MCUS Manual

Philip Withnall

philip@tecnocode.co.uk

This manual describes version 0.2.1 of MCUS.

Revision History

Revision MCUS Manual V0.2.1 November 2008
Philip Withnall <philip@tecnocode.co.uk>

MCUS is a microcontroller simulator designed to the OCR microcontroller specifications for the AS electronics syllabus, first taught in 2008. It is designed to completely follow the specification, and provide a helpful interface to introduce students to programming in the language.

1. Introduction

Use MCUS to develop and simulate programs in OCR assembly, using a variety of simulated hardware. It has the following features:

- Full support for OCR assembly.
- Input hardware: hexadecimal input and bit-level switches.
- Output hardware: hexadecimal output, bit-level LEDs, multiplexed SSDs, BCD-encoded SSDs and other encoded SSDs.
- Context-sensitive programming help.
- Step-through debugging and variable-speed simulation.
- Full memory, stack and register listings.

2. Getting Started

2.1. Starting MCUS

You can start MCUS in the following ways:

Applications menu

Choose Education⟶MCUS Microcontroller Simulator.

Command line

To start MCUS from a command line, type the following command, then press **Return**:

mcus

2.2. When You Start MCUS

When you start MCUS, the following window is displayed.

Figure 1. MCUS Main Window

Shows MCUS main window, containing the editing area, simulation panel, and hardware panel.

The MCUS window contains the following elements:

Editing area.

The editing area is where all the code for an assembly program is written. It features syntax highlighting, and also highlights the currently-executing line when a simulation is running.

Simulation panel.

The simulation panel (on the right-hand side of the window) contains data from and controls for the simulation, including listings of memory, the registers and stack; as well as hexadecimal controls for the input, output and analogue ports.

Hardware panel.

The hardware panel displays all the available input and output hardware. All the inputs are linked together, so changing one type of input will be reflected in the others.

3. Usage

3.1. Editing a Program

To open a program, choose File⟶Open, then select the `*.asm` you'd like to open.

To save a program, choose File⟶Save or File⟶Save As, then select where you wish to save the file.

Files written by MCUS are editable in a text editor, and the file only stores plain assembly code.

To edit the program, simply type in the editing area. It is a good idea to put each new instruction on a new line, though this is not required. For a reference to the mnemonics and syntax accepted by MCUS, see Section 4.

3.2. Simulating a Program

Once a program is ready to be simulated, press the Run button on the toolbar, choose Program⟶Run or press **F5**.

The program will step through at the Clock Speed set in the simulation panel, with the current execution being highlighted in the editing area. Hardware outputs in the hardware panel and simulation displays in the simulation panel will be updated as the program executes.

While the program is running, most of the MCUS interface will be disabled. To pause the program and allow controls to be used, press the Pause button on the toolbar, choose Program⟶Pause or press **F6**.

When paused, the program can also be stepped-through, instruction by instruction, to see exactly what's going on. To do this, press Step Forward on the toolbar, choose Program⟶Step Forward or press **F8**.

To stop simulation of the program at any time, press Stop on the toolbar, choose Program⟶Stop or press **F7**.

3.3. Using Inputs

There are two ways to set the data on the simulated input port in MCUS:

The Input Port entry in the simulation panel.

This accepts a hexadecimal value from `00` to `FF` to set the one-byte input port.

The switches on the Inputs page of the hardware panel.

These allow individual bits of the input byte to be toggled. The most-significant bit is on the left, and the least-significant is on the right.

To read data from the input port in a program, use the `IN` instruction; see Section 4.

3.4. Using Outputs

There are several different ways to visualise data put on the output port by an assembly program:

The Output Port entry in the simulation panel.

This displays a hexadecimal value from `00` to `FF` from the output port.

The LEDs on the LEDs page of the Outputs of the hardware panel.

These display each bit of the output port, with the most-significant on the left, and the least-significant on the right.

The SSD on the Single SSD page of the Outputs of the hardware panel.

This displays a value decoded according to the option on the left.

If Segments is selected, segment A is controlled by the least-significant output bit, with segments B to G being controlled by progressively more significant bits. The decimal point is controlled by the most-significant bit.

If BCD is selected, the SSD will display the least-significant binary coded decimal digit in the output (i.e. the lower four bits of the output). If the BCD is invalid, `0` will be displayed.

The SSDs on the Dual SSDs page of the Outputs of the hardware panel.

These display two digits as represented as binary coded decimals in the output. The left-hand SSD displays the most significant BCD. If a BCD is invalid, `0` will be displayed.

