LegOS

Shannon Zelinski
Chris Cortopassi

Introduction

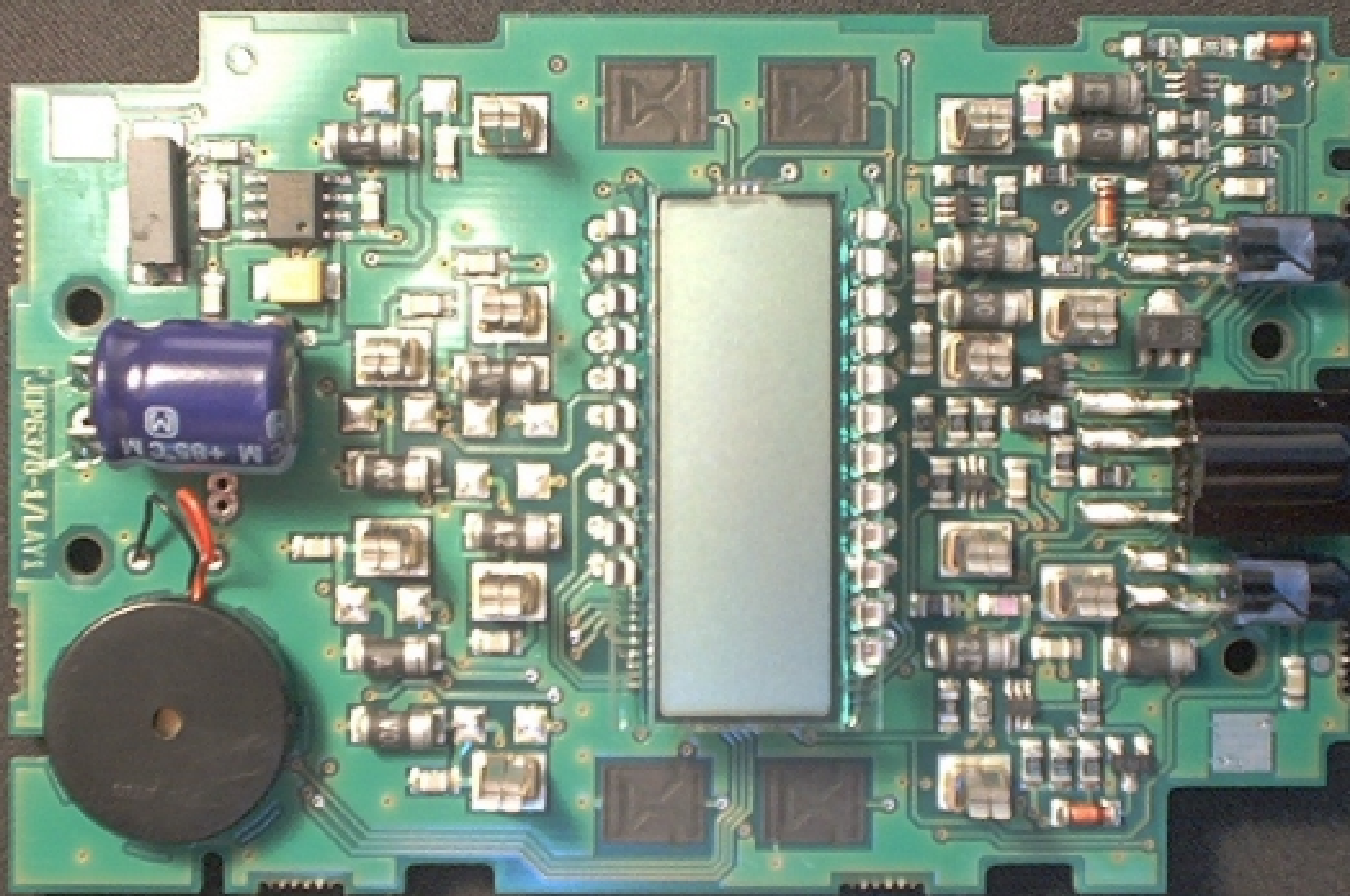
- LegOS + RCX = Real Embedded System
- Not far from real Motion Controllers
  - Scheduling
  - Network
  - Application Developer API
  - Motion
  - I/O
  - User Interface
- Markus Noga's LegOS replaces LEGO byte-code interpreter

Outline (bottom up)

