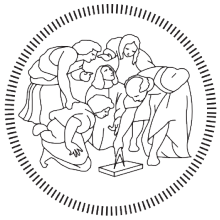
John Conway's Game of Life running on the Miosix embedded OS on a STM32 micro controller

Project Report

Author
Simone Giampà



POLITECNICO
MILANO 1863

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Contents

1 Project Data **2**

2 John Conway's Game of Life on Miosix **2**
2.1 Description . . . 2
2.2 Miosix embedded Operating System . . . 2
2.3 Importance for the AOS Course . . . 2

3 Program structure and description **4**
3.1 Design . . . 4
3.2 Implementation . . . 4
3.3 Concurrency and Multi-Threading . . . 5
3.4 Program execution . . . 5

4 Project results **7**
4.1 Concrete results . . . 7
4.2 Learning opportunity . . . 7
4.3 Existing knowledge . . . 8
4.4 Problems encountered . . . 8

5 Honor Pledge **9**

1 Project Data

- Project supervisor: Professor Federico Terraneo
- Project developer:

First and last name	Person code	Email address
Simone Giampà	10659184	simone.giampa@mail.polimi.it

- Project Repository: Available on Github
  I personally worked on the entirety of the repository contents, and wrote all the code and project report myself.
- Target platform used for this project: STM32 Nucleo 64 board.

2 John Conway's Game of Life on Miosix

2.1 Description

The Game of Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. It is Turing complete and can simulate a universal constructor or any other Turing machine. [1]

Many patterns in the Game of Life eventually become a combination of still lifes, oscillators, and spaceships; other patterns may be called chaotic. A pattern may stay chaotic for a very long time until it eventually settles to such a combination.

The Game of Life is undecidable, which means that given an initial pattern and a later pattern, no algorithm exists that can tell whether the later pattern is ever going to appear. This is a corollary of the halting problem: the problem of determining whether a given program will finish running or continue to run forever from an initial input.

2.2 Miosix embedded Operating System

Miosix is a real-time operating system designed to run on 32bit micro controllers. It is designed to support C and C++ standard libraries and the standard POSIX API. Its kernel code size is very small (10KB on Cortex M3), and this allows to exploit almost the entirety of the flash memory for user applications.

A big advantage of Miosix is that the user-space is optional: one can develop applications directly in kernel-space. Consequently, Miosix is POSIX-compliant also in kernel-space. It is also used as a platform for distributed real-time systems research. [2]

Another advantage is the presence of thread safe standard C++ libraries and C++ language features, such as exception handling. Not all functions and functionalities are implemented though.

2.3 Importance for the AOS Course

This project was really useful for putting in practice and learning two main concepts:

- **Advanced command line usages**
  This program customizes the terminal configuration, enabling the raw mode (non canonical) and disabling the ECHO. This functionalities are useful when designing a user-friendly application

[1] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
[2] https://miosix.org/wiki/index.php?title=Main_Page

that runs on a terminal. This allows to interpret every single characters (or a block of characters) from the user input, in order to simplify and make the user experience better.

- **Concurrent and multi-threading programming in C++**
  Some basic functionalities for concurrent programming have been implemented, using the C++ standard libraries. Since the code runs on the Miosix embedded OS, not all the standard functions are implemented, thus complicating the development with respect to a standard Linux implementation. For example, timed condition variables and timeout mechanisms are not available on this Linux - based OS.

3 Program structure and description

3.1 Design

The Game of Life structure consists in a matrix representation of a cellular automaton. When the program starts, calculates the dimension of the terminal on the host machine. Then the size of the matrix is calculated accordingly, so that it spans the entirety of the terminal. Each cell is represented by a square surrounded by line borders. Each cell is a boolean flag representing its state (living or dead). Every cell state evolves over time based on a predefined set of rules, that depend on the state of the surrounding cells (Moore neighborhood).

The simulation finishes in three cases:

- If the user presses the `q` command, the program immediately terminates.
- If at a certain point in the simulation the matrix is completely empty, the program recognizes that it's an "empty" life and terminates.
- If at a certain point in the simulation the cellular automaton life is "still", that means that it's not going to evolve over time, the program recognizes that it's a still life and terminates.