The SSDs on the Multiplexed SSDs page of the Outputs of the hardware panel.

These display digits as decoded from the output. The most-significant nibble of the output specifies which SSD to set, and the least-significant nibble gives a BCD to use as its value. Invalid BCDs set an SSD to `0`.

All SSDs apart from the one specified in the most-significant nibble of the output are blanked every time the matrix of SSDs is updated.

3.5. Using the ADC

The ADC (Analogue-to-Digital Converter) acts as another input to the simulated hardware, digitising a generated analogue signal, and making it available via the `readadc` built-in subroutine.

There are two different ways the ADC can be set up to generate a signal, and both can be found on the ADC page of the hardware panel:

Constant Signal.

: This allows a constant analogue signal between 0V and 5V to be set, which would provide a constant digital reading from `readadc`.

Function Generator.

: This allows the analogue signal to be produced by a function, whose parameters are set up on this page:

  Waveform

  : The shape of the analogue waveform. Choose from: Sine Wave, Square Wave, Triangle Wave and Sawtooth Wave.

  Frequency

  : The frequency of the waveform, in Hertz. Be wary of setting it as a multiple of the simulation clock speed, as if they're in phase, only one point on the wave will ever be seen by the program.

  Amplitude

  : The amplitude of the waveform, in Volts from 0V to 5V.

  Offset

  : The offset of the waveform from 0V, in Volts from 0V to 5V. To allow the maximum amplitude of 2.5V without clipping, set the offset to 2.5V.

  Phase

  : The phase of the waveform in rads.

4. Assembly Reference

4.1. Syntax

Instructions consist of a mnemonic, followed by zero or more parameters. Mnemonics are not case-sensitive, and a space separates the mnemonic and its parameters. Instructions are separated by any whitespace, although it's clearest to put each instruction on a new line.

Label declarations consist of the label name followed immediately by a `:`. They do not operate at a block level, and merely provide a convenient way to reference a particular location in a program, typically for jumping to it.

The rest of a line can be marked as a comment (and consequently ignored by the compiler) by using a `;`. It is important to document code using such comments.

Parameters can be of the following types, and each instruction is strict as to what it accepts. Parameters are separated by whitespace and an optional comma:

Register: Sn

> A case-insensitive `S`, followed by a register number from `0` to `7`. This type of parameter specifies a register from which data should be read, or to which data should be written (depending on the instruction in question).
>
> Examples: `S0`, `S5`

Constant

> A two-digit hexadecimal constant which would either be treated as a number, or could be treated as a memory address by some instructions, and can be passed in place of a label to any instruction.
>
> Examples: `05`, `7F`

Label

> A case-sensitive reference to a label defined elsewhere in the program, consisting of letters, digits and underscores. Labels do not have to be declared before they're used, but compilation of a program will fail if a non-existent label is referenced.
>
> Note that if a label is of the same form as a hexadecimal constant, it will be mistaken for the memory address given by that constant, rather than treated as a label.

Examples: `foobar3`, `main_loop`

Input

A case-insensitive `I`, which refers to the only input port available in the simulation.

Output

A case-insensitive `Q`, which refers to the only output port available in the simulation.

MCUS introduces the concept of *directives* to the OCR assembly specification, adding a SET directive, to allow memory locations to be pre-set to specific values on compilation.

Directives take the form of a `$` followed immediately by the case-insensitive directive name, followed by its parameters as if it were a normal instruction. Directives will be acted upon by the compiler, but then removed, and not added to the compiled code or executed when the program's simulated.

Warning

The addition of directives is non-standard, and has not been discussed with OCR. Although they are not on the specification, their addition was deemed necessary to make the `readtable` subroutine useful.

4.2. Instructions

MCUS supports the full list of instructions in the OCR specification, plus one extra implicit `HALT` instruction, which has an opcode of `00`, and is thus executed as soon as uninitialised memory is encountered (i.e. the end of the program is reached).

Warning

The `HALT` instruction, while designed to be implicit, can be used explicitly just as a normal instruction. None of this is on the specification, and is an extension specific to MCUS

`HALT`

Terminate the simulation.

Opcode: `00`

Example: `HALT`

`MOVI` *`Sd`*, *`n`*

Copy the byte *`n`* into register *`Sd`*.

Opcode: `01`

Example: `MOVI` *`S0`*, *`5F`*

`MOV` *`Sd`*, *`Ss`*

Copy the byte from *`Ss`* to *`Sd`*.

Opcode: `02`

Example: `MOV` *`S3`*, *`S6`*

`ADD` *`Sd`*, *`Ss`*

Add the byte in *`Ss`* to the byte in *`Sd`* and store the result in *`Sd`*.

