Name: Ciaran Coady

Student Number: 17326951

Date: 11/03/2018

Introduction

In this report the process for developing an ARM assembly language program which is capable of solving a given Sudoku grid if it is solvable will be explained in detail. I will also explain the logic behind each part of the program, presenting all testing carried out for each part respectively. The report is broken up into multiple parts:

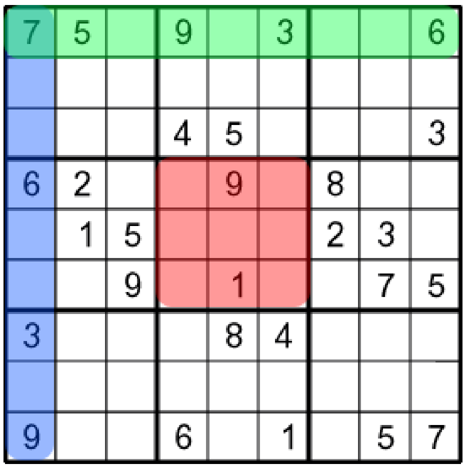
1. **Getting and Setting Digits**
2. **Validating Solutions**
3. **Solving a Sudoku Puzzle**
4. **The Extra Mile**

Before beginning the project I started by familiarising myself with the Sudoku game itself and what it involves. In a game of Sudoku the player is presented with a 9*9 partially filled grid. The goal of the game is fill all the squares with suitable numbers, as per the rules. For a grid to be valid each number must only exist:

1) Once per row (green)

2) Once per column (blue)

3) Once per 3*3 subgrid (red)



If a solution satisfies these three criteria it is deemed valid.

After completing this research and some Sudoku puzzles by hand, I began to develop my solution.

1 Getting and Setting Digits

This section of the program allows us to read a specific cell of the grid or write a value to a cell. The grid itself is stored as a 9*9 2D array in memory.

getSquare subroutine:

Firstly I decided on the interface by which the parameters would be passed to and returned from the subroutine:

R0 = Start address of the grid

R1 = Row of cell being checked

R2 = Column of cell being checked

R3 = Return value in that cell

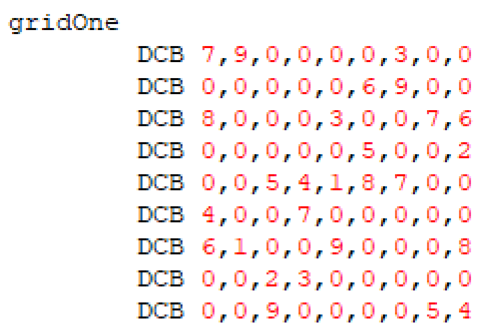
This subroutine must, calculate the correct address in memory of the desired cell and return the byte sized number in the range 1-9 to the calling code.

I then proceeded to develop some sudo code to help me to write the arm assembly language code required (as per all subroutines I push any modified registers and the link register to the stack at the start of the subroutine, popping them off the stack at the end)

```
index = (row * row_size) + column;
SquareValue = memory.byte[gridStartAddr + index];
```

Testing

To test the subroutine I used a sample Sudoku grid stored in memory (gridOne) as below:



```
gridOne
        DCB 7,9,0,0,0,0,3,0,0
        DCB 0,0,0,0,0,6,9,0,0
        DCB 8,0,0,0,3,0,0,7,6
        DCB 0,0,0,0,0,5,0,0,2
        DCB 0,0,5,4,1,8,7,0,0
        DCB 4,0,0,7,0,0,0,0,0
        DCB 6,1,0,0,9,0,0,0,8
        DCB 0,0,2,3,0,0,0,0,0
        DCB 0,0,9,0,0,0,0,5,4
```

