

# PIC programming for CBUS using CBUSlib

Ian Hogg M5144

December 2021 – relates to CBUSlib version 2m

## Table of Contents

1	Introduction.....	5
2	Development Environment.....	6
2.1	Computer.....	6
2.2	MPLAB X.....	6
2.3	C Compiler.....	7
2.4	PICkit.....	7
2.5	Programming Cable.....	8
2.6	Datasheet.....	8
2.7	CBUSlib.....	8
3	Program Structure.....	9
3.1	Waits and delays.....	9
3.2	Interrupts.....	9
3.3	Source code Files.....	10
4	First Program – LED flasher.....	11
4.1	Getting CBUSlib.....	11
4.2	Set up MPLAB X.....	12
4.3	Project Source files.....	13
4.3.1	hwsettings.h.....	13
4.3.2	hwsettings.c.....	14
4.3.3	Main.c.....	15
4.4	Compile.....	16
4.5	Simulate.....	17
4.6	Test.....	19
4.7	Hardware Debugging.....	19
5	Using more of CBUSlib.....	22
5.1	#defines used by CBUSlib to enable functionality.....	22
5.1.1	AREQ_SUPPORT.....	22
5.1.2	BOOTLOADER_PRESENT.....	22
5.1.3	CBUS_OVER_CAN.....	22
5.1.4	CBUS_OVER_MIWI.....	22
5.1.5	CBUS_OVER_TCP.....	22
5.1.6	DEBUG_PRODUCED_EVENTS.....	22
5.1.7	HASH_TABLE.....	22
5.1.8	NV_CACHE.....	23
5.1.9	PRODUCED_EVENTS.....	23
5.1.10	SAFETY.....	23
5.1.11	TEST_MODE.....	23
5.2	#defines used by CBUSlib to specify information.....	23
5.2.1	ACTION_T.....	23

5.2.2	AT_EVENTS.....	23
5.2.3	AT_NAME_ADDRESS.....	23
5.2.4	AT_NV.....	23
5.2.5	AT_PARAM_ADDRESS.....	23
5.2.6	CAN_INTERRUPT_PRIORITY.....	23
5.2.7	CHAIN_LENGTH.....	24
5.2.8	EVENT_TABLE_WIDTH.....	24
5.2.9	EvperEVT.....	24
5.2.10	FLiM_SW.....	24
5.2.11	HAPPENING_BASE.....	24
5.2.12	HAPPENING_T.....	24
5.2.13	HASH_LENGTH.....	24
5.2.14	LED1Y.....	24
5.2.15	LED2G.....	24
5.2.16	LED_OFF.....	24
5.2.17	LED_ON.....	24
5.2.18	LOAD_ADDRESS.....	25
5.2.19	MAJOR_VER.....	25
5.2.20	MANU_ID.....	25
5.2.21	MAX_WRITEABLE_FLASH.....	25
5.2.22	MIN_WRITEABLE_FLASH.....	25
5.2.23	MINOR_VER.....	25
5.2.24	MODULE_ID.....	25
5.2.25	MODULE_TYPE.....	25
5.2.26	NUM_EVENTS.....	25
5.2.27	NUM_HAPPENINGS.....	25
5.2.28	NV_NUM.....	25
5.2.29	TRIS_LED_1Y.....	25
5.2.30	TRIS_LED_2G.....	25
5.3	Variables and defines provided by CBUSlib.....	25
5.3.1	EE_APPLICATION.....	25
5.3.2	flimState.....	26
5.4	CAN/CBUS/FLiM Functionality.....	26
5.4.1	Initialisation.....	26
5.4.2	Main loop.....	27
5.4.3	Callbacks to be provided by the Application.....	28
5.5	Event Handling Functionality.....	29
5.5.1	Callbacks.....	29
5.6	Action Queue Functionality.....	29
5.7	EEPROM and Flash Functionality.....	29
5.7.1	Initialisation.....	29
5.7.2	Bounds checking.....	30
5.7.3	Data versions.....	30
5.8	Bootloader Functionality.....	30
5.9	Oscillator settings.....	30
5.10	Global Variable initialisation.....	30
5.10.1	Analogue disable.....	31
5.11	Memory Map.....	31
5.11.1	Flash Memory.....	32
5.11.2	EEPROM.....	33

5.11.3 Linker.....	34
6 Using CBUSlib.....	35
6.1 CBUS CAN processing.....	35
6.1.1 Receiving CBUS messages.....	35
6.1.2 Handle the CBUS messages.....	35
6.2 TickTime.....	35
6.2.1 Function void initTicker(unsigned char priority).....	35
6.2.2 Typedef TickValue.....	36
6.2.3 Function DWORD tickGet(void).....	36
6.2.4 Macro DWORD tickTimeSince(DWORD t).....	36
6.2.5 Macro Time constants.....	36
6.2.6 Example.....	36
6.3 Module status.....	36
6.3.1 Push Button.....	37
6.4 Status LEDs.....	37
6.4.1 Initialise.....	37
6.4.2 Main loop call.....	37
6.4.3 Flicker.....	37
6.5 NV operations.....	38
6.5.1 Validating NV changes.....	38
6.5.2 Acting upon an NV change.....	38
6.5.3 Accessing NVs.....	38
6.6 NV_CACHE.....	39
6.6.1 Initialisation.....	39
6.6.2 Updating Cache contents.....	39
6.6.3 Accessing NVs.....	39
6.7 Event Handling.....	39
6.7.1 Receiving events.....	39
6.7.2 Getting EVs.....	40
6.7.3 Getting the number of EVs.....	40
6.7.4 Adding an Event.....	40
6.7.5 Removing an event.....	41
6.7.6 Action queue.....	41
6.8 Sending events and CBUS messages.....	41
6.9 ActionQueue.....	41
6.9.1 Initialisation.....	41
6.9.2 Push.....	42
6.9.3 Pop.....	42
6.9.4 Peek.....	42
6.9.5 Queue size.....	42
6.10 RomOps.....	42
6.10.1 Initialisation.....	42
6.10.2 Reading EEPROM.....	42
6.10.3 Writing EEPROM.....	43
6.10.4 Reading Flash.....	43
6.10.5 Writing Flash.....	43
6.10.6 Flush Flash copy.....	43
6.10.7 Obtaining the CPU type.....	43
6.11 Handling other CBUS opcodes.....	44
6.11.1 Special handling Pre CBUSlib.....	44

6.11.2 Special handling Post CBUSlib.....	44
7 Typical Application.....	45

# 1 Introduction

This document acts as a mini tutorial to guide the reader through the process of developing software using a PIC for a CBUS module using CBUSlib.

The development environment is described and then the reader is taken through a simple LED flasher example. Although this is not a complete CBUS module it provides a good starting point and helps the reader gain confidence in using the PIC tools.

The CBUSlib is described in more detail to allow the reader to develop a complete CBUS module.

CBUSlib can certainly be improved, made more adaptable and easier to use. Please help by participating in its development. You are welcome you submit Pull Requests using Github.

Things about module params that need improvement within CBUSlib

Things about NV cache which currently isn't part of CBUSlib

Things about Action Queue which isn't currently part of CBUSlib

## 2 Development Environment

To develop a CBUS module you will need a development environment

### 2.1 Computer

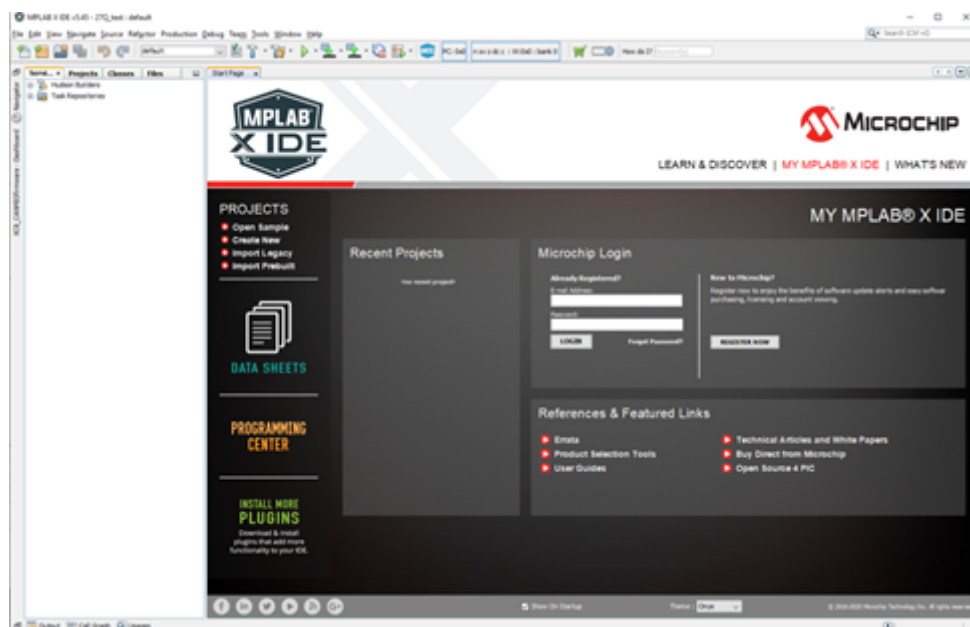
- You'll need a computer on which to develop your software.
- Needs to support the development tools. The more powerful computer you have the quicker it will be able to compile and so the time to loop through making changes and compiling will be quicker and an overall less frustrating experience. That being said any computer produced within the last 5 years would be adequate.
- Can be Windows, Linux or Mac.
- Requires 1 USB port although a second may be useful later but a USB hub can be used to convert a single port into many.

### 2.2 MPLAB X

MPLAB X is an Integrated Development Environment (IDE). This is a tool which allows for editing, compiling and debugging of programs you are developing. All the tools are integrated together to provide smother progression through the workflow of Edit, Compiler, Test.

MPLAB X is available free from Microchip. It is designed specifically to work with Microchip tools for the development of PIC and AVR programs.

A Project is used to contain all the files, data and configuration for a program under development.



## 2.3 C Compiler

In this tutorial we'll be using the C programming language so you'll need a C compiler.

A choice of two compilers are available free from Microchip.

- Use XC8 for new designs
- C18 may be needed to make changes to older code.

This tutorial assumes the use of XC8.

There are differences between the two compilers which mean that program source code is not 100% compatible. In particular there are differences in #pragma statements for memory usage and specifying memory addresses.

It is a good idea to put compiler specific code within #if .... #endif statements, for example CBUSlib uses

- #if defined(\_\_18CXX)
- #if defined(\_\_XC8\_\_)

## 2.4 PICKit

A PICKit is used to load programs onto blank PICs and for debugging of programs under development whilst running on real hardware.

The PICKit connects to the computer using a USB port. It also connects to target PIC board using a in-circuit serial programming (ICSP) connector which is a 1 x 6way 0.1" header.

