Ben-Gurion University
Electrical and Computer Engineering
Project Report
Functional Programming in Distributed Systems

Elias Assaf - 315284729
Ron Zilbershtein - 205996929

August 20, 2023

Contents

1 Project Proposition and Motivation

Implementing a fault tolerant distributed and dynamic sensor network, composed of MSP430 MCU's that act as our sensors, and multiple erlang nodes that act as the controllers, gui and supervisors to the MCU's.

2 PC Architecture

We propose the following architecture:

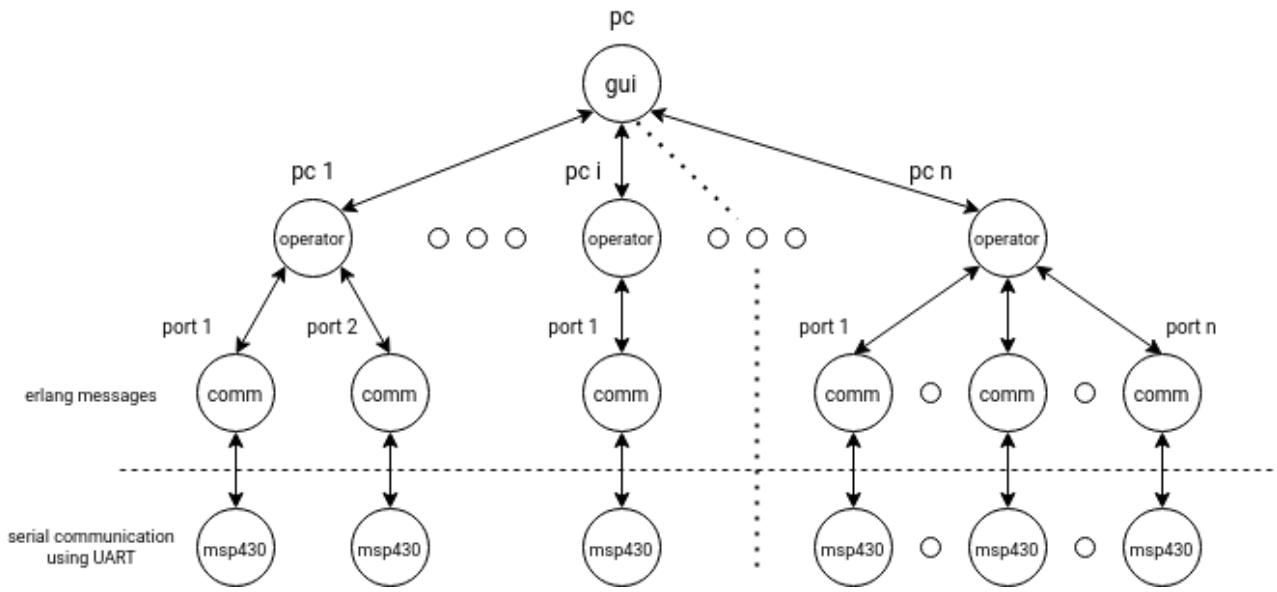

Figure 1: Project Architecture

2.1 GUI

The GUI is a gen_server and our largest module, it is responsible for handling user inputs, wx events and to show the sensor data. Additionally it handles nodes disconnecting and deletes the radars belonging to it, and saves their location to a dets table for persistence even across different instances of the application.
To handle wx events efficiently we added a callback specific to wx events dubbed handle_event/2, and so we treat them as a separate event as the cast/info/call events, which really simplified the design of the gui since we treated the events as part of the OTP stack; We took the idea from the wx_object behavior, but using it we cannot create a gui with a global name since in wx_object behavior we need the local environment that exists only on one node. We made extensive use of the continue callback of the gen_server to abstract away certain events, we pass as argument a list of atoms or tuples that specify side effects (redrawing the background, logging) or mutation to the state (removing old samples, incrementing counters) that will happen after we finished dealing with the current event, but before we accept any other event, we used the following scheme:

```
handle_continue(Continue, State) when is_list(Continue) ->
  NewState = lists:foldl(fun do_cont/2, State, Continue),
  {noreply, NewState};

handle_continue(Continue, State) ->
  {noreply, do_cont(Continue, State)}.
```

