The Anatomy of a Micromouse Programme

With enough C to get by!

By Chris Jobling

Acknowledgement

The code that this document is based on is tachovoid.c written by Dr Timothy Davies for EG-252 Group Design Exercise.

File header

Each programme should have an explanatory comment at the top which describes the purpose of the programme and gives some technical detail. In this case, `tachoavoid` is a test programme that provides the minimum functionality needed to successfully achieve obstacle avoidance under the assumption that there are two infrared sensors giving a digital input of logic "1" if they detect an obstacle and two touch bars which go to logic "0" if they touch an obstacle.

The mouse will drive forward if there are no obstacles visible.

The motors are driven using the same PWM set up examined in Exercise 4.

In addition there is some feedback, provided by tachometers, that is used to adjust the PWM duty cycles in order to keep the motors are roughly the same speed so that the mouse will travel in a straight line.

This is summarized in the main comment

```
/*

Test Programme for the AW60 micromouse board


This version gives direction control of the two motors but with
interrupts to
give PWM speed control. Inputs from the Hall Effect tachometers are
used to
adjust the motor speeds so that the machine travels in a straight
line.

Three interrupts are used to PWM the motors: the timer overflow for
TPM1 timer,
which sets the overflow frequency of 100 Hz with bus clock rate 2MHz,
turns both
motors on. TPM1 channel 1 and TPM1 channel 2 turn off the left and
right motors.
Variables `PWM_LEFT` and `PWM_RIGHT` are the "instantaneous" values
of the PWM.

Two interrupts are used to measure the speed of the left and right
motors.
```

```
Consecutive TPM2 channel 0 and channel 1 interrupts are subtracted to
give
two variables `DIFF_LEFT` and `DIFF_RIGHT` which are used by the
procedure "speedcon".

The main programme performs a simple avoidance action, using active
low inputs from
the touch bars and active high inputs from the IR sensors.

Variable `COUNTER` is a 16-bit integer which is decremented every 10
ms.
Variable `DISTANCE` is a 16-bit integer which is decremented after
every
tachometer pulse.

One mm of travel is approximately 4 units in `DISTANCE`.

*/
```

Include files and Macros

Since its early days, C has used a two pass compilation process. In the first pass, a *macro processer* is used to interpret macro instructions and replace any symbol that matches the definition with the result of the macro expansion. This is a multiple pass process and the macro processor loads and reloads the source files sufficient times to ensure that all macros have been replaced by their definitions.

One this is done, the c-compiler takes over and compiles the actual source code. If you are sufficiently interested, you can view the resulting pure c programme, but this is rarely necessary with modern compilers and IDEs.

There are two types of macro that we use: `#include` and `#define`

Hash-include `#include`

This literally *includes* the *content* of a file at the point of declaration.

So in the following, we are loading the definitions needed for the `EnableInterrupts` macro and a file that lists all the standard names for the registers etc that are defined in the AW60 reference manuals.

```
#include <hidef.h>      // for EnableInterrupts macro
#include "derivative.h" // include peripheral declarations
```

There is a subtle difference in semantics between the syntax `#include <hidef.h>` and syntax `#include "derivative.h"` which need not concern us here. Just be aware that any programme for the AW6p should include these two lines.

Hash-define `#define`

The hash-define keyword is used to define constants and sometimes structures that look like functions with arguments. The macro processor expands these in place so the second argument is literally written into the code at the point at which the second argument is used in the code.

Our main purpose is to to define symbols for constants, registers, or bits within registers, so that we can provide readable definiations for values that have some useful meaning. Note that these are not assignments, the definitions do not create any new variables. There are no semi colons in the definitions.

Why do we have macros?

The main benefit of the macro processor is to ensure that we avoid typing raw numbers into code or tie ourselves down to a particular hardware configuration.

We can thus change the global definition of a symbol, constant or memory location simply by changing a single definition, or loading a different `derivative.h` file.

When you look at the code in a debugger, there will not be a variable for a macro, rather there will be a number.

In a microcontroller code, the values yielded by the macr pre-processor will be a data value to be used in an instruction, equivalent to the *load immediate instruction* in assembler, or the address of a register in memory, or the content of a bit, or group of bits, in one of those registers.

Conventions for use of macros

Macro definitions are usually written using all caps with underscores between words. This is so they standout visually when you read the code.

It is also conventional to define all your macros at the top of the programme before the actual code starts.

It's a good idea to add comments to your macro definitions and give them meaningful names.

Professional C-programmers will often put macro definitions into separate include files so that they can easily be shared across multiple source files within a project.