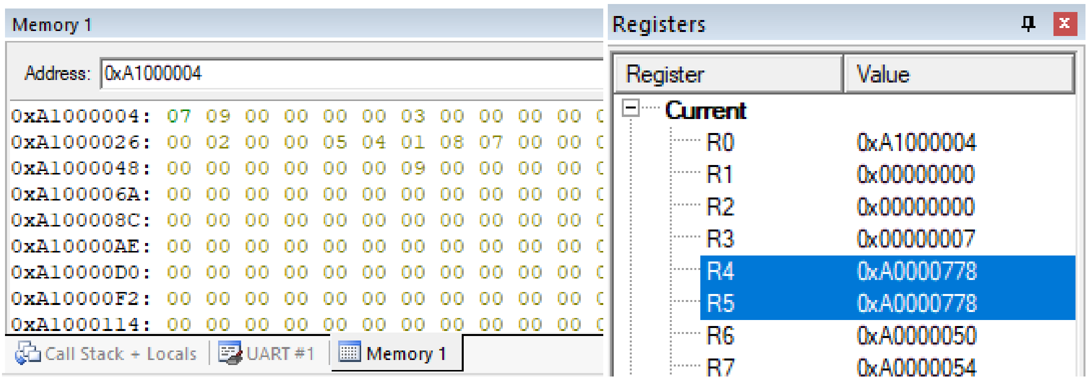
I then placed the start address of the grid in R0 as per my decided interface and conducted the the following test cases:

Test	Row	Column	Expected Value	Outcome
1	0	0	7	7
2	4	2	5	5
3	8	8	4	4
4	0	8	0	0
5	8	0	0	0

To verify the results I stepped through the subroutine for each different test case. I monitored the values in memory using the memory window, and recorded the returned result in R3.

Take test case 1:

After stepping through the subroutine we can see that the correct element was accessed in memory as it is now highlighted green and also 7 has been returned in R3 as we expected.



Address: 0xA1000004

```
0xA1000004: 07 09 00 00 00 00 03 00 00 00 00
0xA1000026: 00 02 00 00 05 04 01 08 07 00 00
0xA1000048: 00 00 00 00 00 00 09 00 00 00 00
0xA100006A: 00 00 00 00 00 00 00 00 00 00 00
0xA100008C: 00 00 00 00 00 00 00 00 00 00 00
0xA10000AE: 00 00 00 00 00 00 00 00 00 00 00
0xA10000D0: 00 00 00 00 00 00 00 00 00 00 00
0xA10000F2: 00 00 00 00 00 00 00 00 00 00 00
0xA1000114: 00 00 00 00 00 00 00 00 00 00 00
```

Register	Value
Current	
R0	0xA1000004
R1	0x00000000
R2	0x00000000
R3	0x00000007
R4	0xA0000778
R5	0xA0000778
R6	0xA0000050
R7	0xA0000054

setSquare subroutine:

This subroutine will allow us to set new values in a specific cell of the Sudoku grid.

setSquare interface:

R0 = Start address of the grid

R1 = Row of cell being set

R2 = Column of cell being set

R3 = Value to be stored in cell

This subroutine must, calculate the correct address in memory of the desired cell and place the byte sized number in the range 1-9 into that slot in memory.

Sudo code

```
index = (row * row_size) + column;
memory.byte[gridStartAddr + index] = newSquareValue
```

Testing

My testing remained similar to the getSquare subroutine, the only difference being that I pass the new cell value into the subroutine.

Test	Row	Column	New Value	Outcome
1	0	0	9	9
2	4	2	5	5
3	8	8	0	0
4	0	8	7	7
5	8	0	1	1

I again followed my previous test methodology using both the memory and register windows to monitor results.

2 Validating Solutions

To validate a potential full or partial solution a number cannot occur more than once in each row, column and subgrid. In light of this, sub-dividing the problem seemed like a good option. In this part the subroutine isValid was developed and as part of its execution it invokes three further subroutines, isValidRow, isValidColumn and isValidSubgrid.

isValidRow subroutine:

This subroutine checks if a passed cell is valid within its row

isValidRow interface:

R0 = Start address of the grid

R1 = Row of cell being checked

R2 = Column of cell being checked

R3 = Returns a Boolean (true/1) or (0/false)

Sudo code

```
boolean isValidRow(byte[][] grid, byte row, byte column)
	{
		byte square = grid[row][column];
		if(square != 0)
		{
			for(byte index = 0; index < grid[0].length; index++)
			{
				if(index != column)
				{
					byte square1 = grid[row][index];
					if(square == square1)
						return false;
				}
			}
		}
		return true;
	}
```

This will compare each element of the given row with the test element, for efficiency it doesn't check 0 cases as we know that they are valid and just returns a true result. An early bug was forgetting to skip the test element itself and returning a false result, but after the first test this bug was removed.