And do_cont always returns an updated state.
Using this scheme we really simplified handling general events, and we only need to change the implementation of a event in a single place, which allowed us to do some optimizations, or use a better method without refactoring much of the code, for example we first saved the statistics of the system in a record inside the state, but that was not very efficient since we need to update the whole state to only increment a counter, we decided to move to an ets, where we only changed the implementation of do_cont and the correct field in the state.
The GUI also has an application and supervisor module that start and supervise it in case of an error, and to handle exiting gracefully whenever we close the application.
To benchmark the full performance of our application (in terms of Erlang concurrency and message handling), we also added

an option to disable the graphical redrawing in the GUI by setting the default value of 'draw_samples' to false inside the definition of the state record.

2.2 Operator

Also a gen_server, its job is to manage the communication ports that talk to the msp430's, by spawning them, formatting the messages related to them, and informing the GUI for new connections/disconnections that happen dynamically, additionally it can buffer the samples received from the communication ports, and send the results in batches to lighten the load on the network communication (send in one big batch instead of 100 individual samples).

Connections and Disconnections To monitor new devices connecting/disconnect we turn to the linux kernel for help, when a new serial device such as the MSP430 connects, the kernel allocates a new file for it in the "/dev/serial" directory that includes it's name, and anything we write to that file gets sent to the MSP430 using the UART (RS232) protocol which we will explain about in 2.3.1.
We use the inotify subsystem of linux to receive updates on filesystem changes related to the serial ports (instead of polling), for that we used the inotify open source project which we modified to work with recent versions of rebar3, it uses gen_event and a C port to call a specific callback function whenever a filesystem event that we subscribed to happens, i.e we first spawn the inotify server and gen_event, we watch the "/dev/serial" directory for newly created files, whenever that happens the C ports sends a message to the inotify module in Erlang which calls our module (operator) callback function which sends a gen_server cast to our operator, with that we spawn a new communication port; we note however that this is not entirely accurate since when there are no ports connected the "/dev/serial" folder doesn't exist and we can't monitor it, so our implementation first looks at "/dev" for creation of the "serial" directory and then watches "/dev/serial" for new connections.
Disconnections happen automatically as when the devices gets removed from the USB port, the C port file descriptor of the MSP430 gets an EOF, exits, which in turns sends a message to the Erlang process that itself exits, and then to our communication state machine which also exits, and finally to the operator which sends a message to the GUI that the specific radar exited.
We note that disconnecting the simulated radars happens a little bit differently, since there is no C port reading from it's file, we need to monitor the filesystem for when the file gets deleted, and so again we use the inotify module that we added to monitor for the deletion of the file that describes the specific radar, i.e when the simulated radar gets an inotify event that it's file got deleted it closes itself gracefully, which gets caught again by the operator and passed to the gui.
Since our Operator is dependant on the inotify modules, we set up a supervisor with rest_for_one restart strategy, where if one of the inotify modules fall, we crash the operator, then restart them in the correct order (inotify then operator).

buffering We implementing buffering at the operator and gui levels, so buffering is invisible to the communication layer, we also need to deal with buffered samples having correct time, even as we send them back to the gui which could reside on a different pc.
For each sample that arrives to the operator, we timestamp it using erlang:monotonic_time, and when it time to send them we change the timing to be an offset of the first sample (for each individual communication process), for example if sample 0 arrived at time 15 and sample 1 arrived at time 50, we send that sample 0 arrived at time 0, sample 1 at time 35, this way the gui can use it's own timing system and consider the offsets of the arrived samples.
To enable buffering we call:

```
operator:set_send_rate(true, Time).
```

from the operator shell, where Time is the time (in milliseconds) between sending 2 batches. To disable we call:

```
operator:set_send_rate(false, _).
```

Where _ is any Erlang term.

