# RTOS graduate project: Process Migration

## Goals

- Establish high-speed CAN communication between two microcontrollers,
- Integrate capability to load ELF files from disk into dynamically allocated memory,
- Develop a capability for OS to mark all the threads for migration and stop scheduling them.
- Enable reliable communication of the process' data structures through CAN bus.
- Add commands to your interpreter to query about the running threads and processes.

## Relevant files (Enhanced Lab5)

• RTOS_Lab5_ProcessLoader (code for the project)
– heap.c and heap.h containing prototypes for a heap memory manager.
– ff.c, ff.h, ffconf.h and integer.h containing a FAT file system implementation
– eFile.c containing an eFile-compliant wrapper around the FAT file system API
– loader.c, loader.h, and elf.h containing an ELF file loader implementation
– loader_config.h configuration of the ELF file loader for integration into OS
– RTOS_Labs_Common/can_project.c with CAN driver code for sending and receiving messages.
– Lab5.c with receiver thread code and main.
• RTOS_Lab5_User project with user application
– OS.h (user-level OS API)
– osasm.s (trampolines to call routines in the OS kernel via SVC exceptions/traps)
– Display.c and Display.h (user-level display driver stub)
– User.c (user application)

## Software Design and Implementation

The lab's goal is to build on top of Lab5 and facilitate OS and CAN drivers to reliably migrate a process from one microcontroller to the other by making use of the HEAP memory that we built in Lab5. In this project, we extend the user process from lab5 loaded from SD-card to blink LEDs continuously.

# The semantics of CAN communication

The CAN_FRAME_SIZE macro, defined in can_project.h, determines the frame size of CAN physical layer communication. To increase the availability of the migrated process, we need migration latency to be minimal. Hence, we need to determine the maximum possible frame size.

We experimented with different CAN frame sizes and discovered that we could not send more than 8 bytes of data through the CAN bus at once. The receiver fails to receive any data if the frame size exceeds 8 bytes. Hence, we chose 8 bytes as our frame size for the project.

The CAN bus is very reliable in the physical layer. Hence, we do not have any CRC frames (or frames) in the message layer.

## CAN packet (message) format:

**FRAME1 - size in bytes (n)**
**FRAME2…FRAME(n/8) - rest of the message**

Here are the associated function prototypes for sending and receiving a message:

```
// dequeue size number of bytes from the FIFO queue and
// put in the ptr pointer.
void CAN0_ReceiveMessage(uint32_t size, uint8_t* ptr);

// convert the next 4 bytes in the FIFO into uint32_t type and
// get rid of extra 4 bytes from the first frame
uint32_t CAN0_ReceiveSize(void);

// start by sending a size frame
// followed by packing the rest of the data into a stream of bytes
void CAN0_SendMessage(uint32_t size, uint8_t* data);
```

## Timer thread for sending messages

To reliably communicate the data, the sender should allow some time between the frames to transmit the current FRAME. To achieve this, we used a TIMER thread that sends data through the CAN bus — can_timer_proc. The sender process uses MAILBOX to communicate the frame data to the TIMER thread. When the timer thread periodically wakes up — 126 microseconds period — if there is valid data in the mailbox, it sends the frame through the CAN bus.

```
// timer proc for sending the data
void can_timerproc(void);
```

### Interrupt handler for receiving messages

Data transmission through the CAN bus triggers CAN interrupt handler. If the frame corresponds to the current node, i.e., the message-id and RX id matches, it puts each byte of a frame into a FIFO queue of size 64. From the FIFO, the receiver thread keeps unpacking the data while it realizes different data structures related to a process, as explained below.

### Configuring CAN ids of the sender at runtime

We have the same code flashed on the two controllers — the same CAN send and receive IDs. To convert one of them into a sender at runtime, we need to swap the send and receive IDs. To achieve this, we have an SW1 interrupt handler. We reset the CAN subsystem and exchange the send and receive ids whenever SW1 is pressed.