Testing

For my testing I selected very specific test cases to ensure all elements of the subroutine were working as I had intended. I chose a valid case, an invalid case and an empty cell.

Test	Row	Column	New Value	Expected Outcome	Outcome
1	0	0	8	false	false
2	4	2	5	true	true
3	8	8	0	true	true

isValidColumn subroutine:

This subroutine is nearly identical to isValidRow, but it checks columns validity.

isValidColumn interface:

R0 = Start address of the grid

R1 = Row of cell being checked

R2 = Column of cell being checked

R3 = Returns a Boolean (true/1) or (0/false)

Sudo code

```
boolean validColumn(byte[][] grid, byte row, byte column)
	{
		byte square = grid[row][column];
		if(square != 0)
		{
			for(byte index = 0; index < grid[0].length; index++)
			{
				if(index != row)
				{
					byte square1 = grid[index][column];
					if(square == square1)
						return false;
				}
			}
		}
		return true;
	}
```

This will compare each element of the given column with the test element using the same logic from isValidRow.

Testing

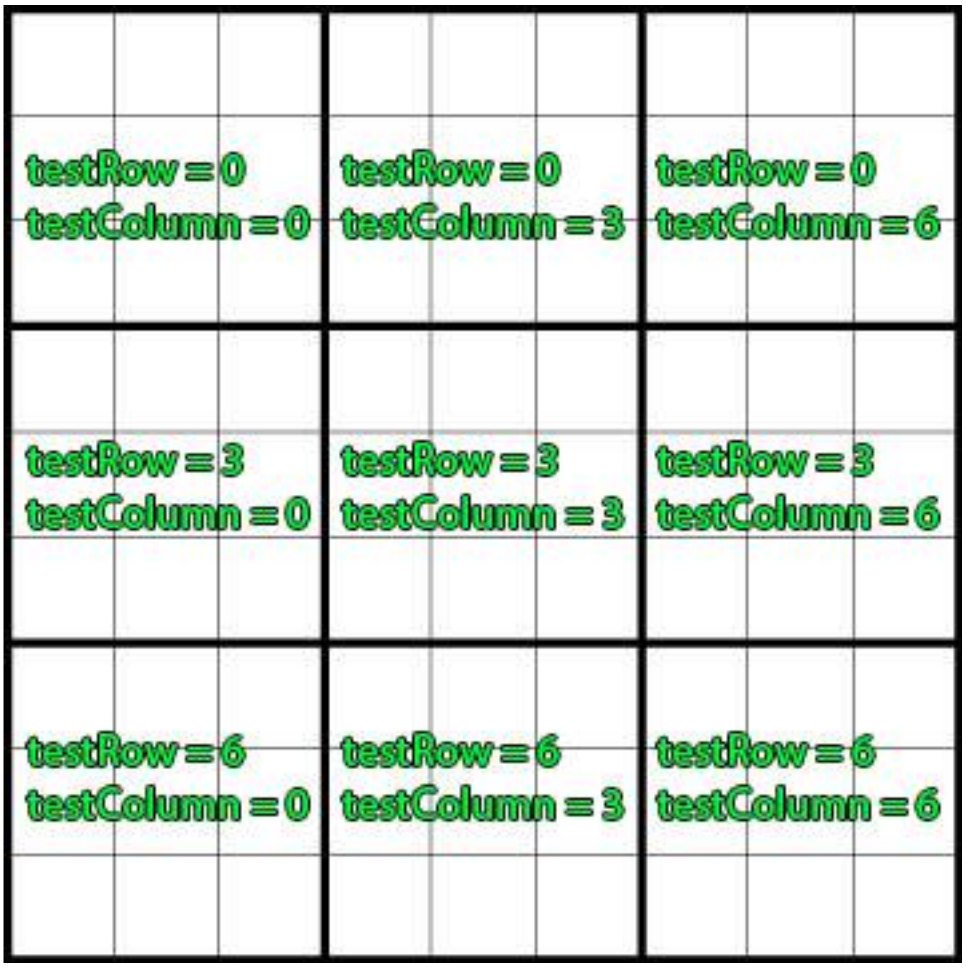
Testing remained identical to what was used for isValidRow