PICKits may be purchased from Farnell, RS. Also available from Amazon, Ebay etc. but some may be clones. As of September 2021 Farnell are charging £30 for a PICKit 3 and £62 for a PICKit 4. On Ebay a PICKit 3 is available for £12.

The latest version of PICKit is the PICKit 4 which supports the latest PIC and AVR devices. For this tutorial we'll be using a PIC18F25K80 device which is also supported by the PICKit 3.



## 2.5 Programming Cable

MERG boards use a different pinout for the ICSP connector from that of the standard Microchip connector therefore a special cable is used to convert from PICKit to the MERG format.

See [https://merg.org.uk/merg\\_wiki/doku.php?id=helpsystem:pickit2cbus](https://merg.org.uk/merg_wiki/doku.php?id=helpsystem:pickit2cbus) for details of making up a cable or connector.

Since the ICSP connector is unpolarised I also paint one end of the connector with white paint to ensure it is less likely that I get the PICKit plugged in the wrong way around. I also use red paint for a Microchip ICSP connector.

## 2.6 Datasheet

You will need the datasheet for the PIC you are targeting.

You'll need the memory map, register names and to understand the peripherals supported and how to operate the peripherals.

It is best to have a read through everything once and then you can keep going back for specific information as and when required.

## 2.7 CBUSlib

You'll need to obtain a copy of CBUSlib from the MERG-DEV Github site.

CBUSlib is a source level library and not an object file library therefore you don't link the library or object files but instead add the required source files to your project in MPLAB X.

The library is available from MER-DEV Github <https://github.com/MERG-DEV/CBUSlib>

- Contains functions for:
  - CAN interface (can18.c/h)
  - CBUS messages (cbus.c/h)
  - CBUS push button and SLiM/FLiM LEDs (Flim.c/h)
  - Reading and writing to EEPROM and Flash memory (romops.c/h)
  - Main loop timer (ticktime.c/h)
  - Event teaching, learning (events.c/h)
  - Bootloader (Bootloader.asm)

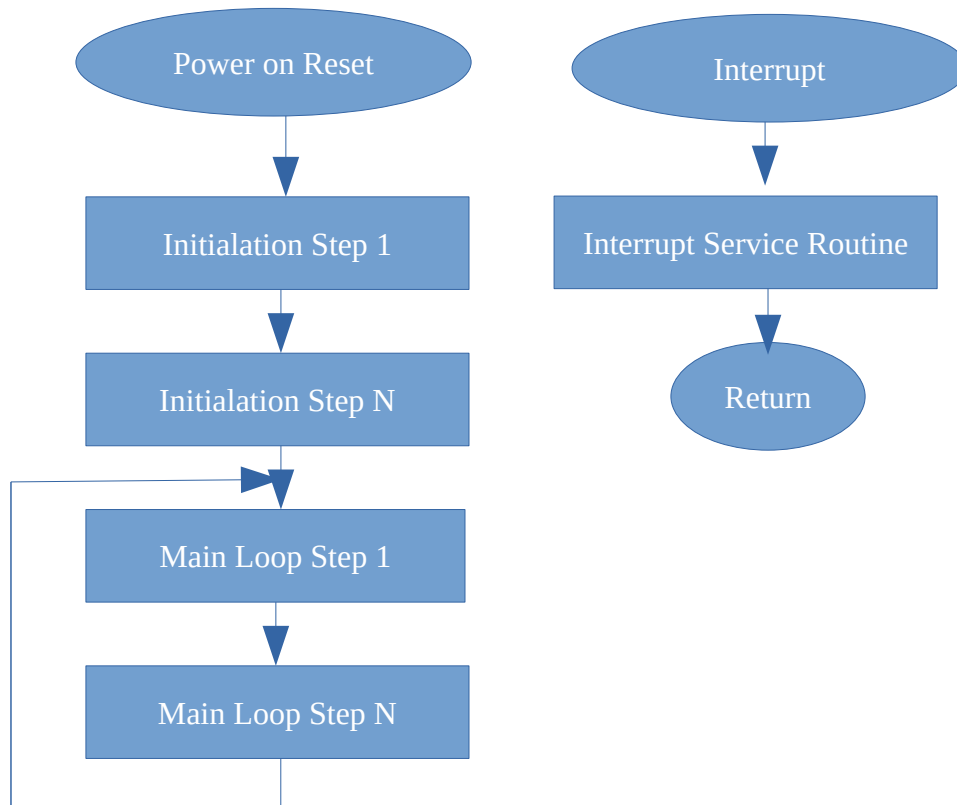
You will possibly/probably also need to get a copy of cbusdefs which is available from <https://github.com/MERG-DEV/cbusdefs>



### 3 Program Structure

It is beneficial to get an understanding of the overall structure of a module's program structure.

There isn't anything special about the PIC or CBUSlib and the program is structured as any other microprocessor system such as Arduino:



#### 3.1 Waits and delays

Each of the steps in the main loop should finish as quickly as possible so that other steps can be serviced and have their share of the processor. Any steps in the main loop should not perform blocking/waiting I/O nor perform idle wait/delay operation since that would delay servicing other steps.

To implement a delay, set a timer and resume when the timer has expired the next time the step is called. CBUSlib facilitates this process using the TickTime functionality, which we'll be using in this tutorial.

#### 3.2 Interrupts

Interrupts can be generated by hardware that needs to be serviced by software. The interrupt indicates that something has happened.

Generally interrupt service routines should return as quickly as possible. In the interrupt service routine set a flag to indicate that something is needed to be done in the main loop. Then one of the steps in the main loop should check the flag and perform the necessary processing.

PIC peripherals often support this flag in hardware without needing to use interrupts. (PIR registers) so it is therefore sometimes possible to check the hardware flag in the main loop without needing to use an interrupt.

Remember to clear the flag once the hardware has been serviced.

Interrupts can have different priorities. Choose the priorities carefully, it is possible that a peripheral handling a function of lower importance has the most time critical demands. For example a hardware timer used to time the length of a servo pulse must have a higher priority than that of the CAN peripheral since a delay to servicing the servo can cause the servo to misbehave and the CAN peripheral has its own buffering and can cope with a delay.

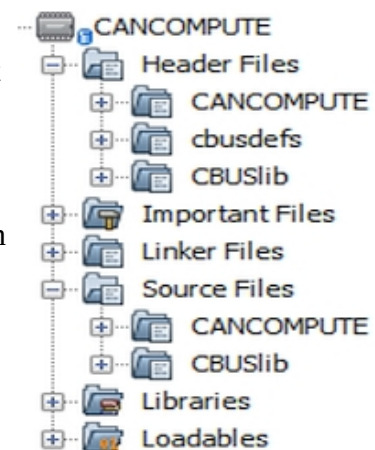
### 3.3 Source code Files

Organise your program so that related code is put together in a file (xxxx.c). Put common definitions associated with that code in the C header file (xxxx.h). Use multiple files with each piece of functionality in separate files.

Include the header file in other sources which need to know about the functionality.

Add the source and header files to the project in MPLAB X.

It is useful to keep the CBUSlib files and your own project files in separate directories. The diagram on the right shows the project structure used for CANCOMPUTE with CBUSlib, cbusdefs and CANCOMPUTE specific files in their own folders.



## 4 First Program – LED flasher

This section takes the reader through the steps needed to get a simple LED flashing program running on a module. We'll use the Green SLiM LED.

As we're doing this as a tutorial but do it "properly" using interrupts and no blocking waits. We need to consider:

- Getting the CBUSlib
- Setting up the project in MPLAB X
- Use of TickTime for timing the flashes
- Initialisation, setting everything up
- Using a digital output to the LED
- Main Loop to do the flashing

### 4.1 Getting CBUSlib

The simplest way of getting a copy of CBUSlib is to download a ZIP of the files.

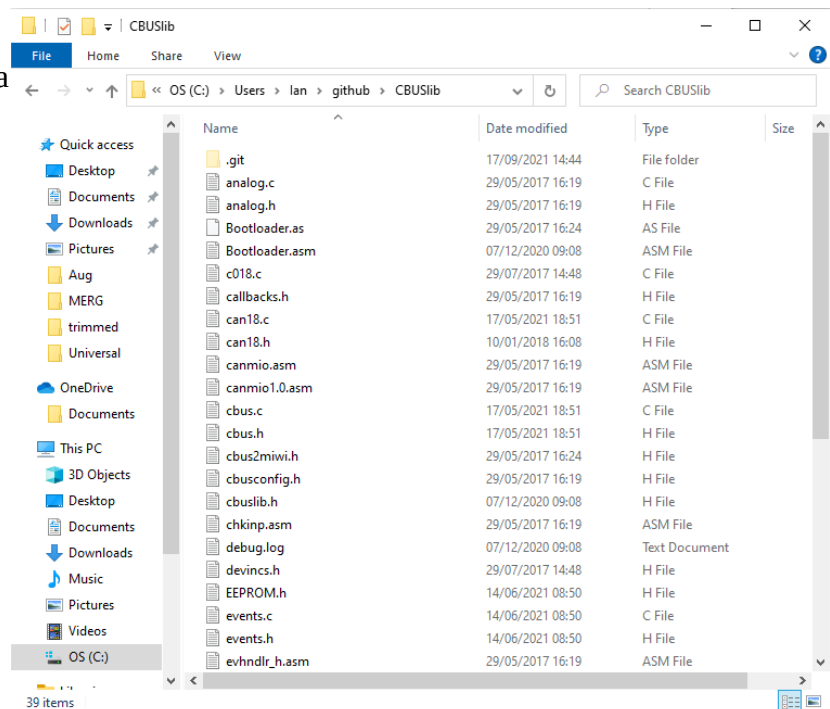
<https://github.com/MERG-DEV/CBUSlib/archive/refs/heads/master.zip> and then unpacking these into a local github folder.

It would be better to use the git utility to clone the repository from github. It should be cloned into a local folder and I suggest this is called github.

```
$ cd github
```

```
$ git clone https://github.com/MERG-DEV/CBUSlib.git
```

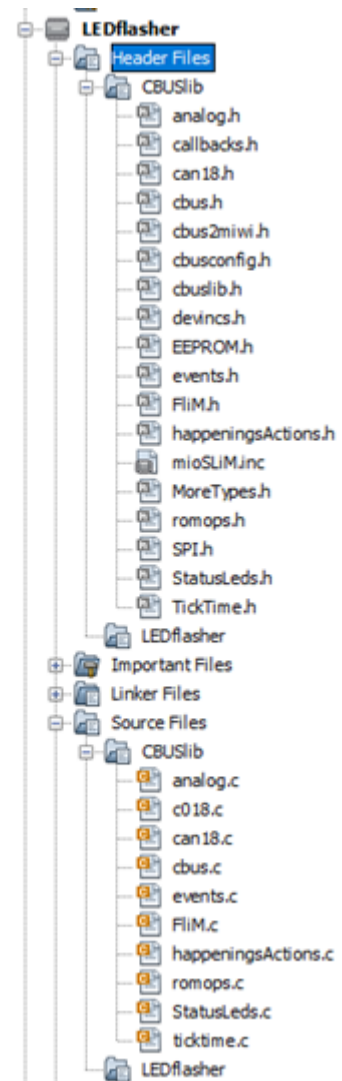
Alternatively it is possible to use a GUI Clone the CBUSlib repository from github into a folder in your local filesystem.