Opcode: `03`

Example: `ADD` *`S2`*, *`S4`*

`SUB` *`Sd`*, *`Ss`*

Subtract the byte in *`Ss`* from the byte in *`Sd`* and store the result in *`Sd`*.

Opcode: `04`

Example: `SUB` *`S4`*, *`S2`*

`AND` *`Sd`*, *`Ss`*

Logical AND the byte in *`Ss`* with the byte in *`Sd`* and store the result in *`Sd`*.

Opcode: `05`

Example: `AND` *`S0`*, *`S3`*

`EOR` *`Sd`*, *`Ss`*

Logical EOR the byte in *`Ss`* with the byte in *`Sd`* and store the result in *`Sd`*.

Opcode: `06`

Example: `EOR` *`S1`*, *`S6`*

`INC` *`Sd`*

Add 1 to *`Sd`*.

Opcode: `07`

Example: `INC` *`S0`*

`DEC` *`Sd`*

Subtract 1 from *`Sd`*.

Opcode: `08`

Example: `DEC` *`S4`*

`IN` *`Sd`*, *`I`*

Copy the byte at the input port into *`Sd`*.

Opcode: `09`

Example: `IN` *`S0`*, *`I`*

`OUT` *`Q`*, *`Ss`*

Copy the byte in *`Ss`* to the output port.

Opcode: `0A`

Example: `OUT` *`Q`*, *`S5`*

`JP` *`e`*

Jump to label *`e`*.

Opcode: `0B`

Example: `JP` *`main_loop`*

`JZ` *`e`*

Jump to label *`e`* if the result of the last `ADD`, `SUB`, `AND`, `EOR`, `INC`, `DEC`, `SHL` or `SHR` was zero.

Opcode: `0C`

Example: `JZ` *`if_zero_branch`*

`JNZ` *`e`*

Jump to label *`e`* if the result of the last `ADD`, `SUB`, `AND`, `EOR`, `INC`, `DEC`, `SHL` or `SHR` was not zero.

Opcode: `0D`

Example: `JNZ` *`if_non_zero_branch`*

`RCALL` *`s`*

Push the program counter onto the stack to store the return address and then jump to label *`s`*.

Opcode: `0E`

Example: `RCALL` *`summing_routine`*

`RET`

Pop the program counter from the stack to return to the place from which the subroutine was called.

Opcode: `0F`

Example: `RET`

`SHL` *`Sd`*

Shift the byte in *`Sd`* one bit left, putting a `0` into the least-significant bit.

Opcode: `10`

Example: `SHL` *`S2`*

`SHR` *`Sd`*

Shift the byte in *`Sd`* one bit right, putting a `0` into the most-significant bit.

Opcode: `11`

Example: `SHR` *`S7`*

Tip: A good way to remember the order of parameters to instructions is to remember that the destination parameter always comes before the source parameter.

4.3. Subroutines

The OCR specification also gives three built-in subroutines, to be called with `RCALL` as if they were any normal subroutine.

Warning

Due to the fact that the subroutines have no simulated in-memory representation, the operand for the `RCALL` instruction is set a little differently when using any of these three subroutines.

- For the `readtable` subroutine, the operand is set to the address of the `RCALL` opcode instance.
- For the `wait1ms` subroutine, the operand is set to the address of the operand itself.
- For the `readadc` subroutine, the operand is set to the address of the opcode following the `RCALL` instruction.

This is the only case where different legitimate code could produce the same in-memory representation. However, any code which would produce this sequence of operands and opcodes would be useless as part of a program anyway.

`readtable`

Copies the byte in the lookup table pointed at by *`S7`* into *`S0`*. The lookup table is a labelled table: when *`S7`* is `0` the first byte from the table is returned in *`S0`*.

In the absense of further guidance, `readtable` reads from memory, and is designed to work with the `$SET` directive, which allows lookup tables to be built easily.

Example: `RCALL readtable`

`wait1ms`

Waits 1ms before returning.

Example: `RCALL wait1ms`

`readadc`

Returns a byte in *`S0`* proportional to the voltage at the ADC. The ADC input can vary from 0V to 5V, which are given values of `00` and `FF` respectively.

Example: `RCALL readadc`

5. About MCUS

MCUS was written by Philip Withnall (<philip@tecnocode.co.uk>). To find more information about MCUS, report a bug or make a suggestion about the application or this manual, please visit the MCUS Web page (http://tecnocode.co.uk/projects/mcus).