## The semantics of Process migration

The microcontrollers start with a receiver thread that continually listens if any process transmission occurs on the CAN bus.

Both the sender and receiver agree on a predefined sequence of data structures transmitted across the CAN bus. The following is the series,

PCB -> text -> data -> (stack pointer -> TCB -> stack) [repeat for all the threads]

The following flowcharts illustrate the sequence better.

### Sender

After SW1 is pressed, one of the MCU transforms into a process sender and launches a new sender thread whose flow chart is as shown below.

The first step is to launch the migrated process. We use Lab5 in fracture to establish a process whose program resides in the SD-card. This program is derived from the Lab5 user program that it toggles LEDs and indicates that the process is running. We have two threads in the migrating process — one main thread and one subordination thread. Both of them toggle different LEDs, PF2 and PF3.

The second step is waiting for the user to signal migration. We use another switch, SW2, for the user's input. We enter the critical section and mark all threads for migration once SW2 is pressed. The scheduler ignores all threads marked for migration.

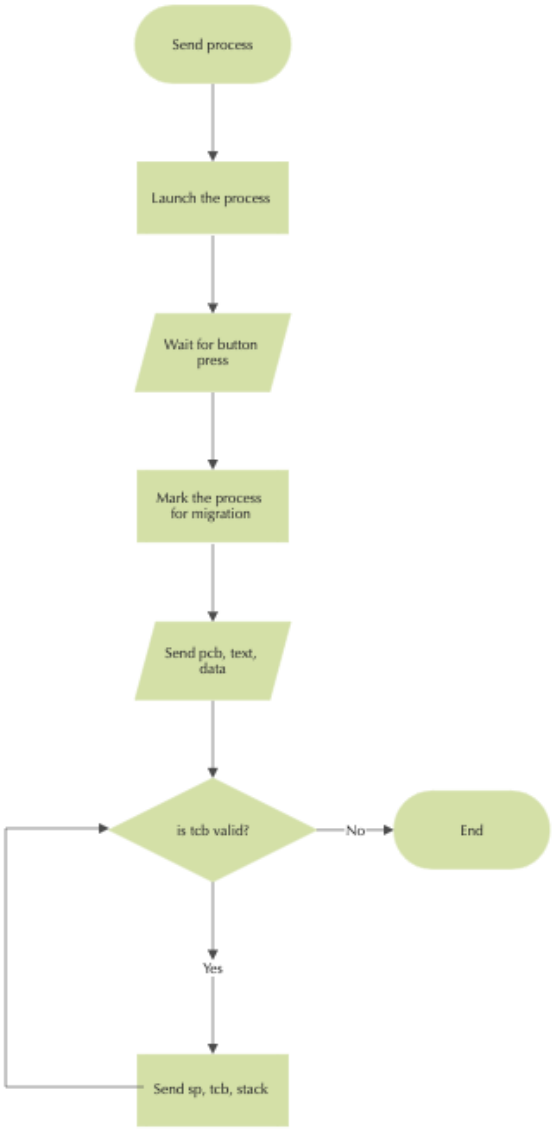
***// mark the threads of the process as being migrated.***
***int OS_mark_for_migration(int pcb_id);***

To implement easy interaction with OS TCB structures, we implemented iterator design pattern methods that walk through the list of threads that belong to a process. We use these methods while sending threads, and to print the interpreter's process statistics as described in the debugging section.

```
// get the process by its name
pcbType* OS_get_pcb_by_fl_name(char fl_name[OS_PCB_FL_NAME_LEN]);

// initilize the pcb_walker to the first process in the list
void OS_pcb_init_walker(pcbType** pcb_walker);

// go to the next valid pcb and set the passed pointer to it
void OS_pcb_walk(pcbType** pcb_walker);

// return the pcbid of the process
int OS_get_pcb_id(pcbType* pcb);

void OS_tcb_init_walker(tcbType** tcb_walker_ptr, int pcb_id);

void OS_tcb_walk(tcbType** tcb_walker_ptr, int pcb_id);
```

The sender simply uses the CAN message semantics in the previous section to send PCB, text, data, and each thread data structure in the sequence.

flow chart




## Receiver

As mentioned above, at the start of the OS the receiver thread is launched. It starts by allocating a free PCB and waits for the FIFO to get filled with the required data. Once the sender starts sending data through the CAN, it fills the local PCB with the migrated data.