## 4.2 Set up MPLAB X

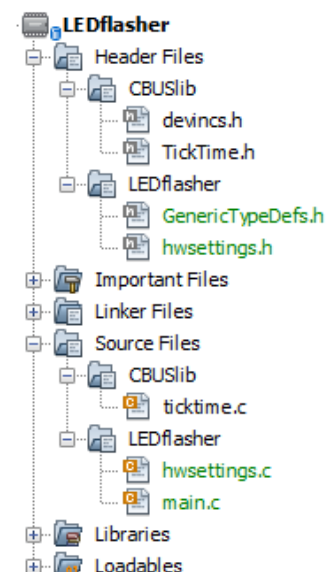
Now let's get the project for the LED flasher set up in MPLAB X and the necessary files imported.

1. Create a new folder for the LEDflasher project source files.
2. Using MPLAB X create a new project: Microchip Embedded, Standalone Project.
  - Family: Advanced 8-bit MCUs (PIC18)
  - Device: PIC18F25K80
  - Tool: PICkit if you have it plugged in otherwise "No Tool".
  - Compiler: XC8 (v2.32). The C18 compiler could also be used but isn't covered by this tutorial
  - Project Name: "LEDflasher".
3. Create a LEDflasher Logical Folder under Header Files and another under Source files
4. Select "Add Existing Items From Folders..." from the Headers folder, select the CBUSlib folder and the "Header Files" type of file.
5. Select "Add Existing Items From Folders..." from the Source folder, select the CBUSlib folder and the "C Source Files" type of file.



The above should result in all the CBUSlib files being added to the project as indicated in the picture on the right.

6. For LEDflasher we actually only need a few files. We need ticktime.c and ticktime.h we'll keep devincs.h too and we need GenericTypeDefs.h if using XC8. Remove the other unnecessary library files from the project by right clicking them and select "Remove from project". In this simple example it probably would have been quicker to just add the files we wanted but adding all is useful for future projects.
7. We also need to create the necessary hwsettings.h, hwsettings.c and main.c files for our LEDflasher application.
8. Make sure that you select the project source file folder when creating these files.



9. It is necessary to configure the project in MPLAB X in order to tell the compiler there are two folders of include files. This is done by editing the project properties so right click the LEDflasher project in the hierarchy and select “Properties”. Under XC8 Compiler add the two folders containing the header source files to the Include directories



### 4.3.1 hwsettings.h

Page 13

```
#define ei()      INTCONbits.GIEH = 1;INTCONbits.GIEL = 1
#define di()      INTCONbits.GIEH = 0;INTCONbits.GIEL = 0
#endif
```

```
// Global routine definitions
void setclkMHz( void );
```

### 4.3.2 hwsettings.c

```
include "hwsettings.h"
// PIC18F25K80 Configuration Bit Settings
// 'C' source line config statements
// CONFIG1L
#pragma config RETEN = OFF      // VREG Sleep Enable bit (Ultra low-power
regulator is Disabled(Controlled by REGSLP bit))
#pragma config INTOSCSEL = HIGH // LF-INTOSC Low-power Enable bit (LF-INTOSC
in High-power mode during sleep)
#pragma config SOSCSEL = DIG    // SOSC Power Selection and mode
Configuration bits (Digital (SCLKI) mode
#pragma config XINST = OFF      // Extended Instruction Set (Disabled)
// CONFIG1H
#pragma config FOSC = HS1       // Oscillator (HS oscillator (Medium power,
4 MHz - 16 MHz))
#pragma config PLLCFG = OFF     // PLL x4 Enable bit (Disabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor (Disabled)
#pragma config IESO = OFF       // Internal External Oscillator Switch Over
Mode (Disabled)
// CONFIG2L
#pragma config PWRTEN = ON      // Power Up Timer (Enabled)
#pragma config BOREN = SBORDIS  // Brown Out Detect (Disabled in
hardware, SBOREN disabled)
#pragma config BORV = 0         // Brown-out Reset Voltage bits (3.0V)
#pragma config BORPWR = ZPBORMV // BORMV Power level (ZPBORMV instead of
BORMV is selected)
// CONFIG2H
#pragma config WDTEN = OFF      // Watchdog Timer (WDT disabled in hardware;
SWDTEN bit disabled)
#pragma config WDTPS = 1048576  // Watchdog Postscaler (1:1048576)
// CONFIG3H
#pragma config CANMX = PORTB    // ECAN Mux bit (ECAN TX and RX pins are
located on RB2 and RB3, respectively)
#pragma config MSSPMSK = MSK7   // MSSP address masking (7 Bit address
masking mode)
#pragma config MCLRE = ON       // Master Clear Enable (MCLR Enabled, RE3
Disabled)
// CONFIG4L
#pragma config STVREN = ON      // Stack Overflow Reset (Enabled)
#pragma config BBSIZ = BB1K     // Boot Block Size (1K word Boot Block size)
// CONFIG5L
#pragma config CP0 = OFF        // Code Protect 00800-01FFF (Disabled)
#pragma config CP1 = OFF        // Code Protect 02000-03FFF (Disabled)
```

```

#pragma config CP2 = OFF           // Code Protect 04000-05FFF (Disabled)
#pragma config CP3 = OFF           // Code Protect 06000-07FFF (Disabled)
// CONFIG5H
#pragma config CPB = OFF           // Code Protect Boot (Disabled)
#pragma config CPD = OFF           // Data EE Read Protect (Disabled)
// CONFIG6L
#pragma config WRT0 = OFF           // Table Write Protect 00800-01FFF (Disabled)
#pragma config WRT1 = OFF           // Table Write Protect 02000-03FFF (Disabled)
#pragma config WRT2 = OFF           // Table Write Protect 04000-05FFF (Disabled)
#pragma config WRT3 = OFF           // Table Write Protect 06000-07FFF (Disabled)
// CONFIG6H
#pragma config WRTC = OFF           // Config. Write Protect (Disabled)
#pragma config WRTB = OFF           // Table Write Protect Boot (Disabled)
#pragma config WRTD = OFF           // Data EE Write Protect (Disabled)
// CONFIG7L
#pragma config EBTR0 = OFF           // Table Read Protect 00800-01FFF (Disabled)
#pragma config EBTR1 = OFF           // Table Read Protect 02000-03FFF (Disabled)
#pragma config EBTR2 = OFF           // Table Read Protect 04000-05FFF (Disabled)
#pragma config EBTR3 = OFF           // Table Read Protect 06000-07FFF (Disabled)
// CONFIG7H
#pragma config EBTRB = OFF           // Table Read Protect Boot (Disabled)

```

```

BYTE clkMHz;           // Derived or set system clock frequency in MHz

```

```

// Set system clock frequency variable - either derived from CAN baud rate
register
// if set by bootloader, or default value for CPU type if not

```

```

void setclkMHz( void )
{
    clkMHz = 16;
}

```

### 4.3.3 Main.c

```

#include <xc.h>
#include "TickTime.h"
#include "hwsettings.h"

#define FLASH_TIME HALF_SECOND

void main(void) {
    unsigned char currentState = LED_OFF;
    TickValue flashTime;

    // set up the IO port
    // make it an output

```

```

TRIS_LED2G = 0;
// Initialise the timer
initTicker(0); // low priority ticker

flashTime.Val = tickGet();

ei(); // initialisation complete, enable interrupts
while (1) {
    if (tickTimeSince(flashTime) > FLASH_TIME) {
        // change state
        currentState = currentState == LED_OFF ? LED_ON : LED_OFF;
        LED2G = currentState;
        flashTime.Val = tickGet();
    }
}
return;
}

// Interrupt service routines
void __interrupt(low_priority) low_isr(void) {
    tickISR();
}

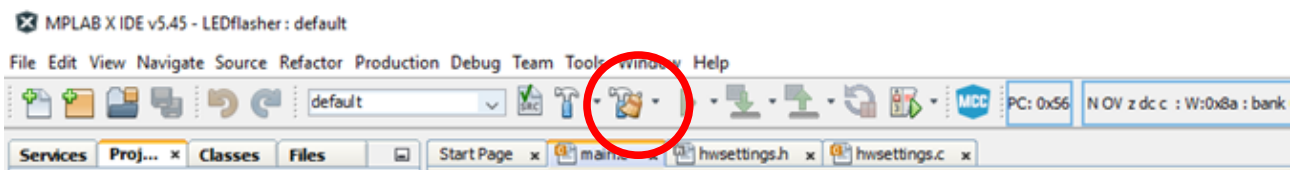
void __interrupt(high_priority) high_isr (void) {
}

```

## 4.4 Compile

Once all the project source files have been created and the CBUSlib files included within the project it is time to attempt to compile the project.

Clean-compile the project using the “Clean and Build Main Project” hammer and brush toolbar button.



The MPLAB X console will show the compile progress and take a few seconds.