2.3 Communication PC Side

2.3.1 Real Ports

A simple State Machine using gen_statem with state function callback mode that implement a simple stop and wait protocol to communicate to the msp430 using the RS-232 (UART) protocol, to ensure reliability and correctness of the communication. Note that we only use stop and wait when sending messages to the msp430 (to receive acks), not when we are receiving information from them.
The UART specifies speeds at which the communication occurs, we chose 9600 bits per seconds as it is widely used, has low error rate and is fast enough for our purposes, we used an open source serial port module with our own modifications to read only 1 byte at a time, that uses Erlang ports to spawn a C process that uses termios to write to the MSP430 at the correct speeds, and communicate with the rest of our Erlang application using messages.

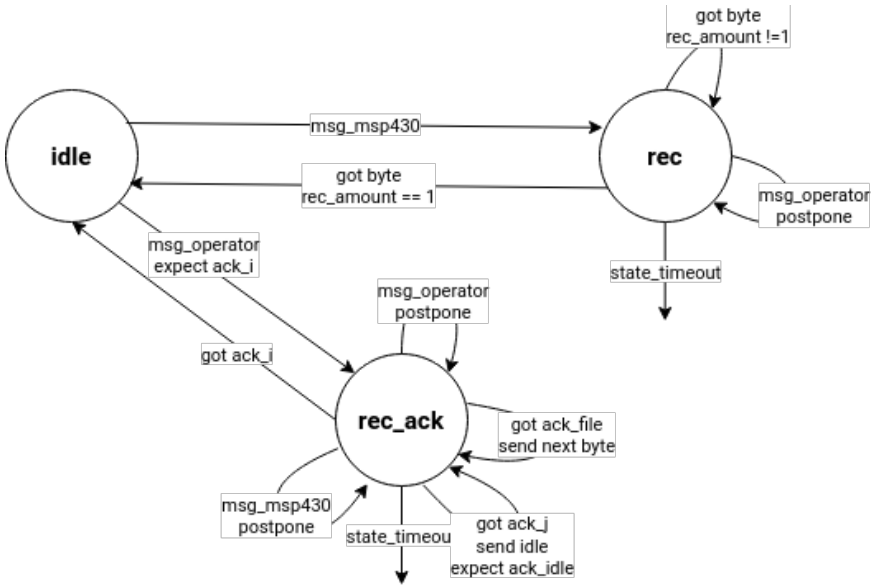

Figure 2: PC Side communication state machine

We leverage Erlang gen_statem postpone transition action, that delays certain events and retries them whenever we enter a new state, for example if we are in the middle of sending a command to the msp430 and are in rec_ack state, and we receive a new command to send to the msp430, we delay that until we return to the idle state, this ensures stability of the msp430 as we do not send it information when it is not ready to receive it.
Since communication can fail on the serial link we need to have a fail-safe, we get the following cases:

- Unexpected ack from the MSP430.
  That means that it received a wrong command or the communication port restarted, so we try to reset it to idle state until we succeed, or it an happen when we reenter a state, for example we finished a 180 degree scan and restarted the state, we receive a new ack, which we discard (or use for debugging purposes).

- Not responding (timeout).
  Should not happen in normal stances, because if we disconnect the MSP430, the port also closes which sends an exit signal to our process, if it does happen it probably means that the MSP430 hanged somewhere, solutions include trying to reset the msp to idle mode, crash the communication port and mark the device as broken, or simple ignore the timeout.
  We opted to ignore the timeout this is a rare event that we have not yet seen.