Test	Row	Column	New Value	Expected Outcome	Outcome
1	0	0	8	false	false
2	4	2	5	true	true
3	8	8	0	true	true

isValidSubgrid subroutine:

This subroutine checks if a passed cell is valid within its subgrid.

As part of the implementation of this subroutine a further subroutine identifySubGrid was developed. This identifies which subgrid is being checked and returns a testRow and testColumn denoting the start point of this subgrid.



identifySubGrid interface:

R3 = testCase

R3 = Returns the start of that particular grid

Sudo code

```
byte subGridStart (byte testCase)
            if(testCase >= 6) {
                  subGridStart = 6;
            }
            else if(testCase >= 3) {
                  subGridStart = 3;
            }
            else if(testCase >= 0) {
                  subGridStart = 0;
            }
```

Testing

The testing required for this subroutine was pretty simple as all it was required to do was compare two numbers.

Test	testCase	Expected Outcome	Outcome
1	0	0	0
2	4	1	1
3	8	2	2

isValidSubgrid interface:

R0 = Start address of the grid

R1 = Row of cell being checked

R2 = Column of cell being checked

R3 = Returns a Boolean (true/1) or (0/false)

Sudo Code

```
boolean isValidSubgrid(byte[][] grid, byte row, byte column)
        {
                byte testRow = identifySubGrid(row);
                byte testCol = identifySubGrid(column);
                byte testSquare = grid[row][column];
                if(testSquare != 0)
                {
                        for(byte index1 = testRow; index1 < subGridYStart+3; index1++)
                        {
                                for(byte index2 = testCol; index2 < subGridXStart+3;
                                    index2++)
                                {
                                        byte testSquare2 = grid[index1][index2];
                                        if(index1 != row || index2 != column)
                                        {
                                                if(testSquare == testSquare2)
                                                {
                                                        return false;
                                                }
                                        }

                                }
                                testCol = subGridXStart;
                        }
                }
                return true;
        }
```

Testing remained identical to what was used for isValidRow, isValidColumn

Test	Row	Column	New Value	Expected Outcome	Outcome
1	0	0	8	false	false
2	4	2	5	true	true
3	8	8	0	true	true

isValid interface:

R0 = Start address of the grid

R1 = Row of cell being checked

R2 = Column of cell being checked

R3 = Returns a Boolean (true/1) or (0/false)

This subroutine implements isValidRow, isValidColumn and isValidSubgrid. If any of these return false the result is false, otherwise the result is valid and true is returned.

Sudo code

```
if(isValidRow(row, column) && isValidColumn(row, column) && isValidSubgrid(row, column))
{
  Return true;
}
Else
{
  Return false;
}
```

Testing

To test this subroutine several different test grids were used, both valid and invalid getting the subroutine to test various different cells within each grid.

Test	Row	Column	Expected Outcome	Outcome
testGrid1	0	0	false	false
testGrid1	4	2	true	true
testGrid1	8	8	true	true
testGrid2	0	0	true	true
testGrid2	4	2	true	true
testGrid2	8	8	true	true
testGrid3	0	0	true	true
testGrid3	4	2	true	true
testGrid3	8	8	true	true

After completing the testing I found my code to be inefficient in places e.g. having two separate identifySubGrid subroutines, one for the row and one for the column. I then made changes to improve it and followed the same testing process to evaluate the improved version.