Check and fix any errors so that you get a successful compilation as shown below.



```

CLEAN SUCCESSFUL (total time: 26ms)
make -f nbproject/Makefile-default.mk SUBPROJECTS= .build-conf
make[1]: Entering directory 'C:/Users/Ian/MPLABXProjects/LEDflasher.X'
make -f nbproject/Makefile-default.mk dist/default/production/LEDflasher.X.production.hex
make[2]: Entering directory 'C:/Users/Ian/MPLABXProjects/LEDflasher.X'
"C:\Program Files\Microchip\xc8\v2.32\bin\xc8-cc.exe" -mcpu=18F25K80 -c -mdfp="C:/Program Files/Microchip/MPLAB
"C:\Program Files\Microchip\xc8\v2.32\bin\xc8-cc.exe" -mcpu=18F25K80 -c -mdfp="C:/Program Files/Microchip/MPLAB
"C:\Program Files\Microchip\xc8\v2.32\bin\xc8-cc.exe" -mcpu=18F25K80 -c -mdfp="C:/Program Files/Microchip/MPLAB
"C:\Program Files\Microchip\xc8\v2.32\bin\xc8-cc.exe" -mcpu=18F25K80 -Wl,-Map=dist/default/production/LEDflasher.

Memory Summary:
Program space      used 17Ah ( 378) of 8000h bytes ( 1.2%)
Data space        used 18h ( 24) of E41h bytes ( 0.7%)
Configuration bits used 7h ( 7) of 7h words (100.0%)
EEPROM space      used 0h ( 0) of 400h bytes ( 0.0%)
ID Location space  used 8h ( 8) of 8h bytes (100.0%)

make[2]: Leaving directory 'C:/Users/Ian/MPLABXProjects/LEDflasher.X'
make[1]: Leaving directory 'C:/Users/Ian/MPLABXProjects/LEDflasher.X'

BUILD SUCCESSFUL (total time: 3s)
Loading code from C:/Users/Ian/MPLABXProjects/LEDflasher.X/dist/default/production/LEDflasher.X.production.hex...
Program loaded with pack, PIC18F-K_DFP, 1.4.87, Microchip
Loading completed

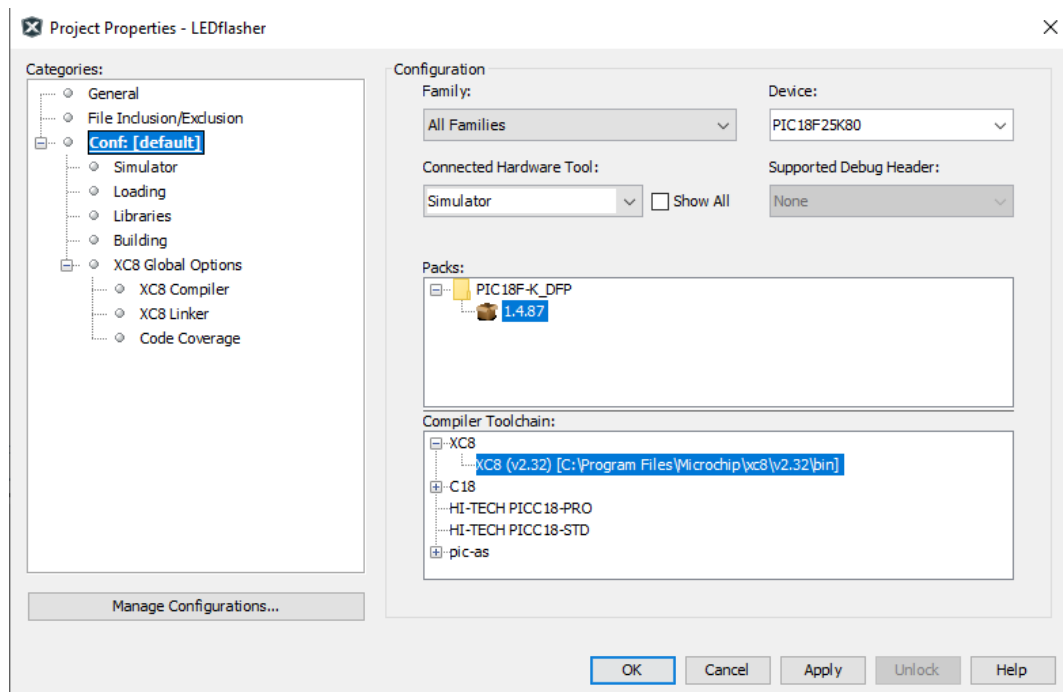
```

## 4.5 Simulate

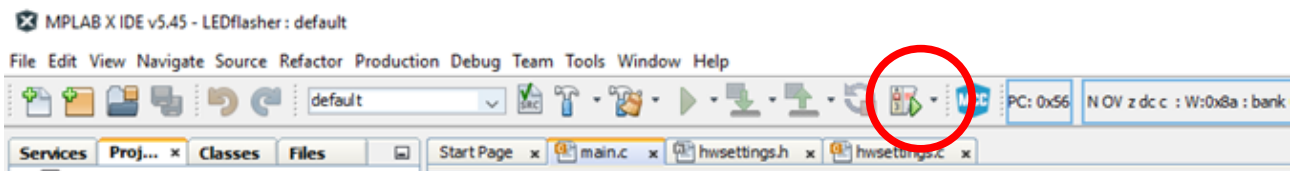
It can be useful to run the project on a simulation of the PIC processor. For projects using more complex hardware the simulator may be less useful as it can be difficult to create the hardware inputs and outputs.

For this simple project we'll try the simulator as it means a PICkit isn't needed and it is useful to understand what the simulator can do.

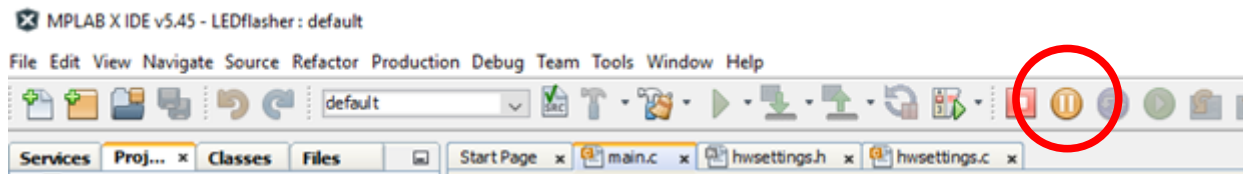
Debug the program using the simulator by pressing the ensuring the project tool is set to "Simulator" in the project properties.



Then pressing "Debug Main Project" button.
















It will initially appear as if nothing is happening so pause the program by pressing the Pause button.



You can display the value of the SFR LATB variable by using menu Debug->New watch and selecting the SFR radio button and selecting the LATB register.

The SFR LATB is the Special Function Register which controls the values output on the PortB pins of the PIC.

Now when you pause the processor the LATB value will change periodically.

Pages	Call Graph	Output	Search Results	Notifications	Variables x	Call Stack
Name		Type	Address		Value	
	<input checked="" type="checkbox"/>  LATB	SFR		0xF8A	 0x80	
 <Enter new watch>						
	 IntFlag1	unsigned char		0x7	 NUL; 0x0	
	 IntFlag2	unsigned char		0x6	 NUL; 0x0	

The most significant bit of LATB is connected to the Green LED on the CANMIO board so as the values change between 0x00 and 0x80 this represents the LED flashing on and off.

Use the green “Continue” toolbar button to resume between checks.

## 4.6 Test

It is time to try running your program on your PIC board.

Ensure the PICKit is plugged into the computer’s USB port.

Go to project properties and select the PICKit as the Tool – it will show as the PICKit’s serial number.

Plug the PICKit into the conversion lead and the lead into the ICSP connector on the target board. Ensure these connections are the right way around! I use a blob of paint to mark Pin 1 on all my boards.

Ensure the target board is powered up.

Press the “Make and Program Device” toolbar button.



MPLAB X will compile the program, connect to the PICkit and program the target device. This will take a few seconds.

After programming the target processor is released from reset and the program will run, flashing the LED.

```

*****
Connecting to MPLAB PICkit 3...

Currently loaded firmware on PICkit 3
Firmware Suite Version.....01.47.12
Firmware type.....PIC18F

Target voltage detected
Target device PIC18F26K80 found.
Device ID Revision = 6

Device Erased...

Programming...

The following memory area(s) will be programmed:
program memory: start address = 0x0, end address = 0xc97f
configuration memory
EEData memory
Programming/Verify complete

Running
|

```

## 4.7 Hardware Debugging

It is possible to use the PICkit to debug the program whilst it is running on the target hardware.

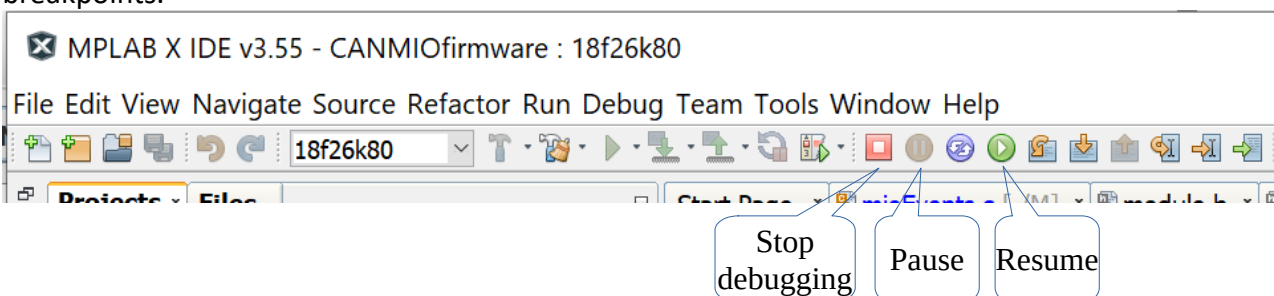
With the PICkit connected to the module's ICSP connector and selected as the Tool in the project properties it is possible to perform debugging of a program whilst it is running on the target hardware.

It is possible to set breakpoints in the program C source code, inspect variables, memory addresses and SFRs and single step through a program whilst watching the variable change.

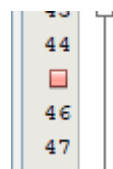
Check that PICkit is selected as Tool within the project properties

Click the "Debug Main Project" button. The project will recompile with debugging enabled and a debugging version of the program will be downloaded to the target module. Via the PICkit.

Once in debugging mode it is possible to pause the running program, inspect variables and set breakpoints.

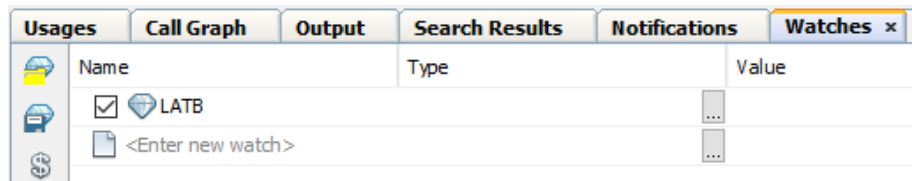


A breakpoint may be set by clicking within the left margin of the program's source code. A small red square is shown at the breakpoint location. Not every source code line may have a breakpoint set, only those lines containing executable code can have a breakpoint set. A broken breakpoint icon can be shown at an invalid location. The PICkit supports a limited number of breakpoints so you need to be selective where they are used.

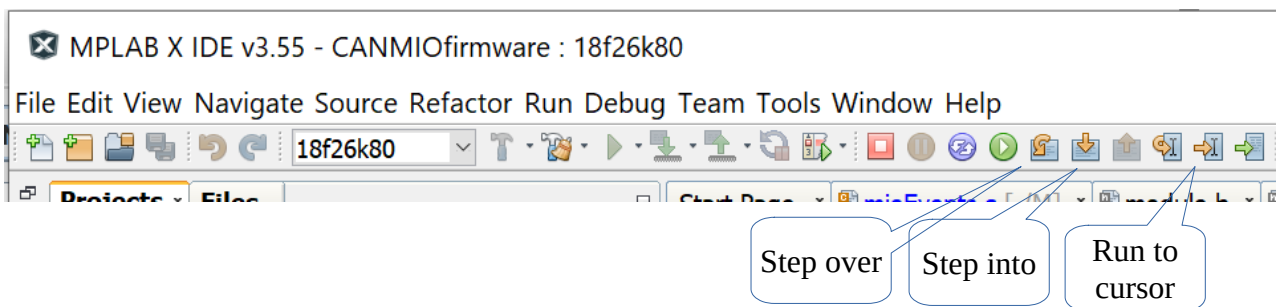


The program will automatically pause whenever the breakpoint is reached. You may select a variable in the source code and hovering over the variable will show the current value of the variable.