2.3.2 MSP Side (not relevant to the course)

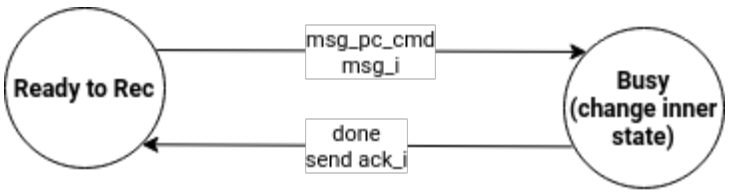

Figure 3: MSP430 Side receive communication state machine

In Fig. 3 we describe the receive side of the MSP, it a very simple FSM, we only receive a new command, turn off interrupts, and enter the new state, then we re-enable the interrupts and sleep.
We note that this approach also works for sending the script file, we first send the cmd from pc to the msp that we are about to send a file (the cmd also includes the file size), and on each acknowledge we send the next byte of the file.

In Fig. 4 we describe the sending state machine, we have a ring buffer with size of 32 bytes, whenever we have a new sample/ack to send, we format the msg and add it to the buffer, and the TX interrupt will send it whenever it can.
A problem we get that the buffer might become full, in the case of sending a sample, we simply discard it, but an acknowledge is important, so we poll the TX busy bit until it is done sending, and we add the acknowledge byte into the buffer, ensuring that we always send the acknowledge.

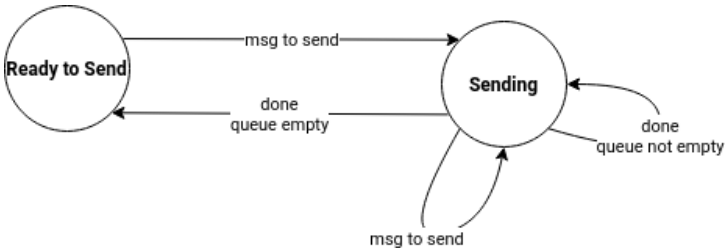

Figure 4: MSP430 Side send communication state machine

2.4 Imaginary Ports

A gen_statem that simulates the MSP430 behavior by using handle_event_function callback mode (that uses a single callback function), we opted to use this callback mode as we have general events (change states) that have the same behavior regardless of the state that we are in, so using the handle_event callback we only need to write 1 function to deal with each event, unlike state_functions callback mode where for each different state we need to deal with each state transition, so for N states we would get N^2 functions that only handle state transitions.

