# Emb4 Log

## Task execution

Queue:

Argument struct per function

Wrapper function per task

Preallocated arg array or struct without malloc can only hold one parameter at a time, which sucks.

Use malloc structs for regular tasks, preallocated struct for interrupts

In func:

Malloc for struct

Fill struct with args and arg pointers

HANG ON

If args are only passed down through pointers and up through editing those pointers, and args are COPIED into arg array/struct

Ex.: regular task wants to schedule next task. Creates new task, fills args with desired values and poss. pointers that were passed down to it like a graphics buffer ptr, calls addToQueue which copies these values into queue, and function can safely exit. No malloc required?

BUT void\* to function arg struct needs to point to something. That’s why you need malloc.

You need this for array too: void\*\* needs to point to variables which you can then cast

In order to free these args, you need to know what you’re dealing with. Destructor function per task?

~~But this is only a big issue if you want to call a function multiple times in one go. When would you want to do this?~~

~~Graphics: no, can pipeline element drawing~~

~~Communication: no, needs to happen in right order so pipelined~~

~~Time: no, one increment at a time~~

~~Buttons: no, function per button~~

So pipeline:

Function adds task, calls malloc for relevant struct, fills struct

Addtask adds to queue

Handletask calls function, then if successful: calls destructor for args

ONLY ARGS (and handler/destr funcs) ARE POINTERS, REST IS COPIED

#### RTOS RULES:

* Variables must be passed in struct to (wrapper) function
* Use preallocated structs for interrupts.
* When using malloc for argument struct in adding new task to queue from a non-interrupt function or task, also declare, write and pass to task struct function pointer to destructor that frees allocated memory.
* When not using malloc, set destructor pointer to to NULL.

RTOS flowchart tekenen:

A diagram of a machine

Description automatically generated

https://www.iar.com/zh/knowledge/learn/programming/using-rtos-diagrams

Flowchart maken van taken!!!

Interrupt -> A -> (als succes) B of (als fail) C -> etc

### Methods

Tasks: Superloop vs FIFO vs LIFO

Superloop is slow, cumbersome, not versatile. Executing all tasks everytime is going to cause blocking and clog.

FIFO: 100% zekerheid dat tasks afgehandeld worden, maar geen prioriteit

LIFO: snelle task afhandeling, maar als dit niet snel genoeg gebeurt en/of er steeds tasks bijkomen kom je nooit bij de oudste taak

Beslissing: FIFO.

|  |  |  |  |
| --- | --- | --- | --- |
| System | Flag based | Queue based | Grouped flags |
| Pros | Possibility for priority in flag handling order | Fast handling: handle, move head, move on  Versatile, variable length | Faster than pure flags:  No need to check every flag every time  Priority |
| Cons | Iterates through all flags every time, wasteful and slow  Only has priority if ex one task per loop, then loops back. Then possibility for stale tasks | No priority without sorting or multiple queues | Only has priority if ex one task per loop, then loops back. Then possibility for stale tasks |

Grouped flags: uses status bits to group flags. Status bit is set if any flag bits are active. Then handler only needs to check status bit to see if flag has changed. Could be possible to just group flags by putting in separate variables.

All options will require some interrupts.

Choice: queue with interrupts.

### Coding rules: Queue

* No returning variables apart from error codes or function pointers for next function to put in queue.
* Variables to write have to be provided as pointers in parameters. This also means pointing to arrays safely by providing their sizes.
* Break up tasks. Next step in overall task can then be added to end of queue (eg. Draw line as part of draw square, which is then a part of updating display buffer). Repetitive tasks should be bundled to save queue modification operations.
* Ensure function can be called repeatedly and out of order without adverse effects. Eg. Retrying a failed task multiple times
* Error handling: failed tasks should be readded to end of queue. Poss. Log failures.
* Timeouts: where there is a risk, set timeout values in case task takes too long and ends up blocking.
* Atomicity: whole operation succeeds or task fails completely and needs to be retried.
* Statelessness (wherever possible)
* Avoid blocking operations. Use flags or later retries for asynchronous tasks such as getting back I2C data. Send and check later.
* Dependencies: avoid task *time* dependency clusters. Each task should be able to complete its task without waiting or blocking.
* Graceful shutdowns on termination.
* Use interrupts only to add tasks and perform the most important time dependent tasks.