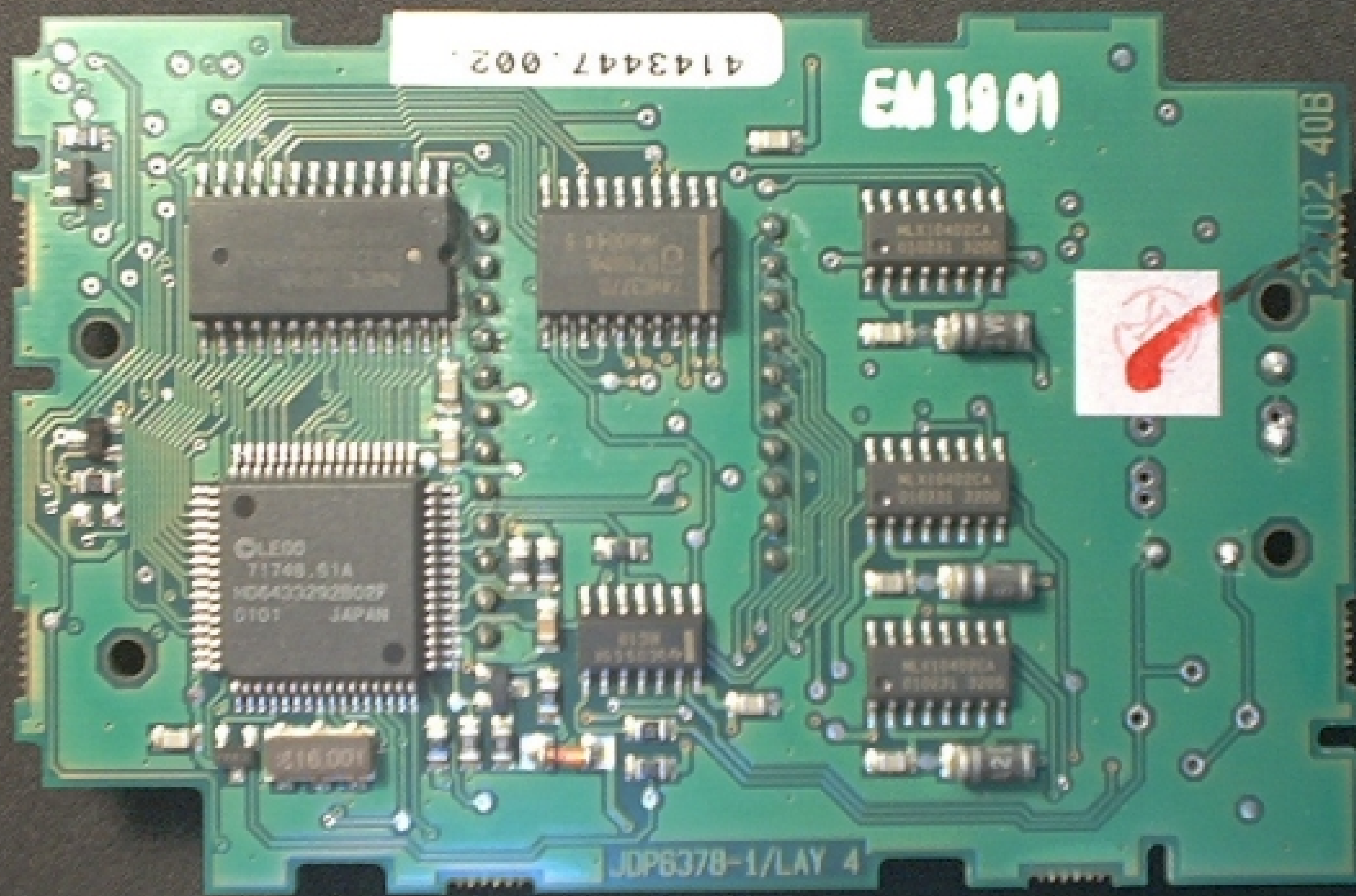
- Hardware
- Assembly Language
- Motor and Sensor Handling
- Task Management: Threading
- Network