The second step is to receive TEXT data of the program. After receiving the size of the text — the frame, the receiver thread allocates the size amount of the heap and keeps waiting on CAN FIFO to fill the data.

The third step is similar to the second and fills the DATA segment.

Based on the number of threads information in the PCB, we loop for each thread and extract stack pointer, tcb and stack information. For each thread, we set the relevant fields of the newly allocated TCB based on the received tcb:

```
// copy the relevant information to the tcb_ptr
tcbType* tcb_ptr = GetFreeThread();
tcb_ptr->pcb = pcb_ptr;
tcb_ptr->priority = tcb.priority;
tcb_ptr->sleep = tcb.sleep; // this may not be accurate
tcb_ptr->blockedTime = tcb.blockedTime;
tcb_ptr->sp = Stacks[tcb_ptr->id] + rel_sp;

*(Stacks[tcb_ptr->id] + rel_sp - 2) = (int32_t)(&RunPt); //Set the RunPt

tcb_ptr->is_migrating = tcb.is_migrating; // we don't want the thread to run yet
```

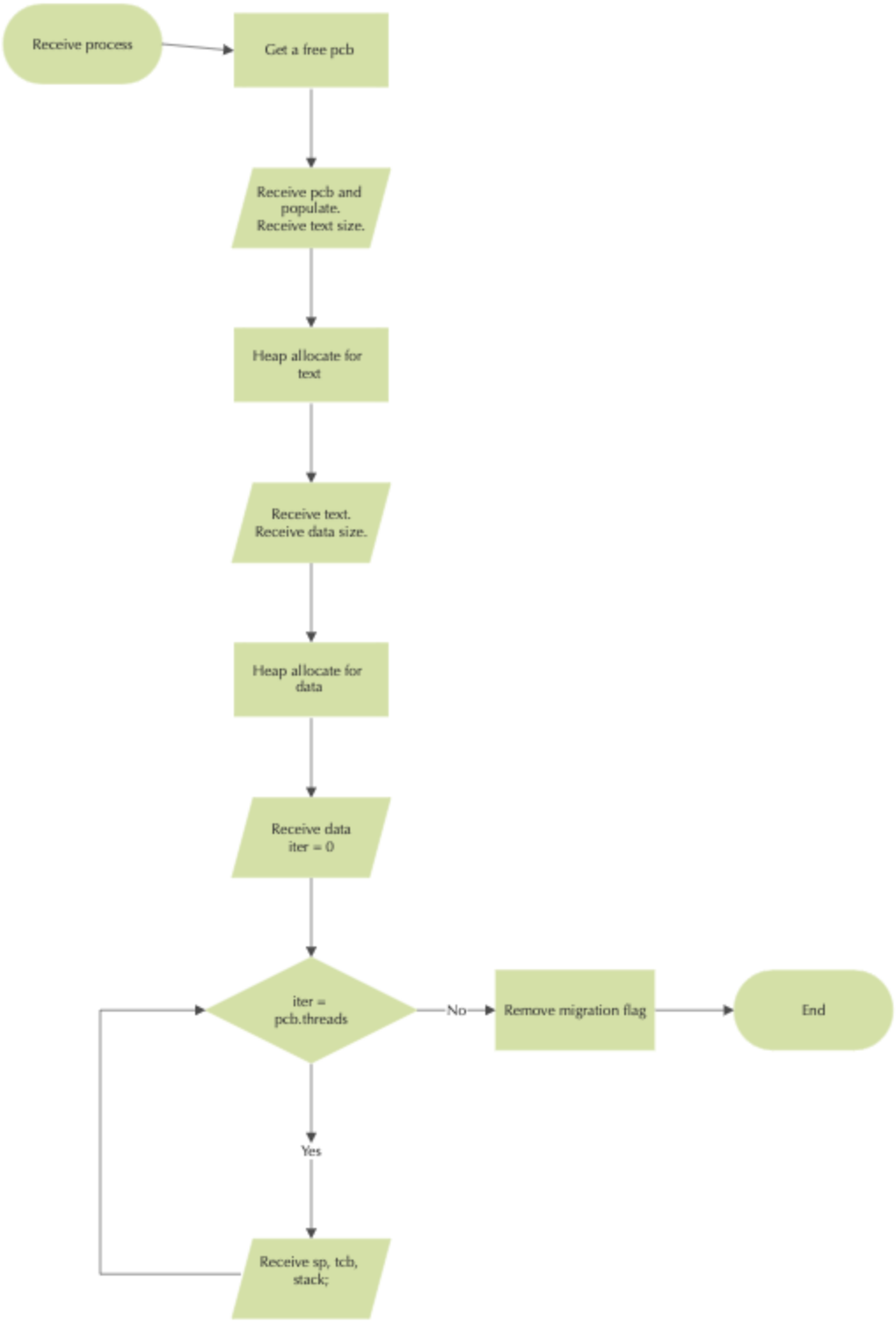
Our Context_Switch assembly implementation needs the stack_pointer -2 to be pointing to the current running thread. Hence we use the relative stack pointer to correct the stacks for each thread, as shown above.

