Monorail Emulator

Design manual

Contents

[Data Structures 3](#_Toc496954915)

[Two Byte Numbers 3](#_Toc496954916)

[Single-Byte Memory 3](#_Toc496954917)

[Multi-Byte Memory 3](#_Toc496954918)

[Algorithms 4](#_Toc496954919)

[Station Name Entry 4](#_Toc496954920)

[Station Travel Time Entry 4](#_Toc496954921)

[Monorail Running 5](#_Toc496954922)

[Appendices 6](#_Toc496954923)

[Appendix A: Single Byte Data Memory Visualisation 6](#_Toc496954924)

[Appendix B: Multi-Byte Data Memory Visualisation 6](#_Toc496954925)

# Data Structures

The system relies heavily on blocks of allocated data memory to store data for operation. Station names, times between stations, stopping time and several flag variables are stored in data memory.

## Two Byte Numbers

Several operations for the emulation of the system require two byte operations (such as pointers). This is handled simply, through the use of two registers for each number. These numbers are not treated as words (which would allow operations such as *adiw r21:r20*), as this would require that all two byte numbers were manipulated using adjacent registers. Instead assuming this, two byte operations were conducted by performing a one byte operation on the low byte, then on the high byte with carry.

## Single-Byte Memory

Flag variables and single byte integers are each stored in a single byte of data memory. This data is then accessed through load or store direct to data space commands. All operations conducted on or with this data occurs in registers, then the data is stored back in data memory after the operation is complete.

See Appendix A for visualisation.

## Multi-Byte Memory

Station names and timings are stored in multi-byte blocks. Station names are stored in an 10 \* 11 block of data memory, representing 10 strings of eleven characters each (up to 10 letters followed by a terminating character). Before accessing this memory each time the relevant pointer must be aligned to an 11-byte interval to ensure data is read from the correct point, and then read until the trailing terminating character. Station timings are stored in a single 10-byte block (10 \* 1-byte numbers). As these timings are accessed in sequence (First through last byte, always), the only pointer management required is that data in this memory space is accessed with a post-increment on the pointer.

See Appendix B for visualisation.

# Algorithms

See below for pseudo-code explanations of algorithms for various sections of emulations

## Station Name Entry

Entering station names occurs one name at a time, saving each letter as it is entered. Between each station name, the pointer to data memory where the string is stored is aligned to the border of 11 bytes.

*while stationsPreviouslyNamed < numStations:*

*ptr = stationMemoryBlock + stationsPreviouslyNamed \* 11*

*while letters <= 10 && not hashPressed:*

*if backspace && notFirstLetter:*

*ptr = ptr – 1*

*next*

*letter = keypadInput()*

*storeInDataMem(ptr, letter)*

*ptr = ptr + 1*

*storeInDataMem(ptr, nullChar)*

*stationsPreviouslyNamed = stationsPreviouslyNamed + 1*

## Station Travel Time Entry

Station travel times are stored sequentially, so the algorithm uses a simple write to data memory followed by an increment of the pointer.

ptr = timesMemBlock

while timesEntered < numStations:

num = keypadInput()

if num = backspace:

next

storeInDataMem(ptr, num)

timesEntered = timesEntered + 1

## Monorail Running

The monorail loops forever, travelling around its emulated circuit of stations. The LCD display shows the name of the next station, or the station name currently stopped at. When arriving at a station the emulator checks if a button has been pressed (Simulated a passenger wanting to get on or off), and if so stops at the station for the required time. If not, it continues on to the next station, traveling for the relevant amount of time. Software timers are used to handle travel and stop times, hence the inclusion of *countTime()* in the below pseudo-code. Appropriate checks and handling are included for a hash key press (Emergency stop between stations) and the flashing of LEDs to indicate the monorail is stopped at a station.

while true:

for timeToNextStation:

print “*Next Station: …”*

spinMotor()

if hashPressed:

clear hashPressed

while not hashPressed:

wait

clear hashPressed

countTime()

if ButtonInterruptTriggered():

stopMotor()

print “*At Station: …”*

for timeAtStation:

if thirdOfSecond:

flashLEDS()

countTime()

## Keypad Scanner

Scanning the keypad for input was done with a simple scanner, which would check all rows for a signal, and if one was found the columns would be scanned to identify which key was pressed. Depending on which stage the emulation is currently running, the input is then handled accordingly. This code is contained in a loop (Read until end of input) .The pseudo-code for this algorithm is as follows:

if inputReceivedOnRow:

row = rowNumberOfInput

while not inputOnCol:

checkForInputOnCol()

col = colNumberOfInput()

if col < 3:

if row = 3:

input = convertInputToSymbol(col)

else:

input = convertInputToNumber(row, col)

else:

input = convetInputToLetter(row)

handleInputBasedOnStageOfEmulation(input)

For keypad scanning during the running of the monorail, handling is a little different as only the hash key will trigger an action. When the hash key is pressed, the program simply waits for a second hash press, which has the side-effect of stopping all other software timers (Such as timer counting down until reaching the next station). Therefore pseudo-code is as follows:

if inputReceivedOnRow:

row = 3

checkForInputOnRow()

if inputOnRow:

col = 2

checkForInputOnCol()

if inputOnCol:

clearInput()

while not hashInput:

row = 3

checkForInputOnRow()

if inputOnRow:

col = 2

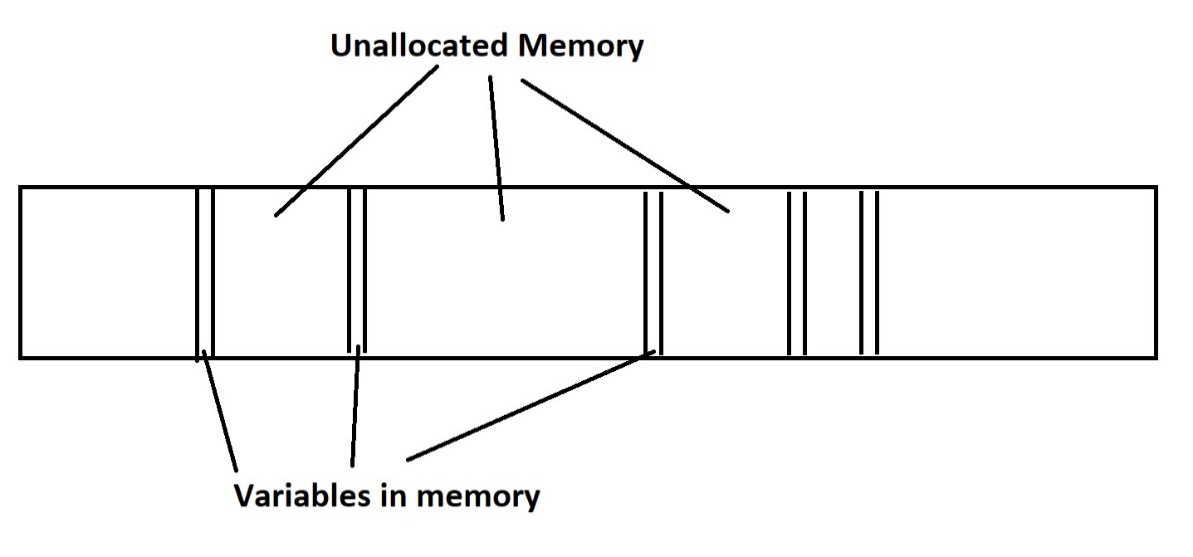
checkForInputOnCol()

if inputOnCol:

hashPressed = true

# Appendices

## Appendix A: Single Byte Data Memory Visualisation



## Appendix B: Multi-Byte Data Memory Visualisation