#### Queue structure

Requirements reasoning:

1. Easy insert, delete -> Linked list
   1. Next item
2. Want to be able to add to end of queue fast
   1. Last item, end of list pointer
3. Handle functions:
   1. Handler function pointer
   2. Variable number of arguments
      1. Arguments array with pointers to argument variables
4. Want to know queue length for limiting
   1. All queue items point to single struct with data about queue, length decr/incr by insert/handler functions
5. queue length can cause running out of memory, max length is not easy to determine
   1. Use an array to pre-allocate limited/static amount of memory
6. Want debug method to know which functions are in queue
7. Splitting up big functions: need way to know which function goes next since queue handler doesn’t
   1. Func process pointer with index? Means have to store process order in array or LL like 2nd queue. Not ideal
   2. Throwing everything in at once is going to cause blockage and wrong order if something fails
   3. State machine for processes
   4. Do I really need this? Could keep functions big and not have this issue, apart from poss. USB communications. Could do those with DMA
8. No sorting because it’s slow
   1. No priority

Queue item structure:

[Debug] function/task name

Function pointer: function to invoke with parameters filled

Next: pointer to next item in queue

End: pointer to end of queue (to save iterating next to get to end of queue to add a new item)

*God OOP would make this so much better*

#### Queue functions

##### Enqueue:

* Add an item to back of queue:

Get item at back of queue from head End pointer

Create new item

Update next pointer at End

Update End pointer at Head

##### Dequeue:

* Remove item from front of queue:

Operations:   
Get Next pointer from Head

Update End pointer at Next

Delete old Head

Update Head pointer to Next

## Display

### GFX

Pipeline:

Set background colour

Draw elements over each other

Differentiate against old buffer to determine pixels to be updated

Send updated pixels

Display has (partial, boxed) scroll mode on p. 137

### IO Notes

RST on low, pull down on startup

CS on low (pull up res)

DC: data on high, command on low

The SPI communication protocol of the data transmission uses control bits: clock phase (CPHA) and clock polarity (CPOL):

The value of CPOL determines the level when the serial synchronous clock is in idle state.

CPOL=0, that its idle level is 0.

The value of CPHA determines the timing of the data bits relative to the clock pulses.

CPHA=0, data is sampled at the first clock pulse edge.

The combination of these two parameters provides 4 modes of SPI data transmission. The commonly used is SPI0 mode, it is that GPOL=0 and CPHA=0.

From the figure above, SCLK begins to transfer data at the first falling edge. 8 bits data are transferred at one clock period. Use SPI0 mode, High bits transfer first, and LOW bits following.

Slow update rate, screen tearing:

p. 84: MPY write slower than panel read, causes panel to update from buffer mid write which causes screen tearing.

### Startup seq.

RST pull down

120ms low

Rst high

### Registers

Page 104: Systems commands

Startup seq:

LCD\_Reset();

//\*\*\*\*\*\*\*\*\*\*\*\*\* Start Initial Sequence \*\*\*\*\*\*\*\*\*\*//

LCD\_WriteReg(0xB1); // FRMCTR1: Normal mode, full colours p155

LCD\_WriteData\_Byte(0x01); // RTNA set 1-line period: 0001

LCD\_WriteData\_Byte(0x2C); // FPA: Front porch

LCD\_WriteData\_Byte(0x2D); // BPA: Back porch

LCD\_WriteReg(0xB2); // FRMCTR2: In Idle Mode (8-colors) p155

LCD\_WriteData\_Byte(0x01); // RTNB: set 1 line period 0001

LCD\_WriteData\_Byte(0x2C); // FPB: Front porch

LCD\_WriteData\_Byte(0x2D); // BPB: Back porch

LCD\_WriteReg(0xB3); // FRMCTR3: In partial mode + full colours p155 (Partial update??)