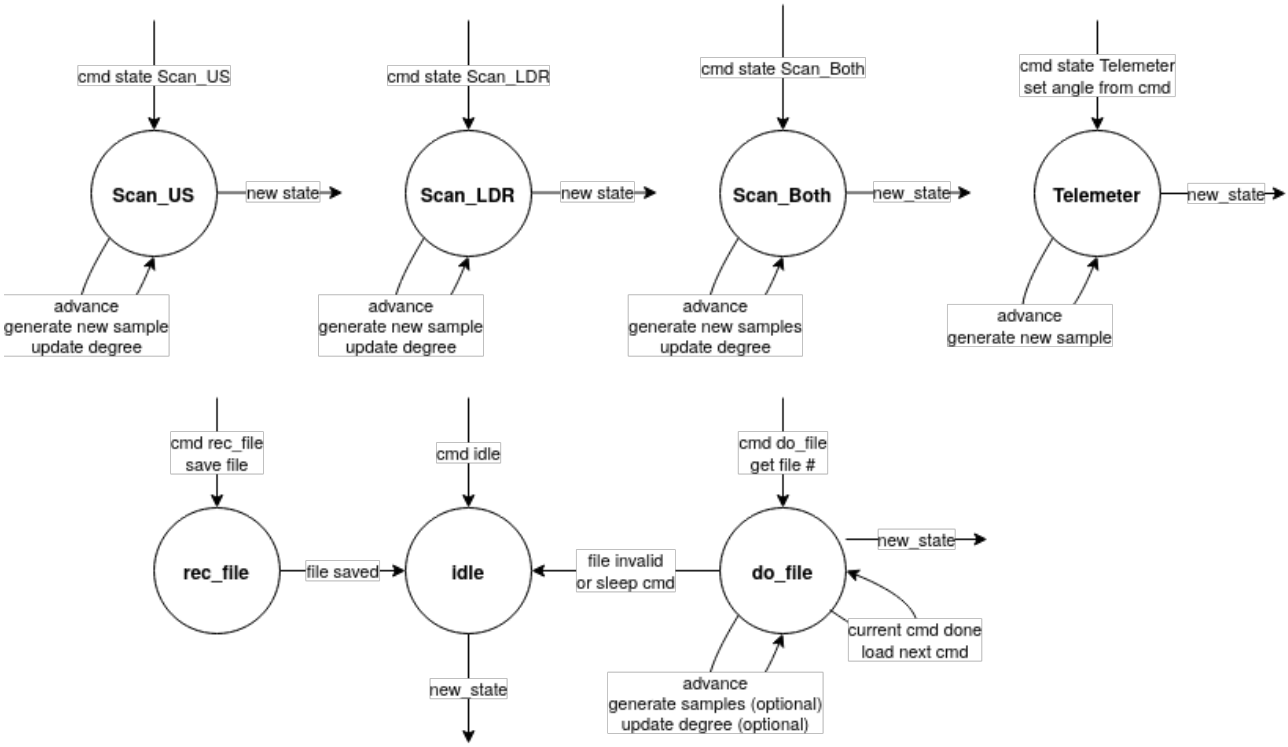

Figure 5: Imaginary communication state machine

To generate the samples, we first set the intensity to low/medium/high so that we can benchmark the performance of our application and find the bottleneck, Additionally every once in a while we change the average distance of the generated sample to simulate a real environment.

3 Benchmarks

3.1 Test Enviroments

To benchmark the results we tested on 2 environments on the lab computers, the first being the GUI and one operator node on the same PC, to minimize the effects of network resources, the second we ran in distributed mode on 4 different computers (each having the same number of connected radars), We also ran the tests with graphical environment enabled to measure it's bottleneck, and without graphics to measure the erlang communication bottlenecks, and finally the effect of caching the results at the operator before sending them, when we enabled caching we sent batches every 500 ms.

3.2 Results

- **Single PC, Graphics**: handles around 400 radars without caching enabled, 1200 radars with caching.
- **Single PC, No Graphics**: handles around 900 radars without caching enabled, 2400 radars with caching.
- **Distributed, Graphics**: also 400 radars without caching, 1000 radars with caching.
- **Distributed, No Graphics**: also 800 radars without caching, 2000 radars with caching.

3.3 Conclusions

We can clearly see that the graphics is the main bottleneck, the largest obstacle that in our implementation on each frame we redraw the whole canvas, and not just the updated samples, we can get better performance using a layered approach to graphics, where we draw different layers separately (background, radars, samples).
Secondly we see that without caching we get many small events that the GUI needs to update its internal state on between each frame drawing, slowing performance.
Finally we can see that in Distributed mode we get a slightly worse performance, due to the added overhead of the networking layers.
Overall this is an excellent performance, since in a real world scenario samples would be much more scarce, and the number of connected devices is significantly lower.

4 Message Formats

First we define the following notation: <<ID:2, Opcode:14>>
Which is a series of bits, where ID is of length 2 bits, Opcode has 14 bits, overall size is 16 bits or 2 bytes.

4.1 PC (Communication) to MSP430

<<ID:2, Opcode:6>>
ID can be:

- 0:
  Opcode is the operation code (state).
- 1:
  Next state is telemeter, Opcode $\in \{0, ..., 60\}$ is the angle/3.
- 2:
  Expect to receive a new file, Opcode is the length of the file in bytes, upto 60.