3 Solving a Sudoku Puzzle

For this part of problem we will be implementing all the other subroutines that were developed in parts 1 and 2. I adopted the brute force approach and used the provided sudo code. This works by first checking is a square empty, if it's not we want to skip this square and move onto the next one, otherwise we want to try and fill it with a number from 1-9. Starting at 1 we will check does that number satisfy the row, column and subgrid, if so we will move onto the next square otherwise we will increment our test number and try again. If we cannot fill the square successfully we will backtrack to the last square and reset the current one, this is achievable as every time we move onto the next square we are recursively calling the Sudoku subroutine. This means it is very easy to move back a step to where the mistake was made. If the puzzle is solved, true will be passed back through all Sudoku subroutine calls otherwise the puzzle is unsolvable and false will be returned.

Sudoku subroutine interface:

R0 = Start address of the grid

R1 = Start row of the grid

R2 = Start column of the grid

R3 = Returns true/1 or false/0

Sudo code

```
bool sudoku(address grid, word row, word col)
{
    bool result = false;
    word nxtcol;
    word nxtrow;

    // Precompute next row and col
    nxtcol = col + 1;
    nxtrow = row;
    if (nxtcol > 8) {
        nxtcol = 0;
        nxtrow++;
    }

    if (getSquare(grid, row, col) != 0) {
        // a pre-filled square
        if (row == 8 && col == 8) {
            // last square — success!!
            return true;
        }
        else {
            // nothing to do here - just move on to the next square
            result = sudoku(grid, nxtrow, nxtcol);
        }
    }
    else {
        // a blank square - try filling it with 1 ... 9
        for (byte try = 1; try <= 9 && !result; try++) {
            setSquare(grid, row, col, try);
            if (isValid(grid, row, col)) {
                // putting the value here works so far ...
                if (row == 8 && col == 8) {
                    // ... last square — success!!
                    result = true;
                } else {
                    // ... move on to the next square
                    result = sudoku(grid, nxtrow, nxtcol);
                }
            }
        }

        if (!result) {
            // made an earlier mistake - backtrack by setting
                        // the current square back to zero/blank
            setSquare(grid, row, col, 0);
        }
    }

    return result;
}
```

Testing

For testing I used a number of test grids, each one to test a different scenario. TestGrid1 is an unsolvable grid, TestGrid2 is a solvable grid and TestGrid3 is an empty grid.

TestGrid	Expected Outcome	Outcome
TestGrid1	false	false
TestGrid2	true	false
TestGrid3	true	false

Throughout my testing I discovered that the subroutine had a lot of unexpected behaviour. It would not fill more than one row and it also would not backtrack properly. The cause of this behaviour was that I had incorrectly implemented the if-else statements, where the if branch would be executed it would also execute the else branch as I had omitted the unconditional branch at the end of the if statement. After fixing these problems I tested again using the same test grids and got the expected results.

TestGrid	Expected Outcome	Outcome
TestGrid1	false	false
TestGrid2	true	true
TestGrid3	true	true

4 Extra Mile

For the extra mile section I implemented an output to the console, this way the user could get a graphical representation of the grid before the execution of the Sudoku subroutine and after. The way I designed this was as follows:

Firstly I made a blank array which is essentially the skeleton of the display, consisting of various symbols so that when printed, an easy to read Sudoku board is displayed. This blank board is stored in memory at the start of the program along with any testGrids.

I then developed three subroutines:

storeGridToBoard – this subroutine takes a 9*9 grid of byte sized values in the range 0-9 and places it into the correct cells within the blank board. This board is now ready to be printed.

printBoard –this subroutine takes a 19*19 grid of byte sized values and prints them to the UART window.

printMessage – this subroutine prints if the grid was solvable or unsolvable to the UART window.

