs to be defined as an array on the corresponding pages. If the X axis is fixed you need to create a PC variable with the same values as the code.

3.3.2 3D Lookup

To be done…

3.4 CAN

3.4.1 CAN Overview

A simple implementation of a Controller Aera Network (CAN) Broadcast network is provided in the OS. In this network, each node broadcasts messages onto the Bus at a specific rate. The sender has no knowledge of which modules are receiving the messages.

Every module receives all messages and compares them against a number of pre-defined receive message parsers. The parser decodes the message and loads the data from the message into variables for use internally.

This sort of messaging is typically used in vehicles to transmit data to dashboard displays or to the Transmission controller.

It is important in this system to detect if an expected message is no longer being received and execute a default “safe” state. Otherwise, the last received value will be held forever.

All CAN messages consist of:

- “ID” – 11Bits (001h to 800h) that defines the message address. When BUS load is high lower numbers have higher priority than higher numbers. There is also 29bit messaging id’s possible but this is not implemented in this ‘light’ CAN bus implementation.
- “Data Length” – A variable specifying the length of the expected message payload.
- “Data” – 1 to 8 bytes as the payload for the message.

3.4.2 Implementation

The code is expecting a MCP2515 controller connected via the SPI interface. The pinout is different depending on the board. Most of the pins and initialisation is configurable via the settings in Tuner Studio. The actual CAN messages are hardcoded in canbus.ino

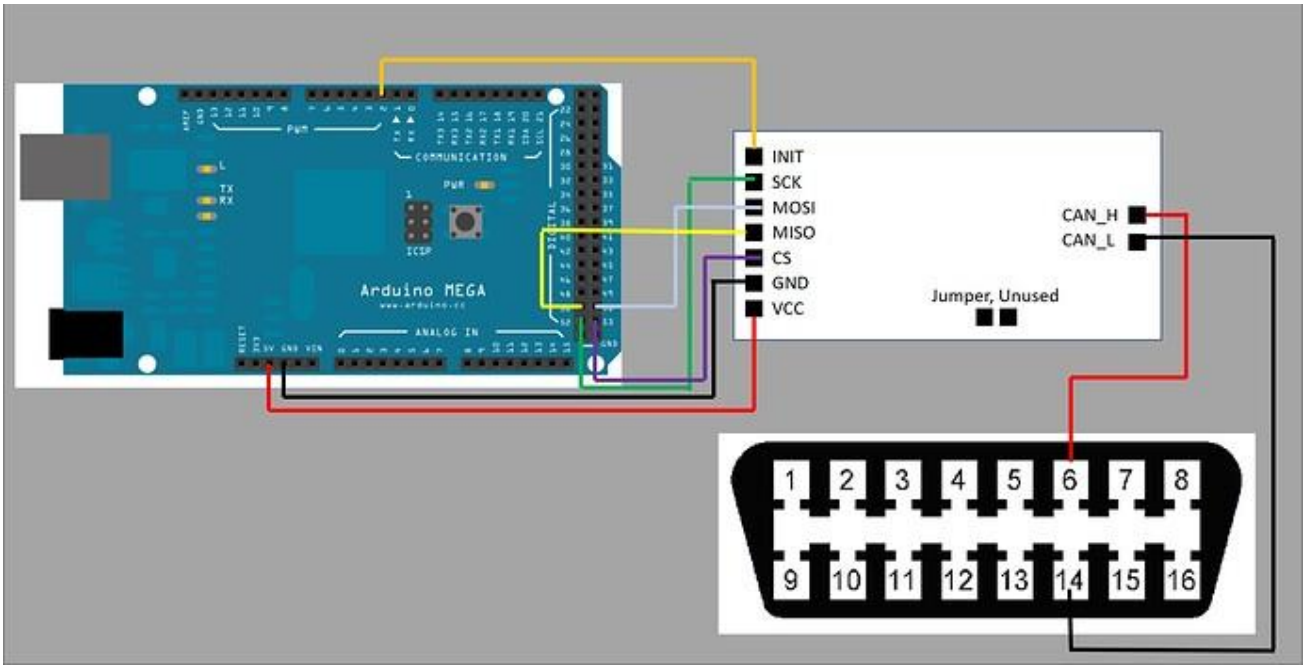

Figure 14 Example of MCP2515 connection on Arduino Mega

If CAN is enabled. The Arduino attempts to initialise the module and begins broadcasting if successful. If not successful, the code will send status bits to Tuner Studio which can be visualized at the bottom of the screen.

If you are having CAN bus errors. Check the following:

1. Are all nodes on the network configured to the same frequency?
2. Do you have appropriate termination resistors? You should have 120ohms at each end of the bus. Most MCP2515 boards have the provision to enable a termination resistor.

3.4.3 CAN Broadcast

CAN broadcast uses simple functions called from the main task scheduler.

Example: void canBroadcast_5ms(void)

The message variables can then be parsed into the canmsg[] object and sent at the appropriate rate.

3.4.4 CAN Receive message

CAN receive uses the interrupt pin from the MCP2515 to tell the Arduino that CAN data is waiting. As of 0.3 implementation it does not actually interrupt the code to receive the message, this pin is polled from the main loop.

If the user does implement this as an interrupt don't forget that variables need to be declared volatile when used in ISR's, also that heavy CAN bus traffic can cause the controller to become overloaded.

Once a message is received, the ID of the message is checked against each defined message object. Then the length is compared in the object to make sure the full message was received. If both of these checks pass the message is parsed and the timeout counter reset.

If you are attempting to connect the CAN to an existing vehicle CAN remember that the vehicle may send many messages that can overwhelm the small processor on the Arduino. It is possible on the MCP2515 to

configure a message window to filter in hardware for only the CANID's of the messages you are interested in. Check the datasheet on how to do this.