LCD\_WriteData\_Byte(0x01); // RTNC

LCD\_WriteData\_Byte(0x2C); // FPC

LCD\_WriteData\_Byte(0x2D); // BPC

LCD\_WriteData\_Byte(0x01); // RTND

LCD\_WriteData\_Byte(0x2C); // FPD

LCD\_WriteData\_Byte(0x2D); // BPD

LCD\_WriteReg(0xB4); // INVCTR: Display inverion control: Column inversion p155 p162

LCD\_WriteData\_Byte(0x07); // Set inversion: currently 111

//ST7735R Power Sequence

LCD\_WriteReg(0xC0); // PWCTR1 Power control setting p156

LCD\_WriteData\_Byte(0xA2); // AVDD 2:0, VRHP 4:0

LCD\_WriteData\_Byte(0x02); // 000 + VRHN 4:0 set GVDD voltage

LCD\_WriteData\_Byte(0x84); // MODE 1:0 + 000100

LCD\_WriteReg(0xC1); // PWCTR2 Power control setting p156

LCD\_WriteData\_Byte(0xC5); // VGH2 1:0 + -- + VGLS EL 1:0 + VGHB T 1:0

LCD\_WriteReg(0xC2); // PWCTR3: Power control in normal mode(full colours) p156

LCD\_WriteData\_Byte(0x0A); // DCA (dc booster) 9:8 + SAPA 2:0 + APA (opamp adj) 2:0

LCD\_WriteData\_Byte(0x00); // DCA (dc booster) 7:0

LCD\_WriteReg(0xC3); // PWCTR4: Power control idle mode(8colours) p156

LCD\_WriteData\_Byte(0x8A); // DCB (dc booster) 9:8 + SAPB 2:0 + APB (opamp adjust) 2:0

LCD\_WriteData\_Byte(0x2A); // DCB (dc booster) 7:0

LCD\_WriteReg(0xC4); // PWCTR5: Power control partial mode + fullcolours

LCD\_WriteData\_Byte(0x8A); // DCC (dc booster) 9:8 + SAPC 2:0 + APC (opamp adjust) 2:0

LCD\_WriteData\_Byte(0xEE); // DCC (dc booster) 7:0

LCD\_WriteReg(0xC5); // VMCTR1: VCOM control 1

LCD\_WriteData\_Byte(0x0E); // -- + VCOMS (voltage control) 5:0

//ST7735R Gamma Sequence

LCD\_WriteReg(0xe0); // GAMCTRP1 Gamma adjustment + polarity

LCD\_WriteData\_Byte(0x0f);

LCD\_WriteData\_Byte(0x1a);

LCD\_WriteData\_Byte(0x0f);

LCD\_WriteData\_Byte(0x18);

LCD\_WriteData\_Byte(0x2f);

LCD\_WriteData\_Byte(0x28);

LCD\_WriteData\_Byte(0x20);

LCD\_WriteData\_Byte(0x22);

LCD\_WriteData\_Byte(0x1f);

LCD\_WriteData\_Byte(0x1b);

LCD\_WriteData\_Byte(0x23);

LCD\_WriteData\_Byte(0x37);

LCD\_WriteData\_Byte(0x00);

LCD\_WriteData\_Byte(0x07);

LCD\_WriteData\_Byte(0x02);

LCD\_WriteData\_Byte(0x10);

LCD\_WriteReg(0xe1); // GAMCTRN1 Gamma adjustment - polarity

LCD\_WriteData\_Byte(0x0f);

LCD\_WriteData\_Byte(0x1b);

LCD\_WriteData\_Byte(0x0f);

LCD\_WriteData\_Byte(0x17);

LCD\_WriteData\_Byte(0x33);

LCD\_WriteData\_Byte(0x2c);

LCD\_WriteData\_Byte(0x29);

LCD\_WriteData\_Byte(0x2e);

LCD\_WriteData\_Byte(0x30);

LCD\_WriteData\_Byte(0x30);

LCD\_WriteData\_Byte(0x39);

LCD\_WriteData\_Byte(0x3f);

LCD\_WriteData\_Byte(0x00);

LCD\_WriteData\_Byte(0x07);

LCD\_WriteData\_Byte(0x03);

LCD\_WriteData\_Byte(0x10);

// TODO WTF are these commands??