It is not possible for the program to automatically terminate if the cellular automaton is representing an "oscillating" life. This is because of the intrinsic undecidability of the Game of Life. There is no program that can tell if it's a life that continuously oscillates between repeated states and is going through a cycle (unless it's already known to be a cyclic pattern).

3.2 Implementation

The terminal is configured by using the `termios` library and data structure [3] in this way:

```
1  //checks whether the file descriptor refers to a terminal
2  if (!isatty(STDIN_FILENO)) {
3      std::cerr << "Standard input is not a terminal.\n\r";
4      return 1; // failure
5  }
6
7  /* Save old terminal configuration. */
8  if (tcgetattr(STDIN_FILENO, &oldConfig) == -1 || tcgetattr(STDIN_FILENO, &config) == -1) {
9      std::cerr << "Cannot get terminal settings: %s.\n\r";
10     return 1; // failure
11 }
12
13 // non-canonical mode activated with ~ICANON
14 // ~ISIG implies reading some special terminating key combinations to be read as normal input
15 // ~ECHO does not echo out the input characters
16 config.c_lflag &= ~(ICANON | ISIG | ECHO);
17
18 config.c_cc[VMIN] = 0; //minimum number of characters for canonical read
19 config.c_cc[VTIME] = 1; //timeout for non-canonical read = 100ms
20
21 // if the custom settings for the terminal cannot be set, it resets the default configuration saver
   ↪  previously
22 if (tcsetattr(STDIN_FILENO, TCSANOW, &config) == -1) {
23     tcsetattr(STDIN_FILENO, TCSANOW, &oldConfig);
24     std::cerr << "Cannot set terminal settings: %s.\n\r";
25     return 1; // failure
26 }
```

[3] https://en.wikibooks.org/wiki/Serial_Programming/termios

The reader thread function runs this loop as long as the user doesn't input the quit signal:

```
char input[4] = "";
// reader thread stops when reads the exit command
while (input[0] != 'q' && !terminate) {
    // it should not wait for user input if the queue contains the quit signal

    ssize_t data = read(STDIN_FILENO, input, sizeof input);
    if (data > 0) {
        // accepted input commands
        if (input[0] == 's' || input[0] == 'r' || input[0] == 'q' || input[0] == 'f' || input[0] ==
        ↪  'S' || input[0] == 'R' || input[0] == 'Q' || input[0] == 'F') {
            this->put(input[0]); // adds command to the list
        }
    }
}
```

The synchronized queue is composed of two methods: `put` and `get`. The latter should not be called if the queue is empty, because it is programmed to wait for a new element to be inserted if it's empty. The queue uses a mutex lock and a condition variable to operate.

```
/* Synchronized Queue, using FIFO policy */

char Controller::get() {
    std::unique_lock<std::mutex> lock(mutex);
    while(actionsQueue.empty()) {
        listCondition.wait(lock);
    }
    char result = actionsQueue.back();
    return result;
}

void Controller::put(char action) {
    std::unique_lock<std::mutex> lock(mutex);
    actionsQueue.push_front(action);
    listCondition.notify_one();
}
```

3.3 Concurrency and Multi-Threading

The concurrency management comes into play between a frame and the next one displayed. The user can input commands at any time, while the simulation is going on. After finishing printing a frame, the main thread checks if there is any user input to be processed. This way the main thread executes the commands. There is a maximum amount of time that can elapse between a frame and the other one, in which the user can send other commands. The user can choose to momentarily pause the simulation though, and resume whenever they want.

There is another thread which purpose is to read the user input and insert it into a synchronized queue. The queue follows the FIFO policy, so that the main thread can process the input in the same order it was received by the program. The reader thread executes as long as the main thread is running, so they are running in parallel. The reader thread stops when receives the quit signal from the user, and the main thread joins it when it processes the relative command in the input queue.

3.4 Program execution

The first phase of the program simulation is the setting up of the initial state of the cellular automaton. So the user is asked to place the living cells in the cellular automaton. When the user starts the

simulation, the cellular automaton evolves over time according to the Game of Life's rules.