Micromouse macros

```
#define TBFL PTAD_PTAD1

#define STOP 0b00000000
#define FWD  0b00000101
#define REV  0b00001010
#define ACW  0b00001001
#define CW   0b00000110
#define SFL  0b00000001 //Swerve forward and left
#define SFR  0b00000100 //Swerve forward and right

#define bitClear 0
#define SCALE 1
```

Function prototype definitions

Standard C recommends that you define the structure of any subroutines you will be using later using "prototypes".

This is primarily to ensure that a single definition of a function can be shared across a project and to faciltate the creation of shared libraries. But it also helps the compiler to check that function usage is consistent across a project.

In our case, this set of prototypes, which define the high-level programming interface for the micromouse, could be in a separate include file as would the definition of the subroutines.

This would also facilitate the creation of a library for the AW60 and another for say, an Arduino, easier to develop and compile in as appropriate.

The function prototypes for the micromouse are:

```
void revleft(void);
void revright(void);
void stop(void);
void reverse(void);
void turnleft(void);
void turnright(void);
void iravoidl(void);
void iravoidr(void);
void speedcon(void);
```

Note a `void` function with a `void` argument passes no arguments and returns no values. We call a function that returns nothing a *subroutine*.

As the project develops, you will probably wish to add more functioms to this list.

Variable declarations

Any variables that you are going to use should be defined next. The types are important in C, not only in the sense that they define the size of a data value (usually 4 bits for a character and 8 or 16 bits for integers), but also how they are interpreted in programmes, e.g. signed, unsigned, ASCII character, floating point number, etc.

You can also use `typedef` to define new names for specific types or even to define structures. We have largely avoided using `typedef` here, though you have seen examples of its use in the lab exercises, and there are others in the *derivative.h* include file.

It is conventional to use lower-case letters for variables, but Dr Davies regularly violates this convention!

Data types

How much memory is assigned to the standard data types in C values is usually an implementation issue and you need to refer to the user manual.

In the AW60, the 1-byte datatypes `short`, `byte` or a `char` are signed 4-bit integers; the 2-byte `word` and `int` datatypes are signed 8-bit integers. A `long` is a 4-byte (32-bit) signed integer.

You can prefix the type with the keyword `unsigned` which prevents twos-complement arithmetic.

Other options which are rarely used in microcontrollers are `float` (32 bits), and `double` (64 bits) which can be used to represent real (floating point) numbers.

Constants

```
byte MODCNTH = 0x4E, MODCNTL = 0x20; // values for the modulo
registers (0x4E20)

byte INIT_CHNCNTH = 0x40, INIT_CHNCNTL = 0x00; // set the IOC
register to 0x3000 for output compare function

word REPEAT = 0x4E20; // sets PWM repeat time (frequency = 100Hz)
word NOM_SPEED = 0x2000;
word PW_LEFT = 0x2000;
word PW_RIGHT = 0x2000;
word PW_MAX = 0x3000;
word PW_MIN = 0x1000;
```

Global variables

Global variables are data values that are visible to the entire programme. They are often used in microcontroller applications because you have limited storage.

These are the key global variables that are used in the low-level drive functions.

```
byte DRIVE;

word DISTANCE;
word NEW_LEFT;
word OLD_LEFT;
word DIFF_LEFT;

word NEW_RIGHT;
word OLD_RIGHT;
word DIFF_RIGHT;

word CORR = 0;

word COUNTER;
```

Generally, the use of global variables is to be avoided. Instead, modern practice is to pass values around through function parameters and return values supported by local variables that exist only within the scope of a block or a function. The stack is used for local variable scoping and parameter passing.

```
//The motors are connected to Port F bits 0 to 3 in pairs.
//Sense of switching is 01 = forward, 10 = reverse,
//00 = brake, 11 = freewheel.
```

The Main function

In C, programme execution always starts in the `main` function.

```
void main(void)
{

    // intialization

    // enable interrupts
    // main loop
}
```

Initialization

In microcontroller applications, the first part of `main` is concerned with the initialisation of ports, control registers, interrupts etc. Initialization happens only once.

COP and clock source

In the AW60, we first turn off the watchdog timer and select the external quartz crystal so that we will have a bus frequency of 2 MHz.

```
SOPT = 0x00; // disable COP (watchtimer)
```

```
ICGC1 = 0x74; // select external quartz crystal
```

General Port IO

Next we set up the General I/O ports as digital inputs or outputs using the data-direction registers and the pull-up enable operators as appropriate for the types of input.