As you can note, the receiver follows the same predefined sequence as the sender to communicate the process data structures that are described above.

Once all the process-related data structures are received, we enter the critical section and remove the migrating flag of each thread that belongs to the process. Then the scheduler will start running the process.

```
// unmark the migrated thread
void OS_add_migrated_thread(tcbType* newThreadPt);
```

flowchart


Receive process
Get a free pcb
Receive pcb and populate.
Receive text size.
Heap allocate for text
Receive text.
Receive data size.
Heap allocate for data
Receive data
iter = 0
iter = pcb.threads
No
Remove migration flag
End
Yes
Receive sp, tcb, stack;

# Hardware and Design

## SD-card and User program compilation

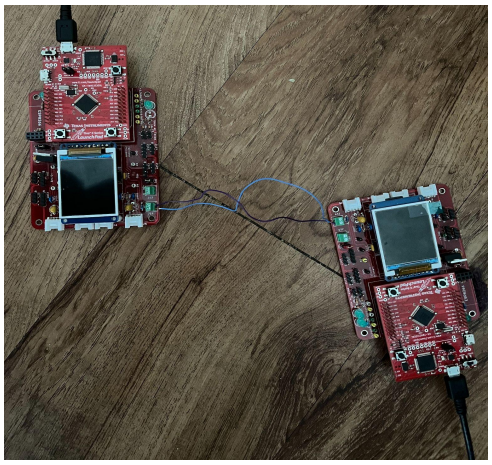
For the sender thread to launch the process, we need the microcontroller to have the SD-card mounted with the USER.axf program. We modified the user process provided in the Lab5 starter code. Hence we needed Keil's license to compile the code. After receiving the license, follow the below steps to compile the code:

1. Step 1- run Keil in administrator mode;
2. Step 2- execute file->license management;
3. Step 3- click get LIC via the internet;
4. Step 4- make an account on ARM/Keil (the email must be valid);
5. Step 5- enter 15 character code;
6. Step 6- they send you an email with the license (30 characters long) and enter 30 character code into New License ID Code (LIC) window.

Once the program is ready, copy the USER.axf file from the Lab5 user project to the SD card and insert it into the microcontroller.

## Connecting CAN port of two Microcontrollers

Lab6 explains the details of CAN bus protocol. The CAN ports of the two microcontrollers need to be connected to set up the communication, as shown below.