“Introduction to the LegOS Kernel” by Stig Nielsson

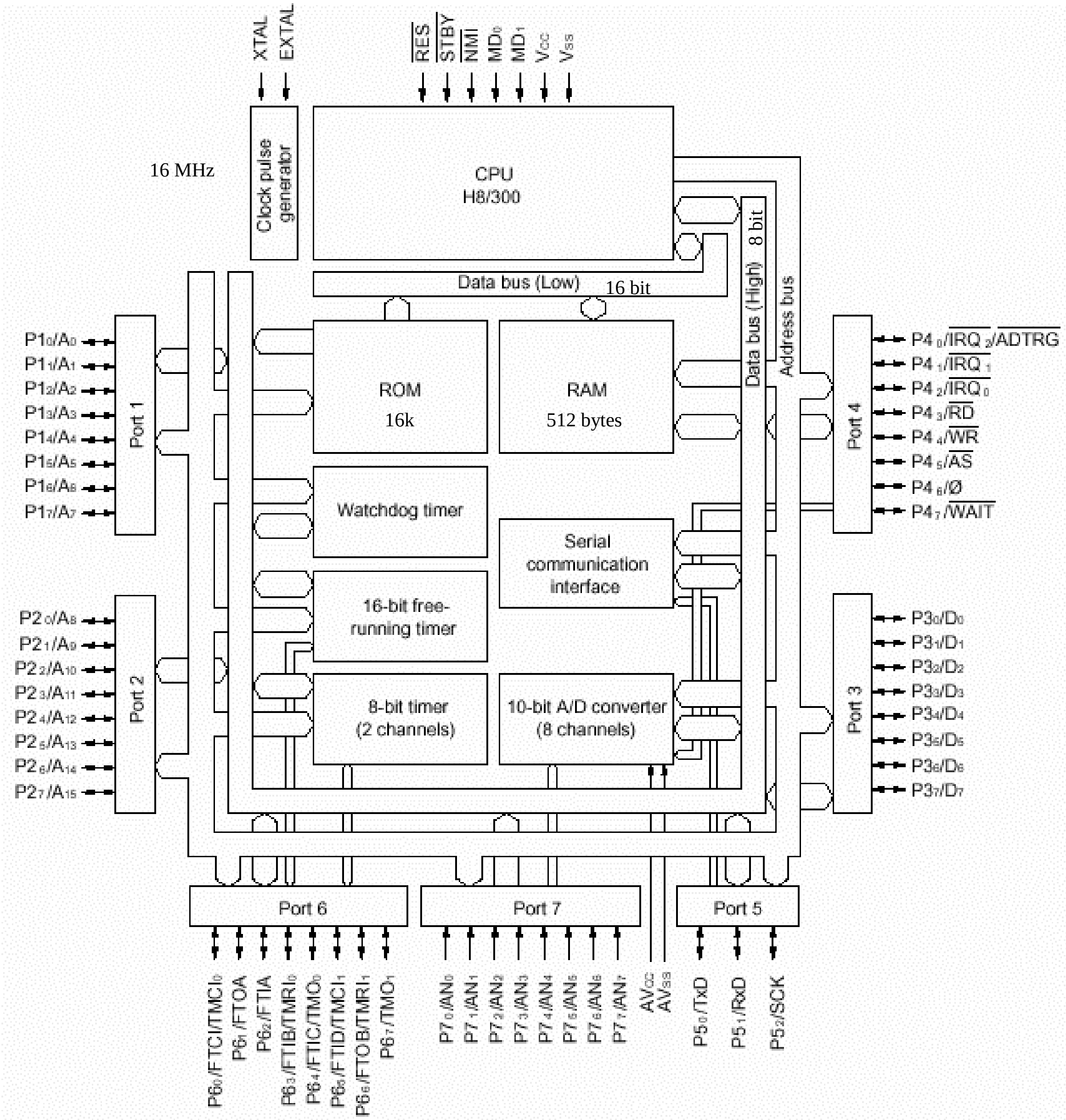
RCX (top)



RCX (bottom)