Note, digital ports are inputs by default!

```
// Init_GPIO init code
```

```
PTFDD = 0x0F; // set port F as outputs for motor drive on bits PTFD
bits 0-3.
```

```
PTDPE = 0b00001100; // use PTDD bits 2, 3 as input ports for touch
bars
```

Timers and PWM

In this case, the timer one is used initialise two completely automatic, interrupt-driven, PWM processes which will drive the motors at a constant velocity and the mouse in a forward direction.

You did this yourselves in Exercise 4.

Use timer 1 to generate a base frequency of 100 Hz.

```
// configure TPM module 1
```

```
TPM1SC = 0x48; // format: TOF(0) TOIE(1) CPWMS(0) CLKSB(0) CLKSA(1)
PS2(0) PS1(0) PS0(0)
```

```
TPM1MOD = REPEAT; // set the counter modulo registers to 0x4E20 (=
20000 decimal).
```

A couple of timer interrupts are used to generate 2 PWM signals.

```
// configure TPM1 channel 1
```

```
TPM1C1SC = 0x50; // TPM1 Channel 1
```

```
TPM1C1V = NOM_SPEED; // set the channel 1 registers to 0x1000
```

```
TPM1C2SC = 0x50; // TPM1 Channel 2

TPM1C2V = NOM_SPEED; // set the channel 2 registers to 0x1000
```

Timer 2 is used as a data-capture system to measure the speed of each motor. The drive algorithm adjusts the duty cycle in order to keep the motors running at the same speed.

You haven't seen this before so it deserves study.

```
TPM2SC = 0x08;   //select bus clock for TPM2, no TOV
TPM2C0SC = 0x44; //turn on edge interrupt for TPM2 C0
TPM2C1SC = 0x44; //turn on edge interrupt for TPM2 C1
```

If nothing else happens, the mouse will move forwards, hopefully in a straight line.

Enable interrupts

We then enable interupts and go into the main loop.

```
EnableInterrupts; // enable interrupts
```

The event loop

This use of an endless loop inside a programme is very common in modern computing and is called an *event loop*. Programmes that use an event loop and rely on polling or interrupts to change their behaviour are called *even-driven programmes*.

Every time around the main loop, the four digital inputs, two touch-bar inputs -- which go to logic 0 if they touch something -- and two infrared sensors (left and right) -- which go to logic 1 if they "see" something -- are polled and suitable commands are issued to change manouvre the mouse depending on the combinations of inputs that are active.

Event-driven programmes are very challenging to build, test and debug because when events will occour is unpredictable.

They are even harder in microcontrollers because you do not have easy access to a `print` function or a console through which you can monitor the current state of your programme.

The event loop for obstacle avoidance is quite simple for the micromouse. Effectively it is drive forward unless you see an obstacle.

```
for (;;)
{
    DRIVE = FWD;

    // poll sensors

    // if touch bars touched ... reverse and turn

    // if IR sensors activated ... turn to avoid

} // loop forever
```

I haven't provided all the code, the detail is to be found in the code tachoavoid.c

Note for Arduino users

Ardino programmers do not have direct access to the `main` function. Instead there is an `setup` method for initialization which is called once and a separate `loop` function that starts the *event loop* and is executed automatically every loop cycle.

Low-level functions

The functions that link the basic operation of the microntroller to the higher-level functions are close to the machine. There are two types in `tachoavoid.c` . These are the interupt service routines and the basic movement functions.

Interrupt service routines

In this example the ISRs are all concerned with timer functions.

I just list these without further comment.

Timers for PWM

Set base frequency and turn on motors.

```
interrupt 11 void TPM1SC_overflow()
{ // interrupt vector: Vtpm1

    TPM1SC_TOF = bitClear;

    PTFD = DRIVE; // turn on motors as configured by DRIVE (port A
switches).

    speedcon();

    if (COUNTER != 0)
    {
        COUNTER--;
    }
}
```

Turn motor 1 off at the end on the "on" cycle.

```
interrupt 6 void TPM1C1SC_int()
{ // interrupt vector: Vtpm1ch1

    TPM1C1SC_CH1F = bitClear;

    PTFD = PTFD | 0x03; // set free-wheel mode for one motor instead
of turn off
}
```

Turn motor 2 off at the end on the "on" cycle.

```
interrupt 7 void TPM1C2SC_int()
{ // interrupt vector: Vtpm1ch1

    TPM1C2SC_CH2F = bitClear;

    PTFD = PTFD | 0x0C; // set free-wheel mode for other motor
instead of turn off
}
```

Timers used for speed measurement

Uses data capture function to measure time between pulses on the two tachos.