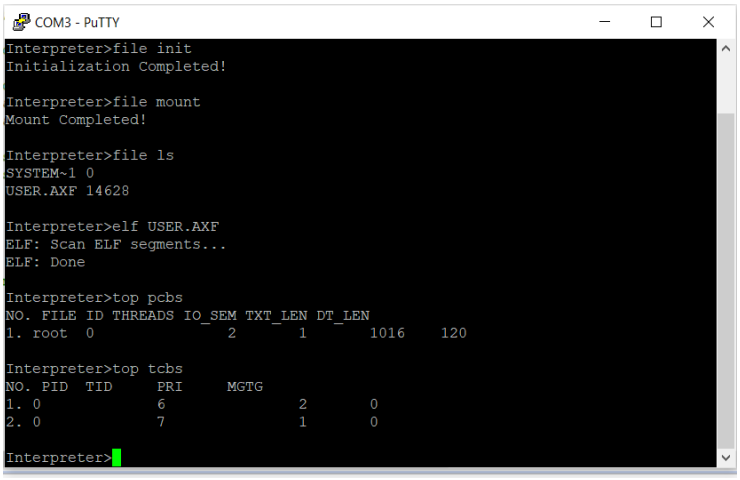
# Debugging

## Top command

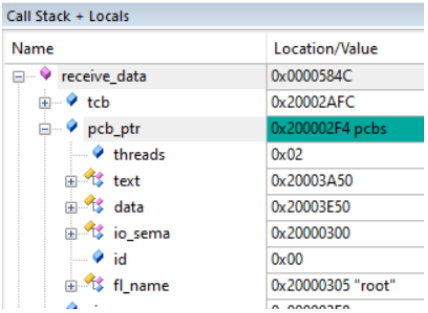
We have added TOP-like command in the interpreter to print the active process and threads interactively.

***# print the list of the active process***
***% top pcbs***

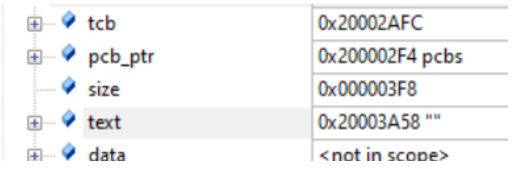
***# print the list of threads***
***% top tcbs***