Here we compressed the data a little bit to allow sending the whole command in 1 byte, and so the angle for the telemeter state is limited to a resolution of 3 degrees (we send 0-60 and the msp multiplies by 3), the length of 1 file is limited to 60 (in reality 64) bytes which is the size of a flash data segment in the MSP, in other words each file will take 1 of the flash data segments, which is at most 20 commands (longest command is 3 bytes).

4.2 MSP430 to PC (Communication)

<<Event_Type:2, Degree:6>>

- 1:
  UltraSonic, Degree is the degree the source was detected, 2 bytes will follow that are the echo high level time (in clock ticks), which we compute the distance with.
  So here we deferred the computation of the distance to the PC, since we need to multiple by 17000 and divide by the clock speed of the MSP, and we don't have that kind of bit resolution in the MSP, this puts the constraint that the PC side must know the clock speed of the MSP, since we are working with a constant clock speed this is fine.

- 2:
  LDR, Degree is the degree the source was detected, 1 byte will follow which is the distance in cm. Here we let the MSP do the needed calculation, because we took empirical samples of the light levels in the room, and let the MSP calibrate itself every once in a while, so whenever we take a new sample, we compare with our empirical data and we take the closest point as the truth and send that to the PC, since we have the data pre-computed we only need to find the minimum distance between the current sample and the data which is computationally inexpensive so we let the MSP do it.

- 3:
  Ack, Degree is the number of state received that we are acking. We also used this for debugging purposes, i.e sending acks that don't match any state that we can see on the PC side as they arrive.

5 Appendix

5.1 Running the project

We recommend using the instructions (same as here) inside the README.md of the repository since there we can copy and paste correctly.
Our project will only run on linux machines since we used the inotify subsystem which only exists on linux (there are equivalents in mac and other UNIX-like OS's but not implemented in the module we used).
We also need a modern Erlang installation with rebar3 and wxWidgets (tested with Erlang/OTP 24-26).
Running the project:

```
git clone https://github.com/EliasA5/radar
cd radar
make radar
```

which will open the gui in distributed mode, to connect operators to it we need to change the RADAR_IP field inside the Makefile, to the one given in the shell that opened the radar, i.e in the shell that we ran 'make radar':

```
Erlang/OTP 25 [erts-13.2.2.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:ns]

Eshell V13.2.2.2  (abort with ^G)
(radar@10.0.0.24)1> ===> Booted radar
```

We set RADAR_IP in the Makefile to:

```
RADAR_IP := 10.0.0.24
```

then we simply run in another shell:

```
make operator
```

and of course we can take the commands from the Makefile and run them manually to give the radar/operators different names.
To generate new imaginary radars, we create a new directory called "dev/" inside the project root directory (created automatically after starting the operator), and inside it we create new files with the names "radar_[0-9a-z]+", and to remove the radar we simply delete it's file from this directory, for example:

```
.../radar> cd dev
.../radar/dev> touch radar_12hi # creates a new radar
.../radar/dev> touch radar_{000..100} # creates a 100 new radars
.../radar/dev> rm radar_12hi # remove the radar
.../radar/dev> rm radar_* # remove all radars
```

5.2 Links

Radar (our project): https://github.com/EliasA5/radar
Inotify Module: https://github.com/EliasA5/inotify
Erlang-Serial: https://github.com/EliasA5/erlang-serial
Ours forks of the projects with some modification to fit our needs.

5.3 MSP430 Architecture

Here we will talk about the architecture of the MSP430, this is not related to the project, but we thought to include it at the appendix for curiosity.
Our system is composed as a state machine, where the transition between each 2 states happen in an interrupt-free context, as to not receive a new state transition in the middle of transitioning a state, additionally, each time we enter a new state, we send an acknowledge to the controlling PC, so it can time it's communication and choose when to send the commands.
The state machine is described as follows:

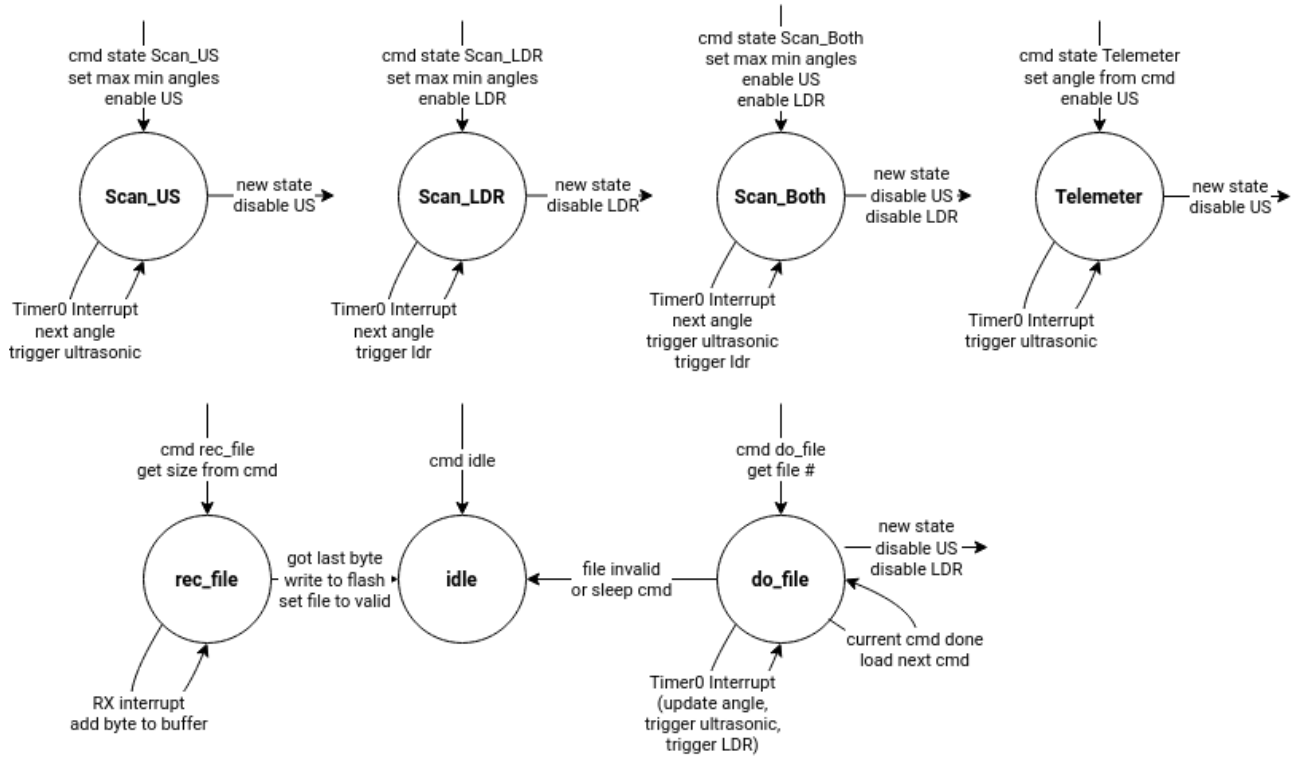

Figure 6: MSP430 state machine

Note that we can move between any 2 states, and even from a state to itself.
Whenever a new sample is ready from the Ultrasonic or LDR sensor either the HW timer or ADC interrupt will go up and it's corresponding ISR will add the sample to the sending msg queue to be sent later, and so our sensors data and sending capabilities are independent of each other, which gives us simplicity in implementing our states.