It is also possible to 'watch' variables to show the value of those variable whenever the program is paused.



Whilst a program is paused it is possible to single step through the program with a choice of stepping into or over functions. The run to cursor is a useful function to quickly reach later lines of code.



## 5 Using more of CBUSlib

CBUSlib is a collection of C source files which can be used to speed up CBUS module development. It provides a variety of common CBUS functionality for handling CBUS messages, event handling, CANID auto enumeration, NV support, push button and SLiM/FLiM LEDs.

If you have written a useful function that could be reused by others please contribute it to the CBUSlib library. If you extend or improve CBUSlib please contribute your changes. If you port the library to a new compiler or new processor contribute those changes too. Even improvements in the documentation would be gratefully received by the users of CBUSlib so please contribute. CBUSlib will only improve if people contribute by submitting a Github Pull Request.

### 5.1 #defines used by CBUSlib to enable functionality

Much of the functionality of CBUSlib can be changed through the use of #define statements. Control defines determine whether a block of functionality is to be supported whereas the value defines specify modifiable values, such as memory addresses.

#### 5.1.1 AREQ\_SUPPORT

Define this if the AREQ CBUS opcode is to be supported. Produced events will be saved in EEPROM at address EE\_AREQ\_STATUS and the polarity of the last event sent will be returned by AREQ.

#### 5.1.2 BOOTLOADER\_PRESENT

Define this if a bootloader present. This will enable the BOOT opcode which toggle the bootloader flag in EEPROM at address EE\_BOOT\_FLAG and resets the CPU to enter the bootloader.

#### 5.1.3 CBUS\_OVER\_CAN

Define this if CAN is supported by the module.

#### 5.1.4 CBUS\_OVER\_MIWI

Define this if MIWI is supported by the module. MIWI is currently unsupported.

#### 5.1.5 CBUS\_OVER\_TCP

Define this if TCP is supported by the module. TCP is currently unsupported.

#### 5.1.6 DEBUG\_PRODUCED\_EVENTS

Defining this will produce a ACDAT message if an event cannot be found within the event table. This should not be defined in a production build.

#### 5.1.7 HASH\_TABLE

When defined this enables a hash table to allow events to find received events within the table more quickly. This should be enabled for large event tables but it comes at the expense of additional RAM usage.

### 5.1.8 NV\_CACHE

NVs are stored in Flash memory when NV\_CACHE is defined a RAM copy is available for faster access at the expense of additional RAM usage.

The function `ModuleNvDefs* loadNvCache(void)` is used to refresh the cache from the Flash storage, this should be called whenever the application updates NVs outside of the NVSET opcode. `LoadNvCache()` must be called before attempting to read NVs from the cache so should be called early in the module initialisation. Prior to calling `loadNvCache()` you must call `initRomOps()` as `loadNvCache()` attempts to read from Flash memory.

### 5.1.9 PRODUCED\_EVENTS

When PRODUCED\_EVENTS is defined additional functions are compiled to support producing events. In addition a hash table is enabled to provide looking up of an event from the event table using a Happening.

### 5.1.10 SAFETY

When defined SAFETY additional checking is undertaken of arguments within the event handling code. This should not be necessary in production builds but can be useful for debugging and additional checks are always useful.

### 5.1.11 TEST\_MODE

TEST\_MODE enables an additional module mode when the FLiM push button is held down for more than 8 seconds. The variable `flimState` is set to `fsPressedTest`.

### 5.1.12 TIMED\_RESPONSE

When defined TIMED\_RESPONSE changes so that responses to NERD CBUS command and SoD responses are sent at a slower rate. This ensures that the module is less likely to overload other modules on the network. If the TIMED\_RESPONSE macro is defined then the `timedResponse.c` file should be included within the project. It will also be necessary to call `initTimedResponse()` in the application's initialisation routine and if `timedResponse != TIMED_RESPONSE_NONE` call `doTimedResponse()` from the modules main loop. It is recommended that the `doTimedResponse()` is called approximately every 10ms.

## 5.2 #defines used by CBUSlib to specify information

### 5.2.1 ACTION\_T

Define ACTION\_T to be the type of an Action identifier. This would normally be unsigned char or unsigned short.

### 5.2.2 AT\_EVENTS

AT\_EVENTS should be defined to be the address at which the event table resides in Flash.

### 5.2.3 AT\_NAME\_ADDRESS

AT\_NAME\_ADDRESS should be defined to be the address at which the module name resides in Flash.

#### **5.2.4 AT\_NV**

AT\_NV should be defined to be the address in Flash where the NVs are stored. The first byte may be used to store the NV data block version. The actual NVs start of AT\_NV+1.

#### **5.2.5 AT\_PARAM\_ADDRESS**

#### **5.2.6 CAN\_INTERRUPT\_PRIORITY**

Define the interrupt priority level for the ECAN peripheral.

#### **5.2.7 CHAIN\_LENGTH**

When HASH\_TABLE is defined a hash table is used for looking up events. The Chain length determines the length of chains for events with the same hash value. There is a relationship between HASH\_LENGTH and CHAIN\_LENGTH so that there is a balance between speed of look up and memory usage. Also see HASH\_LENGTH.

#### **5.2.8 EVENT\_TABLE\_WIDTH**

The event table stores the EVs for each event. When there are a variable number of EVs permitted multiple rows in the table can be utilised for store the EVs beyond that can be fitted in a single row of the table. This define contains the number of EVs in each table row. It is typically set to the lesser of the number of EVs per event and 10.

#### **5.2.9 EvperEVT**

This is the maximum number of EVs that can be stored for each event. Attempts to access EVs beyond this index will return an error.

#### **5.2.10 FLiM\_SW**

Should be defined to be the IO port to which the FLiM switch is attached. For example it can be defined as PORTAbits.RA2 if the push button is connected to the RA2. pin

#### **5.2.11 HAPPENING\_BASE**

HAPPENING\_BASE should be defined to be the lowest Happening identifier. Also see NUM\_HAPPENINGS.

#### **5.2.12 HAPPENING\_T**

HAPPENING\_T should be defined to be the type of the Happening identifier. This would normally be unsigned char or unsigned short.

#### **5.2.13 HASH\_LENGTH**

HASH\_LENGTH is the size of the hash table to be used to look up an event using NN/EN combination. Also see CHAIN\_LENGTH.

#### **5.2.14 LED1Y**

Should be defined to be the Output pin used for the Yellow SLiM LED. For example LATBbits.LATB6.

### **5.2.15 LED2G**

Should be defined to be the Output pin used for the Green FLiM LED. For example LATBbits.LATB7.

### **5.2.16 LED\_OFF**

The value to be written to the LED port in order to turn the LED off.

### **5.2.17 LED\_ON**

The value to be written to the LED port in order to turn the LED on.

### **5.2.18 LOAD\_ADDRESS**

### **5.2.19 MAJOR\_VER**

### **5.2.20 MANU\_ID**

### **5.2.21 MAX\_WRITEABLE\_FLASH**

The maximum address to which Flash writes are permitted. When an application calls romops to write to a Flash memory the address is checked to confirm that it is within the range of MIN\_WRITABLE\_FLASH and MAX\_WRITEABLE\_FLASH so that writes outside of the permitted range are blocked.

### **5.2.22 MIN\_WRITEABLE\_FLASH**

The minimum address to which Flash writes are permitted. When an application calls romops to write to a Flash memory the address is checked to confirm that it is within the range of MIN\_WRITABLE\_FLASH and MAX\_WRITEABLE\_FLASH so that writes outside of the permitted range are blocked.

### **5.2.23 MINOR\_VER**

### **5.2.24 MODULE\_ID**

### **5.2.25 MODULE\_TYPE**

### **5.2.26 NUM\_EVENTS**

NUM\_EVENTS should be defined to be the number of rows in the event table. This is the maximum number of events which can be configured in the module.

### **5.2.27 NUM\_HAPPENINGS**

If PRODUCED\_EVENTS is defined the NUM\_HAPPENINGS is used to determine the size of the event lookup table.

### **5.2.28 NV\_NUM**

NV\_NUM should be defined to be the number of NVs supported by the module.

### **5.2.29 TRIS\_LED\_1Y**

Should be defined to be the SFR used to set the pin used for the Yellow SLiM LED to be used as an output. For example TRISBbits.TRIS6.



### 5.2.30 TRIS\_LED\_2G

Should be defined to be the SFR used to set the pin used for the Green FLiM LED to be used as an output. For example TRISBbits.TRIS7.

## 5.3 Variables and defines provided by CBUSlib

### 5.3.1 EE\_APPLICATION

EE\_APPLICATION is the address in EEPROM down from which is available for application usage. Above this space is used by CBUSlib.

### 5.3.2 flimState

The flimState variable is of type FLiMStates and can be used to determine if the module is in Setup, Learn, SLiM or FLiM mode

## 5.4 CAN/CBUS/FLiM Functionality

The FLiM routines in CBUSlib are simple to use and provide all the functionality to support common CBUS operations including:

- NV reading and setting
- Optional Event support
- Core CBUS commands
- Push button operation
- SLiM and FLiM LED operations
- Self consumption of produced events
- Auto enumeration of CANID
- Optional bootloader support

### 5.4.1 Initialisation

#### 5.4.1.1 Pin usage

The PIC pins used for the pushbutton and LEDs must be set up in hwsettings.h. For example:

```
#define FLiM_SW          PORTAbits.RA2
#define LED1Y            LATBbits.LATB6  // Yellow LED
#define LED2G            LATBbits.LATB7  // Green LED
#define TRIS_LED1Y       TRISBbits.TRISB6
#define TRIS_LED2G       TRISBbits.TRISB7
```

#### 5.4.1.2 Library initialisation

During initialisation the program must call:

```
initStatusLeds();
```

```
flimInit();
```