Through the use of these three subroutines I decided to print the unsolved board to the console at the start of the program. After the execution of the Sudoku subroutine I then print if the grid was solvable and the resulting board. If the grid is solvable the solution will be printed, otherwise the original grid will be printed.

printBoard subroutine interface:

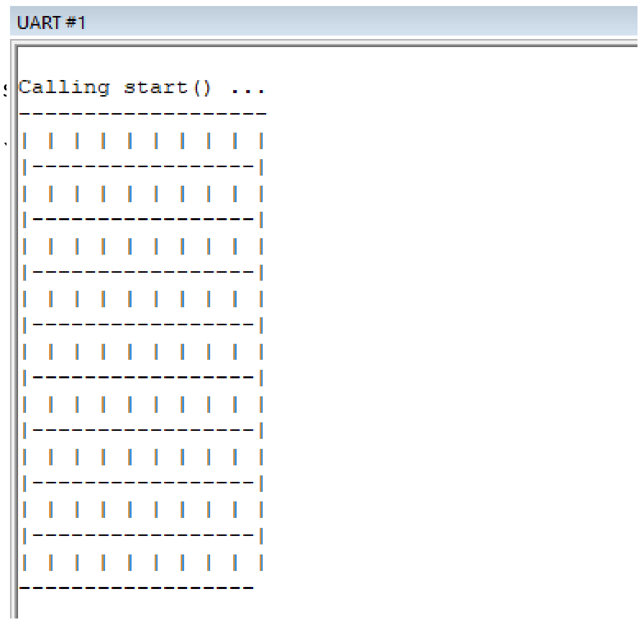
R0 – Start address of the board

Sudo code

```
for(byte rowIndex = 0; rowIndex < board.length; rowIndex++)
            {
                  for(byte columnIndex = 0; columnIndex < board.length;
                      columnIndex++)
                  {
                        toBePrinted = getSquare(rowIndex, columnIndex);
                        sendchar(toBePrinted);
                  }
                  sendchar(”\n”);
            }
```

Testing

The only test required was to see if the empty board I had placed in memory would print, this could be viewed in the UART #1 window. The following output was observed:


UART #1

```
Calling start() ...
-------------------
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
|-----------------|
| | | | | | | | | |
------------------
```

This was the desired result as now the user has a visual representation of the Sudoku board.

storeGridToBoard subroutine interface:

R0 – Start address of the grid to be printed

R1 – Start address of the board to store the grid to

Sudo code

```
void storeGridToBoard(byte[][] grid, byte[][] board){
byte gridRow = 0;
            byte gridColumn = 0;
            for(byte boardRow = 1; boardRow < 18; boardRow+=2)
            {
                  for(byte boardColumn = 1; boardColumn < 18; boardColumn+=2)
                  {
                        if(gridColumn > grid.length-1)
                        {
                              gridColumn = 0;
                              gridRow++;
                        }
                        byte temp = grid[gridRow][gridColumn];
                        temp += 48;//0x30 in hex
                        char extractedNumber = (char)temp;
                        board[boardRow][boardColumn] = extractedNumber;
                        gridColumn++;
                  }
            }
```

Through the use of nested for loops, it is possible to increment through each element of a 2D array. In this case I increment through the grid, taking a number and placing it in the corresponding slot in the empty board.

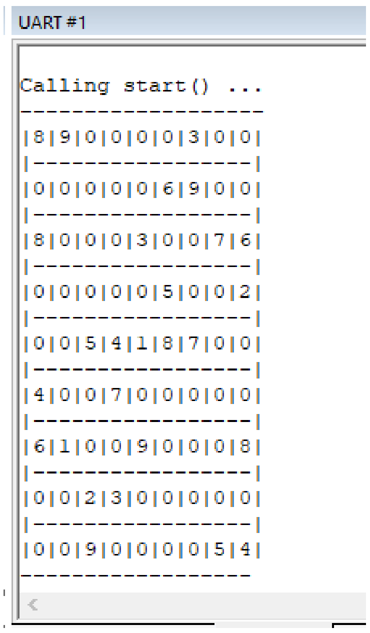
Testing

Having developed and tested the printBoard subroutine I used its console output to debug and test this subroutine. In each test I printed a different board to the console and observed the following results:

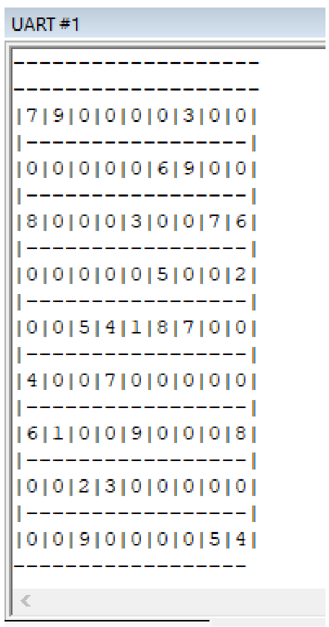
Figure 1 – testGrid1

Figure 2 – testGrid2

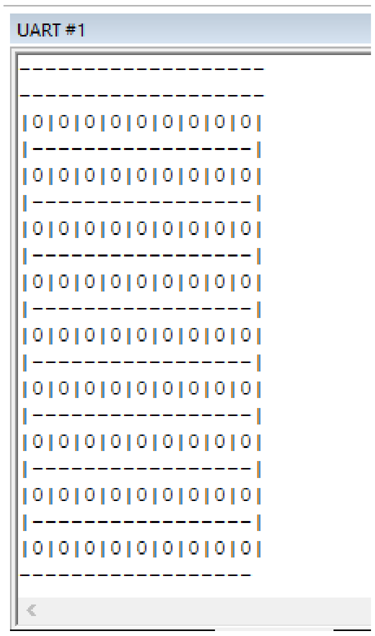
Figure 3 – testGrid3



```
Calling start() ...
-------------------
|8|9|0|0|0|0|3|0|0|
|-----------------|
|0|0|0|0|0|6|9|0|0|
|-----------------|
|8|0|0|0|3|0|0|7|6|
|-----------------|
|0|0|0|0|0|5|0|0|2|
|-----------------|
|0|0|5|4|1|8|7|0|0|
|-----------------|
|4|0|0|7|0|0|0|0|0|
|-----------------|
|6|1|0|0|9|0|0|0|8|
|-----------------|
|0|0|2|3|0|0|0|0|0|
|-----------------|
|0|0|9|0|0|0|0|5|4|
-------------------
```



```
-------------------
-------------------
|7|9|0|0|0|0|3|0|0|
|-----------------|
|0|0|0|0|0|6|9|0|0|
|-----------------|
|8|0|0|0|3|0|0|7|6|
|-----------------|
|0|0|0|0|0|5|0|0|2|
|-----------------|
|0|0|5|4|1|8|7|0|0|
|-----------------|
|4|0|0|7|0|0|0|0|0|
|-----------------|
|6|1|0|0|9|0|0|0|8|
|-----------------|
|0|0|2|3|0|0|0|0|0|
|-----------------|
|0|0|9|0|0|0|0|5|4|
-------------------
```



```
-------------------
-------------------
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
|-----------------|
|0|0|0|0|0|0|0|0|0|
-------------------
```

Figure 1 Figure 2 Figure 3

printMessage subroutine interface:

R3 - bool valid/1 or invalid/0

This subroutine prints a message which has been stored in memory as a null terminated string. The message will either be telling the user that the grid was solvable or that the grid was unsolvable. This subroutine is designed to be called after the execution of the Sudoku subroutine, as a true or false value will be stored in R3.

Sudo code

```
if(gridSolved)
{
        for(int index = 0; letter != 0; index++)
        {
                letter = memory.byte(gridIsSolvable + index);
                printf(letter);
        }
}
else
{
        for(int index = 0; letter != 0; index++)
        {
                letter = memory.byte(gridIsNotSolvable + index);
                printf(letter);
        }
}
```

Testing

I tested this subroutine by running the Sudoku subroutine with various different grids, both solvable and unsolvable resulting in the following:

Test grid used	Expected output	output
testGrid1	"This grid is unsolvable”	"This grid is unsolvable”
testGrid2	"This grid is solvable”	"This grid is solvable”

This concluded all my testing for the program and I am happy that it is functional and works for all of my test cases.