5.3.1 Script Mode

For script mode we defined the following struct to handle the files:

```
const uchar *const segments[3] = {(uchar *) SEGMENT_B,
                                  (uchar *) SEGMENT_C,
                                  (uchar *) SEGMENT_D};
struct file_manager{
  uchar curr_file;
  uchar *file[3];
  uchar valid[3];
  uchar file_to_replace;
  uchar first_enter;
  uchar d;
};
```

Where uchar is unsigned char.
The segments array has the start addresses of the files at flash segments B, C, D.
We use curr_file to save the index of currently executing file.
uchar *file[3] is an array of pointers to the current instruction in a specific file, we can use the current instruction to know the number of arguments, and where the next instruction is located.
The valid array is used to specify if the file is valid, i.e was sent by the PC and written into flash.
file_to_replace is the next file to be replaced, in case we send more than 3, we start overriding the old files.

first_enter is used to initialize some variables and to send an acknowledge to the PC that we are about to start executing one of the files, we set it to true whenever we receive a command to execute a new file.
d is the delay time between 2 actions in the script mode (sampling or changing the LCD), we set it in this struct so we that we can reset it to it's default value on first_enter, and change it using the set_delay command.
We note that we did not save the file name as it has no use, and we would also need to lengthen the command for receiving the file which means more communication.
We also didn't save the length of the file, since we used appended an EOF character at the end of the commands, to know when we reached the end of it.

Additionally we implemented more commands for the script mode:

- servo_deg_ldr: similar to servo_deg, but uses the LDR sensor.
- servo_deg_both: similar to servo_deg, but uses both sensors.
- servo_scan_ldr: similar to servo_scan, but uses the LDR sensor (like scan_ldr state).
- servo_scan_both: similar to servo_scan, but uses both sensors (like scan_both state).

5.3.2 Memory Management

We have multiple buffers that exist in ram that get modified from time to time, they include the file receiving buffer of size 61 bytes, adc samples of size 16 bytes, the ring buffer for TX of size 32, and some variables here and there, overall less than 150 bytes.
For constant value arrays such as the LDR calibration data we set them as const arrays so they will be mapped into flash memory along with the code.
So we use about 150 bytes of ram for data, that leaves us with about 350 bytes for stack usage, which is more than enough as we go a maximum of 4-5 function calls deep.

5.3.3 HW Timers and Ultrasonic

Timer A0
This timer used mainly as a trigger for actions like the Ultrasonic sensor, for the ADC10 to start sampling samples and activating the instructions in script Mode (sampling and changing the LCD).

Timer A1
This timer does 2 main tasks which is getting echo time of the ultrasonic sensor by using capture mode and to send PWM signal to control the servo motor by using compare mode.
In capture mode we captured and saved the value in rising edge and falling edge to calculate the echo time.
In compare mode (servo pwm), we set the values of the compare registers based on the calculations described in the project explanation (0.6 ms for 0 degree, 2.5 ms for 180 degrees).
Calculating the degree of the servo:

$$SMCLK\ Frequency/10000\ (0.1ms)$$
$$\frac{2^{20}}{10000} = 104.8576(105)$$
$$0.1ms = 105\ \text{cycles}$$

$2.5ms$ for max degree 180, $0.6ms$ for degree 0, $1.9ms$ between 180 and 0.

$$\frac{(104.8576 * 19)}{180} = 11$$

We get 11 cycles per degree.

5.3.4 ADC10 and LDR

We used Repeat-Sequence-of-Channels mode of the ADC to continuously sample the ports where the LDR is connected, we set it to sample from port a3 downto a0, where a3 is P1.3, a0 is p1.0, which are the pins of the LDR, and pins p1.1 and p1.2 are used for UART communication, but as we can't skip sampling them we just ignored their results, additionally we used the DTC to transfer the sampling results into an array of length 16 integers, i.e we sampled each pin 4 times and calculated the average.

We set the DTC to interrupt after finishing 16 samples and call the ADC ISR in which we calculate the distance and send a message to the PC (explained in more detail at 4.2).
So overall in order to sample LDR, we turn on the ADC, set the DTC to sample 16 samples and copy them to the samples array, then we trigger the sampling using the timer interrupt by setting the ADC10SC bit, which will call the ISR when it's done.