### 5.4.1.3 CBUS Params

In order for CBUSlib to respond to a request for CBUS params it is necessary to fill out the param structure information. To be compatible with the bootloader this must be placed at 0x820.

```
#define MANU_ID          MANU_MERG
#define MODULE_TYPE      "EXAMPLE"          // MUST be at least 7 character
                                           // long, padded with spaces.
                                           // First 7 are used.
#define MODULE_FLAGS     (PF_COMBI+PF_BOOT+PF_COE) //Producer,consumer,boot
#define BUS_TYPE         PB_CAN
#define LOAD_ADDRESS     0x0800
#define MNAME_ADDRESS    (LOAD_ADDRESS + 0x20 + sizeof(ParamBlock)) // Put
module type string above params so checksum can be calculated at compile
#define PRM_CKSUM MANU_ID+MINOR_VER+MODULE_ID+NUM_EVENTS+EVperEVT+(NV_NUM-
1)+MAJOR_VER+MODULE_FLAGS+CPU+PB_CAN +(LOAD_ADDRESS>>8)+(LOAD_ADDRESS&0xFF)
+CPUM_MICROCHIP+BETA+sizeof(ParamVals)+(MNAME_ADDRESS>>8)+
(MNAME_ADDRESS&0xFF)

const rom ParamVals      FLiMparams = {
    MANU_ID,              // manufacturer
    MINOR_VER,            // minor version
    MODULE_ID,            // module id
    NUM_EVENTS,           // number of events
    EVperEVT,             // number of event variable per event
    (NV_NUM-1),           // number of node variables -1 since NV#0 is reserved
for version
    MAJOR_VER,            // Major version
    MODULE_FLAGS,         // flags
    CPU,                  // Processor Id
    PB_CAN,               // Interface protocol
    LOAD_ADDRESS,         // load address
    0,                    // processor code read from DevID at run time
    CPUM_MICROCHIP,       // manufacturer code
    BETA                   // beta release flag
    // rest of parameters are filled in by doRqnpn in FLiM.c
};

const rom char * const  module_type = MODULE_TYPE;
const rom FCUParams FCUparams = {sizeof(ParamVals), "MIO ",
(WORD) PRM_CKSUM};
```

To obtain values for the constants the cbusdefs.h file in cbusdefs repository can be used. This is also available from the MERG\_DEV Github site. <https://github.com/MERG-DEV/cbusdefs>

## 5.4.2 Main loop

A CBUS module would typically call the following functions within the main loop.

```
if (cbusMsgReceived( 0, (BYTE *)msg )) {
    shortFlicker(); // short flicker LED
    if (parseCBUSMsg(msg)) { // Process the incoming message
        longFlicker(); // extend the flicker
        return TRUE;
    }
    // handle any other CBUS messages
}
FLiMSWCheck(); // Check FLiM switch for any mode changes
checkFlashing(); // Check for any flashing status LEDs
```

## 5.4.3 Callbacks to be provided by the Application

When CBUSlib needs to access application program functions to perform checks or handling it calls back into the application code. The application must provide implementations of these functions.

Actually it isn't required since the functions may be nullified using a #define in module.h so if a function isn't required there may be some performance benefit to null the function out rather than calling an empty function.

### 5.4.3.1 *processEvent()*

If an event message is received and that event is currently provisioned within the event table then the processEvent() function is called to handle the consumed event.

```
void processEvent(BYTE tableIndex, BYTE * msg)
```

The processEvent function can obtain its EVs using getEV() or getEVs() and can determine the OPC of the message using

```
opc=msg[d0];
```

### 5.4.3.2 *getDefaultProducedEvent()*

If the module supports the idea of "software events" i.e. those that are not stored in the event table but are sent is another event has not been provisioned then the event to be sent should be returned in the Event producedEvent structure instance and the function should return true.

If the default software event functionality is not required then the getDefaultProducedEvent should just return false:

```
BOOL getDefaultProducedEvent(HAPPENING_T paction) {
    return FALSE;
}
```

### 5.4.3.3 *validateNV()*

The application's validateNV() function is called to check that value of an NV is valid before it is saved in the NV storage using the NVSET CBUS message.

```
BOOL validateNV(unsigned char index, unsigned char oldValue, unsigned char value)
```

The function should return TRUE if the NV value is valid and FALSE otherwise.

#### **5.4.3.4 *actUponNVchange()***

CBUSlib allows special handling of the NVSET CBUS message so that additional functionality can be performed when an NV is set.

```
void actUponNVchange(unsigned char index, unsigned char oldValue, unsigned char value)
```

The application program can use this callback to perform special handling when Nvs are set.

#### **5.4.3.5 *isSuitableTimeToWriteFlash()***

Writing to Flash can take a relatively long time in which the PIC processor is stopped. Before undertaking a Flash write the romops in CBUSlib asks the application program if it is safe to start the write process. If romops is being used then the application must define `isSuitableTimeToWriteFlash()`. In its simplest case this can just return TRUE.

`isSuitableTimeToWriteFlash()` should return false if the application needs to finish something else. An example is when a servo pulse has been started the CPU should not be stopped during the pulse period otherwise the CPU may not be able to stop the pulse at the correct time and the servo will twitch.

### **5.5 Event Handling Functionality**

Event support is provided within the files `events.h` and `events.c`. The consumed event handling is inherently part of CBUSlib support and cannot be entirely disabled. If no consumed events are need it is possible to provide an empty body to the `processEvent()` vallback.

Produced event handling functionality is not enabled by default and if required must be included by defining the `PRODUCED_EVENTS` #define.

There is an option to use a hash table for faster look up events within the event table which is enabled with the `HASH_TABLE` #define. If `HASH_TABLE` is defined then you also need to define `HASH_LENGTH` and `CHAIN_LENGTH` for the received events and if `PRODUCED_EVENTS` is also defined then `NUM_HAPPENINGS` must be supplied.

#### **5.5.1 Callbacks**

When a consumed event is received by CBUSlib it will call back into the application `processEvent()` to perform module specific processing of the event. EVs can be accessed using `getEv()` or `getEVs()` functions.

A produced event can be sent using the `sendProducedEvent(HAPPENING_T, BOOL)` function.

### **5.6 Action Queue Functionality**

```
ActionQueueInit();
#define ACTION_NORMAL_QUEUE_SIZE 64 // The size needs to be big enough to store all
the pending actions
// Need to allow +1 to separate the ends of the cyclic buffer so need to
```

```
// move the next power of two since cyclic wrapping is done with a bitmask.  
// 64 is safer as we have wait actions  
#define ACTION_EXPEDITED_QUEUE_SIZE 8
```

## 5.7 EEPROM and Flash Functionality

The source files of romops.h and romops.c provide functionality for reading and writing to persistent storage within the PIC.

### 5.7.1 Initialisation

The application should call initRomOps() to initialise the CBUSlib library.

### 5.7.2 Bounds checking

When writing to Flash the bounds of the write are checked against MIN\_WRITEABLE\_FLASH and MAX\_WRITEABLE\_FLASH. The write will only be done if the specified address is within the defined bounds. This provides a level of safety checking that application software does not corrupt the system.

### 5.7.3 Data versions

It is recommended that a data format version number is stored in EEPROM and Flash so that when the application is updated with changes to the persistent data format then the data can be updated.

This is not a requirement of CBUSlib but recommended good practice.

## 5.8 Bootloader Functionality

CBUSlib supports a module needing Bootloading using the existing FCU protocols. To bootloader is provided within Bootloader.asm which should be included with the MPLAB X project.

In addition BOOTLOADER\_PRESENT must be defined to enable the specific functionality within CBUSlib.

Note it is important to include Bootloader.asm within the project whenever BOOTLOADER\_PRESENT is enabled. Failure to do this will cause obscure errors.

## 5.9 Oscillator settings

The CBUS PIC bootloader is executed every time the module is reset. The bootloader tests a byte in EEPROM to see whether it should wait for a new program to be downloaded over CAN or whether to pass control to the module's application program.

The bootloader code needs to be compatible with all possible application programs. Some of the programs will require that the PLL is enabled but other will require the PLL being off. The PIC18F25K80's PLL can't be disabled from the application code so it is necessary that the PLL is disabled in CONFIG and enabled in the application if the PLL is required.

The exact mechanism to enable the PLL is dependent upon the PIC being used but the different instructions can be wrapped by #if / #endif :

```
#if defined(__18F25K80) || defined(__18F26K80)
```

```

    OSTUNEbits.PLEN = 1;
#endif
#if defined(__18F25K83) || defined(__18F26K83)
    OSCCON1bits.NOSC = 0x02;
#endif

```

## 5.10 Global Variable initialisation

The C18 compiler inserts some initialisation code before the application main() function. This function is named \_startup(void). The compiler provides a few options of what the \_startup function does. The \_startup function initialises the stack and stack pointer and optionally initialises all global variables. This has to be modified when the bootloader is used in order to change the application load address.

The Microchip supplied version of the startup function which initialises the global variables is quite an expensive operation in terms of time and code size therefore the CBUSlib version of the \_startup() function does not initialise global variables. It is therefore necessary to ensure that any global variables needing to be initialised are explicitly set in the application's initialisation routines.

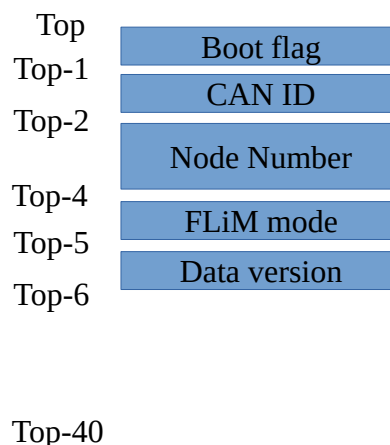
The modified version of \_startup() is in c018.c.

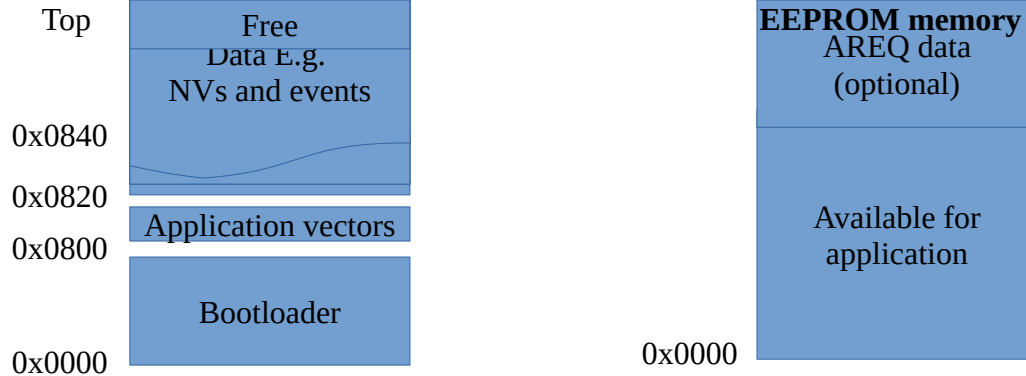
### 5.10.1 Analogue disable

Some of the PIC18F pins can be used for analogue inputs. By default these pins are assigned to be analogue. If digital inputs or outputs are required then the pins will need to be changed from analogue to digital.

## 5.11 Memory Map

Much of the complexity of a C language version of a CBUS module is caused by the memory map imposed by the Bootloader protocol and implementation.





In order to meet the requirements of the memory map the following #defines should be set:

- **AT\_EVENTS.** The event table is stored in Flash memory at the address defined by AT\_EVENTS.
- **AT\_NAME\_ADDRESS.** AT\_NAME\_ADDRESS should be defined to the address at which the module name resides in Flash.
- **AT\_NV.** AT\_NV should be defined to be the address in Flash where the NVs are stored. The first byte may be used to store the NV data block version. The actual NVs start of AT\_NV+1.
- **AT\_PARAM\_ADDRESS.** The parameter block address is specified by AT\_PARAM\_ADDRESS and should be the value of 0x820 for CBUS modules using the FCU bootloader.