LCD\_WriteReg(0xF0); //Enable test command ???

LCD\_WriteData\_Byte(0x01);

LCD\_WriteReg(0xF6); //Disable ram power save mode ???

LCD\_WriteData\_Byte(0x00);

LCD\_WriteReg(0x3A); // Pixel colour mode

LCD\_WriteData\_Byte(0x05); //65k mode (RGB 565)

LCD\_WriteReg(0x36); // Memory data access control

LCD\_WriteData\_Byte(0x60); MY + MX + MV + ML + RGB + MH + 0b10

DEV\_Delay\_ms(200);

LCD\_WriteReg(0X11); // SLPOUT: Sleep out (off?) and booster on

DEV\_Delay\_ms(200);

LCD\_WriteReg(0X29); // Command: display on (display off is 28h)

DEV\_Delay\_ms(200);

### Notes

12-bit, 16-bit and 18-bit per pixel input formats: RGB444, RGB565 and RGB666 color formats.

SPI:

CS should remain low through entire transmission (but doesn’t have to for reg addr + data)

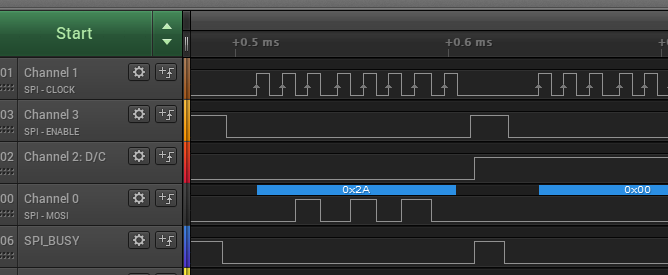
Transferring large amounts of data (for ex. display buffer):

Array? Or setting cursor to next pixel every time??

SPI transferring: SPI module takes a while to go off IDLE. Currently to solve this I’m blocking until it starts transmitting but this isn’t ideal. Options to cut out CS bump below:

* TODO Use SPI\_Module\_STS\_BYTE\_COMPLETE flag to signal end of first byte, meaning register has been set and DC should be changed to DATA from COMMAND

Q: Do I want to do SPI transfers in one go or split them up for better qeue speed?



SPI module flag setting is on p.25 of psoc datasheet

Display update rate:

Check if updates are happening often enough, if not: set interrupt to update display immediately with ISR\_SetPending()

A diagram of a diagram

Description automatically generated

Transmitting large amounts of data:

Sort data properly into word HB, then LB

Fill spi buffer (you know this amount)

Then loop until all data put into buffer

{

Wait for tx fifo buffer to not be full

Load next byte of word

}

Use Union to easily separate HB and LB without fuss?

Or use transmit queue to allow for transmissions to be interrupted by other functions, but this makes for much slower display update

Data to transfer for full display RAM update: 128 \* 160 \* 2 bytes = 40.96 MB of data.

This is a lot. Transfer speed is pretty much maximised with above method.

Next option: differential buffering:

Check which pixels to update and only update those.

How cursor / window setting works: (ST7735S datasheet p74)

First transmit Xstart, Xend, Ystart, Yend coords for window to write in RAM

Then send command to start writing in RAM and start sending display data ( this can continue indefinitely)

Maximum slave clock freq: spi module manual p34

## RTC

### Links

https://www.dfrobot.com/product-2304.html

### I/O

RST active low

### Timings

Doesn’t do milliseconds

But can output 32.678 kHz wave on 32k pin or poss. 1024 Hz or 4096 Hz (or higher freq) on Interrupt pin, but this would disable alarm interrupt.

Timing requirement is 0.0025 ms. Would require min.