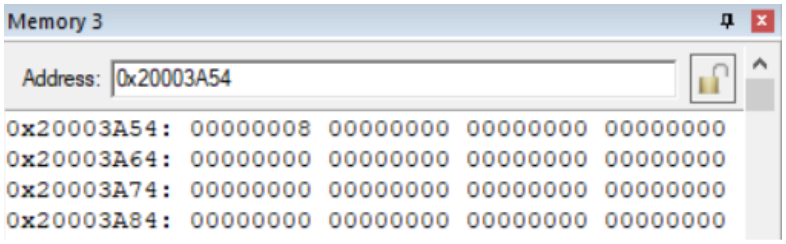



# PCB populated

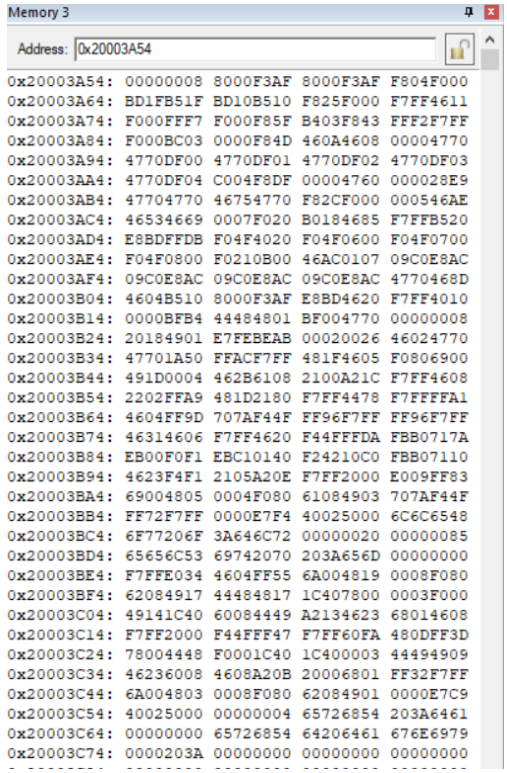



# Text initialised on the Heap

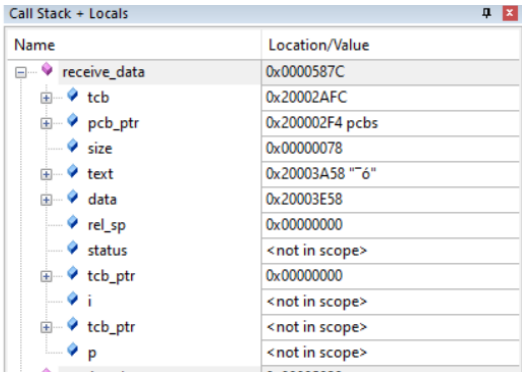


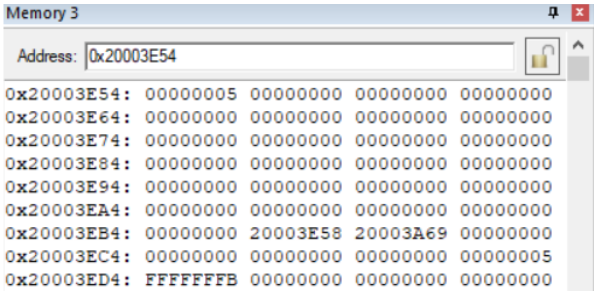



## Text populated



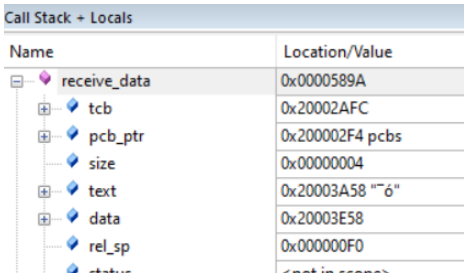

## Data populated

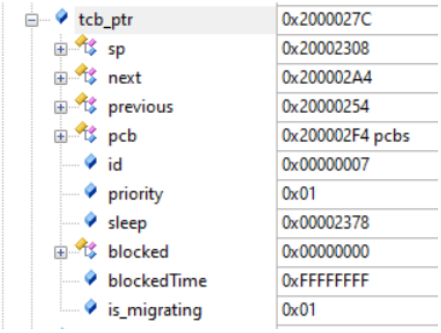





## TCB

Relative stack pointer



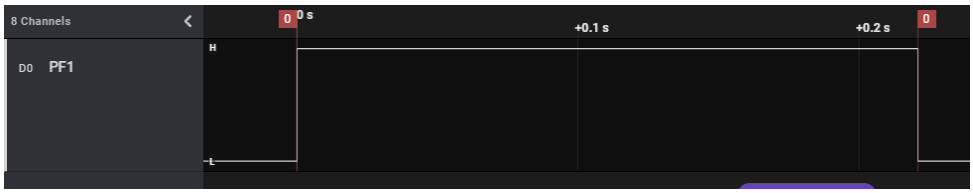

# Analysis and Discussion

## Limitations

1. Only one thread migration is supported at any time; since there is only one FIFO queue, the receiver can't filter the correct data.
2. Currently, the sender and reciver implementation support one thread. However, we can easily extend this for multiple processes.
3. What factors limit the number of threads that can be migrated, and how could it be increased?

a. It depends on the number of processes that the OS can support at once. Currently, our OS supports only 3 three simultaneous processes.
b. Our OS only supports 10 threads as of now. Hence, the number of threads is also a limitation.

Time taken to send the process: 220 ms




# Youtube link

# Github link