Motor 1: Measure and adjust

```
interrupt 12 void TPM2C0SC_int()
{
    TPM2C0SC_CH0F = bitClear;

    NEW_LEFT = TPM2C0V;
    DIFF_LEFT = NEW_LEFT - OLD_LEFT;
    OLD_LEFT = NEW_LEFT;
    if (DISTANCE != 0)
    {
        DISTANCE--;
    }

    //  PTBD = PTBD ^ 0B00000001;
}
```

Motor 2: measure and adjust

```
interrupt 13 void TPM2C1SC_int()
{
    TPM2C1SC_CH1F = bitClear;

    NEW_RIGHT = TPM2C1V;
    DIFF_RIGHT = NEW_RIGHT - OLD_RIGHT;
    OLD_RIGHT = NEW_RIGHT;
    if (DISTANCE != 0)
    {
        DISTANCE--;
    }

    //  PTBD = PTBD ^ 0B00000010;
}
```

Basic movement

These are the basic building blocks for the micromouse. They are the key functions that communicate directly with the hardware. If you were to implement a micromouse using a different microcontroller, these are the only functions that you would need to rewrite in order to port your high-level functions to different hardware.

It is worth noting that global variables `DRIVE` and `DISTANCE` are used to track the state of the mouse drive system at any given time. There are alternative ways to do this which would not require global variables, but that is left as an exercise for the reader.

Once again, the code is presented without too much commentary. The function is fairly obvious from the names chosen for the functions.

```
void stop(void)
{
    DRIVE = 0;
}
```

```
void reverse(void)
{
    DISTANCE = 200;
    DRIVE = REV;
    while (DISTANCE != 0)
    {
    }
}

void turnleft(void)
{
    DISTANCE = 100;
    DRIVE = ACW;
    while (DISTANCE != 0)
    {
    }
}

void turnright(void)
{
    DISTANCE = 100;
    DRIVE = CW;
    while (DISTANCE != 0)
    {
    }
}
```

High-level functions

These are defined in terms of the simple functions `stop` , `reverse` , `turnleft` , `turnright` defined above.

These functions can be used as the building blocks for further higher level functions sunch as `turn_on_a_corner` for line following, or `attack` and `retreat` for combat.

Building up "layers of abtraction" using functions is the key to creating larger systems.

It allows us to free ourselves from the nitty-gritty details needed to drive the mouse and instead concentrate on implementing the desired required functionality: `onstacle_avoidance` , `line_following` , `combat` and even other funtions such as `maze_solving` .

```
void revleft(void)
{
    stop();
    reverse();
    stop();
    turnleft();
    stop();
    //forward
}

void revright(void)
{
    stop();
    reverse();
    stop();
    turnright();
    stop();
    //forward
}
```

```
void iravoidl(void)
{

    stop();
    turnleft();
    stop();
    //forward
}
```

```
void iravoidr(void)
{

    stop();
    turnright();
    stop();
    //forward
}
```

Speed control

This is the most complex code in the micromouse programme. It uses any difference in the speed measurements to adjust the PWM duty cycle of each motor as necessary to ensure that the are both running at the same speed.

I will leave to Dr Davies to explain how it works!

```
void speedcon(void)
{

    if (DIFF_RIGHT != DIFF_LEFT)
    {

        if (DIFF_RIGHT > DIFF_LEFT)
        {

            CORR = (DIFF_RIGHT - DIFF_LEFT) / SCALE;
            PW_LEFT = PW_LEFT + CORR;
            if (PW_LEFT > PW_MAX)
            {
                PW_LEFT = PW_MAX;
                PW_RIGHT = PW_MIN;
            }
            else
            {
                PW_RIGHT = PW_RIGHT - CORR;
            }
        }

        else
        {

            CORR = (DIFF_LEFT - DIFF_RIGHT) / SCALE;

            PW_RIGHT = PW_RIGHT + CORR;
            if (PW_RIGHT > PW_MAX)
            {
                PW_RIGHT = PW_MAX;
                PW_LEFT = PW_MIN;
            }
            else
            {
                PW_LEFT = PW_LEFT - CORR;
            }
        }
        //The next two lines may need to be swapped over
        TPM1C2V = PW_LEFT;
        TPM1C1V = PW_RIGHT;
    }
}
```

Other Resources

- Dr Clifford's notes on C for EG-151 2019.
- Dr Clifford's C Language Cheat Sheet.
- Source for this document tachoavoid.ipynb
- Printable version of this document tachoavoid.pdf