## 5.11.1 Flash Memory

### 5.11.1.1 Bootloader

The 0x0000 ~ 0x07FF address space in Flash is reserved for the CBUS/FCU bootloader.

### 5.11.1.2 Application vectors

Application boot and interrupt vectors are stored between 0x800 and 0x820.

### 5.11.1.3 Param structure

The FCU bootloader requires a parameter block to be stored at 0x820 to 0x840. The FCU inspects the application HEX file to determine the application's module type and target processor to ensure that it is compatible with the target module.

The application code should define a ParamVal structure to reside at 0x820 containing the CBUS param structure:

```
const rom ParamVals      FLiMparams = {
    MANU_ID,              // manufacturer
    MINOR_VER,            // minor version
    MODULE_ID,            // module id
    NUM_EVENTS,           // number of events
    EVperEVT,             // number of event variable per event
    (NV_NUM-1),           // number of node variables -1 since NV#0 is reserved
    for version
    MAJOR_VER,            // Major version
    MODULE_FLAGS,         // flags
    CPU,                  // Processor Id
```

```

PB_CAN,          // Interface protocol
LOAD_ADDRESS,    // load address
0,               // processor code read from DevID at run time
CPUM_MICROCHIP,  // manufacturer code
BETA             // beta release flag
// rest of parameters are filled in by doRqnpn in FLiM.c
};

```

#### **5.11.1.4 Application Program**

The module's application program resides above the param structure up to any memory reserved for NVs and the event table.

#### **5.11.1.5 Event table**

Events /EVs are stored in the event table which is normally located in memory below the NVs

#### **5.11.1.6 NVs**

The NVs are normally stored at Top of Flash although in some applications storage in EEPROM may be preferred however CBUSlib currently assumes Flash storage at memory address AT\_NV. Storing NVs in EEPROM would be a useful addition to CBUSlib.

It is recommended that the first byte of the NV space is used as a Flash data version byte.

### **5.11.2 EEPROM**

#### **5.11.2.1 Bootflag**

The Bootflag is set by CBUSlib whenever the BOOTM CBUS message is received. When the module restarts execution begins in the bootloader and it checks the bootflag. If the bootflag is clear then execution is passed to \_startup() and then to the application main(). If the bootflag is set then the bootloader waits for extended CAN messages containing the new application program.

The bootflag is 1 byte.

For PIC devices containing 1K of EEPROM the address of the bootflag is 0x3FF.

#### **5.11.2.2 CAN\_ID**

The module's CAN\_ID is stored in EEPROM below the bootflag.

The CAN\_ID is 1 byte.

For PIC devices containing 1K of EEPROM the address of the CAN\_ID is 0x3FE.

#### **5.11.2.3 Node Number**

The module's Node Number is stored in EEPROM below the CAN\_ID.

The Node Number is 2 bytes.

For PIC devices containing 1K of EEPROM the address of the Node Number is 0x3FC.

#### **5.11.2.4 FLiM Mode**

The FLiMstate indicates whether the module is in SLiM, FLiM, setup or learn mode.



The FLiM Mode takes 1 byte.

For PIC devices containing 1K of EEPROM the address of the FLiM mode is 0x3FB.

#### **5.11.2.5 Data Version**

The version number of data stored in EEPROM indicates whether EEPROM contains valid information and the version of the data format.

The Data version takes 1 byte.

For PIC devices containing 1K of EEPROM the address of the Data version is 0x3FA.

#### **5.11.2.6 AREQ storage**

IF the AREQ\_SUPPORT #define is enabled then an area of EEPROM is reserved to store the state of the last transmitted event. One bit of storage is used for each possible Happening.

34 bytes are reserved for the AREQ status.

For PIC devices containing 1K of EEPROM the address of the AREQ storage is 0x3D7.

#### **5.11.2.7 Application EEPROM storage**

For PIC devices containing 1K of EEPROM the application may use EEPROM from 0x000 up to 0x3D6.

### **5.11.3 Linker**

When using the C18 compiler and linker the following pages should be set: in the linker file. The CBUSlib includes example files for PIC18F25K80 and PIC18F26K80 processors.

CODEPAGE	NAME=bootloader	START=0x0	END=0x7FF	PROTECTED
CODEPAGE	NAME=vectors	START=0x800	END=0x81F	
CODEPAGE	NAME=parameters	START=0x820	END=0x84F	
CODEPAGE	NAME=page	START=0x0850	END=0x7A4B	
CODEPAGE	NAME=persist	START=0x7A4C	END=0x7FFF	PROTECTED
CODEPAGE	NAME=userid	START=0x200000	END=0x200007	PROTECTED
CODEPAGE	NAME=cfgmem	START=0x300000	END=0x30000D	PROTECTED
CODEPAGE	NAME=devid	START=0x3FFFFFFE	END=0x3FFFFFFF	PROTECTED
CODEPAGE	NAME=eedata	START=0xF00000	END=0xF003FE	PROTECTED
CODEPAGE	NAME=eeboot	START=0xF003FF	END=0xF003FF	PROTECTED

DATABANK NAME=large\_event\_hash START=0x938 END=0xBFF PROTECTED

SECTION	NAME=large_event_hash	RAM=large_event_hash
SECTION	NAME=PARAMETERS	ROM=parameters
SECTION	NAME=BOOTFLAG	ROM=eeboot

## 6 Using CBUSlib

### 6.1 CBUS CAN processing

#### 6.1.1 Receiving CBUS messages

The application must check if CBUS messages have been received from the application main loop.

```
BYTE message[20];  
BOOL cbusMsgReceived(interface, BYTE * message)
```

TRUE is returned if a CBUS message has been received. The message is placed into the message buffer. The message should be defined to be big enough for the largest CBUS message. 20 bytes is sufficient for a CAN CBUS message.

#### 6.1.2 Handle the CBUS messages

CBUSlib handles many CBUS messages internally. This processing is performed using `parseCBUSMsg()`:

```
BOOL parseCBUSMsg(BYTE * message)
```

TRUE is returned if the CBUS opcode was recognised and the message was processed.

All events messages are separated and passed to the application's `processEvent()` function. In addition to events the CBUS opcodes that are processed by CBUSlib are:

NNLRN	RQNPN	BOOT	RQNP
NNULN	NNEVN	CANID	RQMN
NNCLR	NERD	ENUM	SNN
EVULN	NENRD	AREQ	
EVLRN	RQEVN	ASRQ	
EVLRNI	NVRD	QNN	
REQEV	NVSET		
	REVAL		

### 6.2 TickTime

Provides a means of measuring time since processor reset. Uses a hardware timer and interrupt to regularly increment a time value. Functions to obtain the 32bit time value and return a duration since a specified value are provided.

The time value is incremented approximately every 16 $\mu$ s.

#### 6.2.1 Function void `initTicker(unsigned char priority)`

This function must be called before using any of the other TickTime functions. Initialises the timer, sets up the interrupt and sets the time value to 0. The priority argument determines whether a low priority or a high priority interrupt is used.

## 6.2.2 Typedef TickValue

The TickValue type provides a 32 bit DWORD time value as TickValue.Val.

## 6.2.3 Function DWORD tickGet(void)

Returns the current time value.

## 6.2.4 Macro DWORD tickTimeSince(DWORD t)

Returns the number of ticks from the specified time to now.

## 6.2.5 Macro Time constants

The number of ticks representing time periods are provided:

- HALF\_MILLI\_SECOND
- ONE\_MILI\_SECOND
- TWO\_MILI\_SECOND
- FIVE\_MILI\_SECOND
- TEN\_MILI\_SECOND
- TWENTY\_MILI\_SECOND
- FORTY\_MILI\_SECOND
- HUNDRED\_MILI\_SECOND
- HALF\_SECOND
- ONE\_SECOND
- TWO\_SECOND
- FIVE\_SECOND
- TEN\_SECOND
- ONE\_MINUTE
- ONE\_HOUR

## 6.2.6 Example

```
main(void) {
    TickValue t;
    InitTickTime(1);
    t.Val = tickGet();
    while (1) {
        if (tickTimeSince(t) > ONE_SECOND) {
            // Do something every second
            t.Val = tickGet();
        }
    }
}
```

## 6.3 Module status

FLiM.c handles the processing of the FLiM push button and the SLiM/FLiM LED states. It is possible to determine the current module state by inspecting the FLiMStates flimState enum variable.

The possible values of flimState, are defined by FLiMStates:

```

fsSLiM=0,          // In SLiM, button not pressed
fsFLiM,            // In FLiM mode
fsPressed,         // Button pressed, waiting for long enough
fsFlashing,        // Button pressed long enough, flashing now
fsFLiMSetup,       // In FLiM setup mode (LED flashing)
fsFLiMRelease,     // Release FLiM back to SLiM
fsSetupDone,       // Exiting setup mode
fsFLiMLearn,       // In FLiM learn mode
fsPressedFLiM,     // Pressed whilst in FLiM mode
fsPressedSetup,    // Pressed during setup
#ifdef TEST_MODE
fsPressedTest,     // Pressed during test
fsTestMode,        // Self test mode
fsNextTest,        // Move on to next test
fsTestInput,       // Input to current test mode
#endif

```

It is expected that the main values used by the application software would be fsSLiM, fsFLiM, fsFLiMSetup, fsFLiMLearn.

### 6.3.1 Push Button

The application needs to call FLiMSWCheck() from the application's main loop in order to detect whenever the FLiM button is pressed.

```
void FLiMSWCheck(void)
```

## 6.4 Status LEDs

CBUSlib provides functions to control and manage the SLiM/FLiM LEDs.

### 6.4.1 Initialise

To initialise the LEDs the application should call

```
void initStateLeds(void)
```

### 6.4.2 Main loop call

In order to allow the FLiM LED to flash it is necessary that the application calls checkFlashing() from the application main loop.

```
void checkFlashing(void)
```

### 6.4.3 Flicker

If required, for example to indicate CAN bus activity, it is possible to flicker the non active LED:

```

void shortFlicker(void)    // 100ms flicker
void longFlicker(void)     // 500ms flicker

```

## 6.5 NV operations

CBUSlib handles the setting (NVSET) and reading (NVRD) of NVs from CBUS.

### Storage

NVs are stored in Flash at the location defined by the AT\_NV macro. The number of NVs must be set within the NV\_NUM macro.

The address of the NV to be written must be between the MIN\_WRITEABLE\_FLASH and MAX\_WRITEABLE\_FLASH macro values.

### 6.5.1 Validating NV changes

Before an NV is written to Flash memory the proposed value is validated with the application by calling the validateNV function:

```
boolean validateNV(unsigned char NVindex, unsigned char oldValue, unsigned char NVvalue)
```

This function should return a non-zero value if the NVvalue is acceptable and a zero value if the NVvalue is not to be accepted. If no validation of NV values is required then this function can be defined as 1:

```
#define validateNV() 1
```

Or a function can be provided by the application which simply returns 1.