Counter\_WriteCounter() to write new value, disable counter before writing then re enable

Counter\_ReadCounter() returns counter (capture) value, doesn’t trigger capture event or reset counter

Counter\_WriteCompare() for alarm

VERSLAG INFO

Timekeeping: using a struct vs using UNIX timestamps:

|  |  |  |  |
| --- | --- | --- | --- |
|  | UNIX (one 32-bit reg) | Time, date structs | Sec ctr + structs blend |
| Pros | Fast incrementation  Zero delay for update s->mins etc  Fast comparison | No conversion to display | No conversion to display  Saves on updating seconds |
| Cons | Intensive conversion to displayable time  Needs set start time | Intensive incrementation (days/month especially)  Slower | Intensive incrementation (days/month especially) |

Conclusion: The real question is when you’d rather have intensive logic: when you set/get time or when you increment/compare?

Since gets are happening 5x per second, UNIX’s flaws are amplified a lot. Whereas incrementation on structs happens 1x per second and on blend option only happens 1x per 60s.

#### Error syncing

VERSLAG INFO

Ontdekt dat psoc niet zo accuraat is als RTC (0.25% vs 2ppm). Zorgt ervoor dat deze steeds verder uit elkaar gaan lopen. (integrerende fout)

+ Bonus eis: aan begin verslag aangeven: als reset bij begin assessment blijft semi accuraat op korte termijn, maar door klokvariatie op lange termijn zeer inaccuraat en variabel. Hierom gekozen om ms counter periode steeds bij te stellen op basis van secondeduur van RTC om toch accuraat te blijven tellen.

1s voorbij -> capture ms reg -> interrupt voegt calibratiefunctie toe aan queue -> calibratiefunctie leest capture uit, kijkt naar verschil met “vaste” 1s uit rtc en stelt bij. (idealiter is capt op dat moment precies 0 of TOP. Zo niet:

if(ms\_cur > (millis\_period/2)){

// If timer is running too fast (down counter) it needs a larger period

error = millis\_period - ms\_cur;

}

else {

// If timer is running too slow (down counter) it needs a lower period

error = -ms\_cur;

}

Deze synchronisatie regeling kampt met sterke (error) oscillatie:

Dit komt door overregeling

Of een samplepulse (1s uit rtc) die door een oplopende faseverschil tussen de 1Hz signalen uit rtc en te syncen signaal steeds op een moment met grotere error gecaptured wordt waardoor de timer steeds harder probeert bij te sturen.

DIT METEN MET 2 PROBES EN OSCILLOSCOOP

Dit is op te lossen door:

* Te resetten en te wachten tot volgende 1Hz rtc puls waardoor deze weer synchroon zouden lopen (GEEN OPTIE)
* 32kHz signaal te gebruiken in one shot timer die gestart wordt door tc van ms timer (would work, maar hiervoor moet deze timer werkend zien te krijgen met 32kHz)

Dit is NIET op te lossen door:

* Error begrenzing: vertraagt dit proces enkel

### Alarms

Natuurlijk niet nodig om meer dan 2 alarms per dag te hebben naast patroon

UNIX timestamp?

### Notes

p. 18

DS3232 has slave write and read modes

7 bit address (1101000) then 1 bit for R/W (1/0)

Acknowledge bit sent back after address write

Write: Slave addr | Writebit -> ack bit -> word address , sets pointer-> data

Read: Slave addr | Writebit -> ack bit -> data from where pointer was pointing

The DS3232 then begins to transmit data

starting with the register address pointed to by the

register pointer. If the register pointer is not written to

before the initiation of a read mode, the first address

that is read is the last one stored in the register pointer

p11 has address map

RTC Alarm 2 doesn’t do seconds

## PC interfact:

USB communication:

<https://www.youtube.com/watch?v=nJs6Q4ekVjw>

GUI is goed te doen via python of c, lib vinden

Sources:

<https://community.arm.com/arm-community-blogs/b/embedded-blog/posts/beyond-the-rtos-a-better-way-to-design-real-time-embedded-software>