The output of the program can be seen via a terminal emulator. A suggested utility is *screen*. The output of the program is sent via serial communication via the mini-usb cable connected to the board. The program is executed entirely on the micro controller, and this means that the code on the micro controller cannot run Linux specific functions, for example interacting with the terminal is not possible (because it's just a visual interface and not an actual command line).

The program can be executed this way:

1. Firstly the program source code needs to be copied to the top directory of the Miosix kernel path.
2. The program can be compiled using the `makefile` configuration provided with

   ```
   make
   ```

3. Flash the kernel to the STM32 micro controller using the command

   ```
   st-flash write main.bin 0x08000000
   ```

4. Open the screen utility with

   ```
   screen /dev/ttyACM0 460800
   ```

   and change the baud-rate according to the configuration file.

5. Exit the screen utility with the keys `Ctrl+A -> k -> y`.

4 Project results

4.1 Concrete results

The simulations can be run on a limited size matrix, which limits are established by the terminal size and font. This means that a pattern cannot grow indefinitely. Here there are a few screenshots from the program.

Figure 1: Initial game prompt with tutorial

Figure 2: Initial game setup with 2 spaceships and a glider

4.2 Learning opportunity

The project source code is entirely written in C++. I didn't have any prior coding experience with C++, so this project enabled me to learn and practice C++ programming. Furthermore, I put in

Figure 3: An example of a frame from another simulation

practice the acquired knowledge about concurrent and multi-threading programming in a real time embedded system environment, which is a very useful field to have experience with. I'm really interested in working in the embedded systems field, and this project showed me that I would like to pursue this career field, even though I found that it's quite difficult to approach, especially for novices.

4.3 Existing knowledge

I attended the ACSO course (Computer Architectures and Operating Systems), which gave me the necessary knowledge and understanding about how operating systems work, so that made it possible to attend the AOS course and complete this project. My prior knowledge of low level C programming and MIPS Assembly proved to be fundamental in order to comprehend the topics of the AOS course. Also reading the source code of the Miosix kernel (which was necessary in some cases in order to solve some bugs) required a deep understanding of some of the course topics covered, especially the concurrency and multi-threading part.

A quick introduction on how to use all the terminal features would have been useful though. The `termios` library contains a lot of parameters, that, if used correctly, enable the programmer to create complex GUI-looking command-line utilities and programs. At first, it was not really easy to understand how it works.

4.4 Problems encountered

Several problems have been encountered during the development process. At first, the command line wasn't responding like it was expected. Afterwards a lot of compilation errors often popped up due to the fact that the Miosix OS doesn't implement every function in the standard libraries.

- **Command line malfunctions**
  The biggest problem that I faced was that new lines didn't reset the cursor position at the start of the line. Basically carriage returns weren't working. It turns out it was neither an input or output problem, because there is no flag in the `termios` data structure that could fix the problem. I tried setting every possible flag combination, and I eventually gave up when I realized that it was due to the raw (non canonical) mode. I managed to get carriage returns successfully by adding the character "`\r`" at the end of every printed string, alongside the character "`\n`".

- **Timeouts and timed condition variables**
  Timed condition variables are not currently implemented on the Miosix OS. I needed to apply a timeout on a function that was reading user input, but the function that reads from a file descriptor is a thread-blocking function, thus there is no way of waking up from the I/O wait after a certain time elapses. No alternative methods that were making use of the `termios` library were working.

  Also, the `select` function didn't work because the corresponding library is not implemented in the Miosix OS. I eventually solved the problem by changing the way the main thread interacts with the reader thread, and by making the input management function non blocking.

- **Slow baud rate**
  The standard baud rate was too slow to properly see how the cellular automaton evolved over time, because the entirety of the terminal needed to be rewritten at each displayed frame. This problem was solved by increasing the baud rate to 460800, which proved to be the highest working baud rate. This way, the output is much smoother and the terminal refreshes faster. Even though it is still not as fast and smooth as the Linux command line executed on a pc, this is the fastest achievable speed via mini-usb serial communication.

5 Honor Pledge

I pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents my original production, unless otherwise cited.

I also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the exam of September 2022.

Simone Giampà