### 6.5.2 Acting upon an NV change

If the module is to perform additional functionality whenever a NV is set then it is possible to define an application function which is called by CBUSlib whenever an NV is set:

```
void actUponNVchange(unsigned char NVindex, unsigned char oldValue, unsigned char newValue)
```

The application may check which NV is being changed and perform any additional processing. At the time this callback function is called the newValue has been written to Flash memory but the WRACK CBUS response has not yet been sent.

If it is not necessary to perform any additional processing then this function may be defined without a value:

```
#define actUponNVchange()
```

Or an empty function can be provided by the application.

### 6.5.3 Accessing NVs

For the application to access NV values the romops readFlashBlock function can be used. For example:

```
unsigned char nvValue;  
WORD nvAddress = AT_NV + nvIndex;  
nvValue = readFlashBlock(nvAddress);
```

Also see NvCache for a quicker and simpler way of accessing NVs.

## 6.6 NV\_CACHE

### 6.6.1 Initialisation

A type representing the layout of NVs must be defined as ModuleNvDefs type.

```
typedef struct {  
    BYTE nv_version;        // version of NV structure  
    BYTE one;                // First NV  
    BYTE two;                // Second NV  
    BYTE three;              // Third NV  
} ModuleNvDefs;
```

NV\_Cache declares a RAM based instance of this type.

### 6.6.2 Updating Cache contents

Whenever NVs are changed and NV\_CACHE is defined then loadNvCache is called by CBUSlib.

The application must also call loadNvCache before accessing NVs within the cache. The application must call loadNvCache() whenever the application changes NV values directly.

```
ModuleNvDefs* NV = loadNvCache(void)
```

### 6.6.3 Accessing NVs

LoadNvCache returns a point to the NV cache so that once the pointer has been obtained using

```
ModuleNvDefs* NV = loadNvCache(void)
```

NVs may then be accessed:

```
BYTE value = NV->one;
```

## 6.7 Event Handling

CBUSlib has functionality to manage events and EV settings using CBUS messages such as EVLRN and NERD.

EVs are stored with their event information in the event\_table. The event table is located in Flash memory at the address of the define of AT\_EVENTS.

The event\_table must be location within the bounds of memory addresses MIN\_WRITEABLE\_FLASH and MAX\_WRITEABLE\_FLASH macro values.

### 6.7.1 Receiving events

When a CBUS event message is received and that event is defined with the event\_table then the processEvent() application function is called by CBUSlib. The application should therefore provide a function:

```
void processEvent(BYTE tableIndex, BYTE * message)
```

The tableIndex is the index within the event\_table where the event is stored along with its associated EVs. The message allows the application to check the opcode of the event using

```
BYTE opc = message[d0];
```

CBUSlib provides a macro to check if the opc is for an ON event or OFF event:

```
if (opc&EVENT_ON_MASK) {  
    // OFF event  
} else {  
    // ON event  
}
```

ProcessEvent should return as soon as possible. Any long operations can be started but their processing should be performed within a separate thread from the main loop.

## **6.7.2 Getting EVs**

### **6.7.2.1 Getting a single EV**

It is possible to get a single EV for an event using the function:

```
int evValueOrError = getEv(unsigned char tableIndex, unsigned char evNum)
```

The EV value is returned unless an error is encountered. If an error is encountered then the CMDERR error value is returned as a negative number.

### **6.7.2.2 Getting all of the EVs**

Alternatively it is possible to obtain all the EVs for an event using the function:

```
BYTE result = getEVs(tableIndex);
```

EVs are placed into the global BYTE evs[EVperEVT] array. Returns 0 upon success or non zero CMDERR error value.

It is more efficient to call the function to obtain all the EVs in one go if more than 2 EVs are needed.

## **6.7.3 Getting the number of EVs**

The quantity of EVs for an event can be obtained using the functional

```
BYTE numEv(unsigned char tableIndex)
```

## **6.7.4 Adding an Event**

The module application program may wish to create events programmatically, instead of relying upon CBUS messages. It may wish to do this to provision default events.

```
unsigned char addEvent(WORD nodeNumber, WORD eventNumber, BYTE evNum, BYTE  
evVal, BOOL forceOwnNN)
```

A new entry in the event\_table is created with the single EV as specified.

If successful 0 is returned, if unsuccessful the CMDERR error number is returned.

The forceOwnNN flag indicates that if the modules node number is changed (SETNN) then the event's NN will also be updated. It is suggested that the forceOwnNN flag is set for default events and clear otherwise.

### 6.7.5 Removing an event

An event may be removed from the event\_table programmatically by the application using the function:

```
unsigned char removeEvent(WORD nodeNumber, WORD eventNumber)
```

Returns 0 if the event was found and removed. Returns the CMDERR error number otherwise.

### 6.7.6 Action queue

Since processEvent() should return as quickly as possible so that other CBUS message handling is not impacted it is recommended to perform any longer processing operations within a separate thread from the main loop.

Since this processing can take a relatively long period of time other events may be received and additional actions need to be done. It is possible to utilise the ActionQueue to maintain a backlog of actions to be performed.

## 6.8 Sending events and CBUS messages

CBUSlib provides a variety of functions for sending CBUS events and messages.

```
BOOL sendProducedEvent(HAPPENING_T happening, BOOL on)
```

Looks the Happening up within the event\_table, obtains the NN and EN from the event table and transmits a ACON/ACOF or ASON/ASOF CBUS message depending upon the NN value and the on parameter.

Returns non zero if the CBUS message was successfully sent.

```
BOOL cbusSendEvent(BYTE interface, WORD NN, WORD EN, BOOL on)
BOOL cbusSendMsgNN(BYTE interface, WORD eventNode, BYTE * message)
```

Transmits a CBUS event or message. The interface parameter should be 0 (CBUS\_OVER\_CAN) when using a CAN interface.

Returns non zero if the CBUS message was successfully sent.

## 6.9 ActionQueue

The ActionQueue is supported by a collection of functions for managing a queue of actions to be performed. The type of items stored in the queue is ACTION\_T type.

### 6.9.1 Initialisation

Instantiate a variable of type Queue and initialise the elements of the structure:

```
Queue q;
ACTION_T queueBuf[QUEUE_SIZE];
```



```
q.size = QUEUE_SIZE;
q.readIdx = 0;
q.writeIdx = 0;
q.queue = queueBuf;
```

### 6.9.2 Push

To add a new action onto the queue call:

```
BOOL push(Queue * q, ACTION_T a)
```

FALSE is returned if the queue is full, TRUE is return upon success.

### 6.9.3 Pop

The top of the queue is removed and returned.

```
ACTION_T popAction(Queue * q)
```

NO\_ACTION is returned if the queue was empty.

### 6.9.4 Peek

An element of the queue can be inspected.

```
ACTION_T peek(Queue * q, unsigned char index)
```

Index specifies the locoaon of the queue to be inspected. 0 is top of queue

NO\_ACTION is returned if there is no eleemnt at the specified index.

### 6.9.5 Queue size

The number of elements currently in the queue can be determined by calling

```
unsigned char quantity(Queue * q)
```

## 6.10 RomOps

CBUSlib provides functions for reading and writing Flash and EEPON memory spaces.

Flash must be written in blocks with an erase operation to be done before write. CBUSlib hides this complexity from the application. The current Flash block is maintained in RAM, updates are made to this RAM copy and is updated upon request or if a read or write is performed to a different block.

### 6.10.1 Initialisation

The initialisation function must be called before other romOps functions.

```
void initRomOps(void)
```

### 6.10.2 Reading EEPROM

Read a single byte from EEPROM from the specified address.

```
BYTE ee_read(WORD addr)
```

Read a short (2 byte) value from EEPROM

```
WORD ee_read_short(WORD addr)
```

### **6.10.3 Writing EEPROM**

Write a byte to EEPROM at the specific address.

```
void ee_write(WORD address, BYTE data)
```

Write two bytes to EEPROM.

```
void ee_write_short(WORD address, WORD data)
```

### **6.10.4 Reading Flash**

If the block covering the specified address is not currently loaded then the entire block is read into RAM copy. The byte from the specified address is returned.

```
BYTE readFlashBlock(WORD address)
```

### **6.10.5 Writing Flash**

Write a byte to the RAM memory copy.

```
void writeFlashImage(BYTE * address, BYTE data)
```

Write a byte to the RAM copy and then flash the copy to flash.

```
void writeFlashByte(BYTE * address, BYTE data)
```

Write a pair of bytes to the Flash RAM copy.

```
void setFlashWord(WORD * address, WORD data)
```

### **6.10.6 Flush Flash copy**

After using writeFlashImage only the RAM copy is updated. The RAM copy can be written to Flash by calling flushFlashImage()

```
void flushFlashImage(void)
```

It is recommended that if multiple Flash location are to be written then writeFlashImage() should be used followed by a single call to flushFlashImage()

### **6.10.7 Obtaining the CPU type**

The application may obtain the type of the processor being used by calling readCPUType. This should be returned as the CBUS parameters.

```
WORD readCPUType(void)
```

## **6.11 Handling other CBUS opcodes**

In some applications the application software must be able to handle additional CBUS opcodes. For example it may be required to have special handling for events with data bytes or for handling the opcodes for long messages and streaming.

There are two places where specialised handling can be performed.

### **6.11.1 Special handling Pre CBUSlib**

When `cbusMsgReceived(interface, message)` returns `TRUE` the CBUS message has been placed in the message buffer. The application is able to inspect the opcode of the message using `message[d0]` and perform any action required. If no special action is needed for the message's opcode then the message should be passed to `parseCBUSMsg(message)`.

### **6.11.2 Special handling Post CBUSlib**

If `parseCBUSMsg(message)` returns `FALSE` then the message was not acted upon by CBUSlib and the application should check if any additional processing for that message is required.

## 7 Typical Application

The following is a typical sequence of calls within a PIC18F25K80 based CBUS module. This can be used as a template for producing module application code.

```
main(void) {
    InitRomOps()
    NV = loadNvCache();
    initStatusLeds()
    flimInit()
    initTicker(0); // set low priority
    WPUB = NV->pullups;
    actionQueueInit();
    mioEventsInit();
    mioFlimInit(); // This will call FLiMinit, which, in turn,
                  //      calls eventsInit, cbusInit
    ANCON0 = 0x00;
    ANCON1 = 0x00;
    ei(); // enable interrupts, all init now done

    for (;;) {
        if (cbusMsgReceived( 0, (BYTE *)msg )) {
            shortFlicker(); // short flicker LED
            if (parseCBUSMsg(msg)) { // Process the incoming message
                longFlicker(); // extend the flicker
                return TRUE;
            }
            // handle any other CBUS messages
        }
        FLiMSWCheck(); // Check FLiM switch for any mode changes
        // Check for any flashing status LEDs
        checkFlashing();
    }
}
```