Hitachi H8/3292 Microcontroller



Address

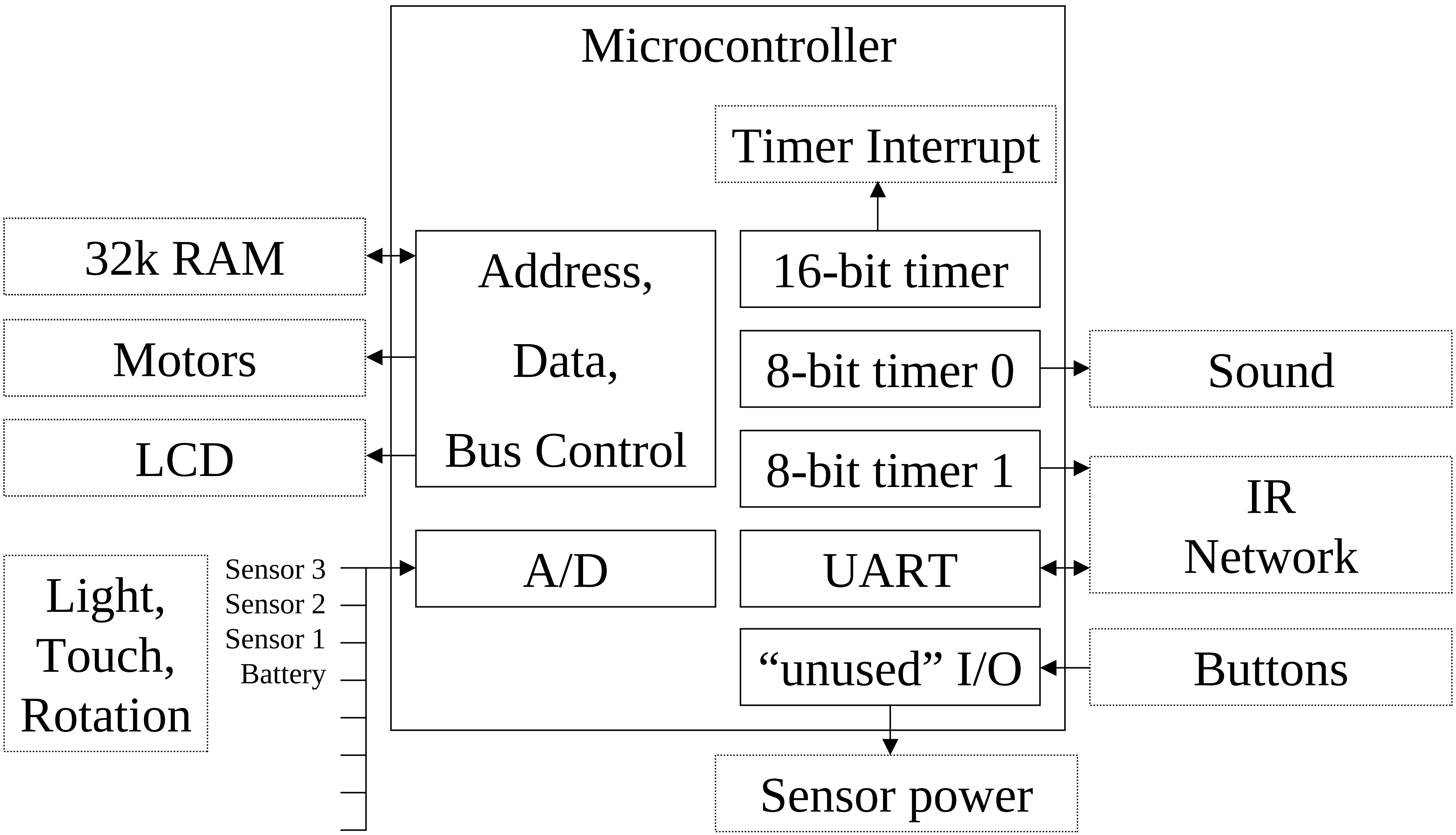
Bus Control

Data

Timers A/D UART

All ports can be configured as general I/O

RCX Connectivity



Hardware Interrupts

Interrupt source		No.	Vector Table Address	Priority
NMI		3	H'0006 to H'0007	High
IRQ0		4	H'0008 to H'0009	
IRQ1		5	H'000A to H'000B	
IRQ2		6	H'000C to H'000D	
Reserved		7	H'000E to H'000F	
		8	H'0010 to H'0011	
		9	H'0012 to H'0013	
		10	H'0014 to H'0015	
		11	H'0016 to H'0017	
16-bit free-running timer	ICIA (Input capture A)	12	H'0018 to H'0019	
	ICIB (Input capture B)	13	H'001A to H'001B	
	ICIC (Input capture C)	14	H'001C to H'001D	
	ICID (Input capture D)	15	H'001E to H'001F	
	OCIA (Output compare A)	16	H'0020 to H'0021	
	OCIB (Output compare B)	17	H'0022 to H'0023	
	FOVI (Overflow)	18	H'0024 to H'0025	
8-bit timer 0	CMI0A (Compare-match A)	19	H'0026 to H'0027	↑
	CMI0B (Compare-match B)	20	H'0028 to H'0029	
	OVI0 (Overflow)	21	H'002A to H'002B	
8-bit timer 1	CMI1A (Compare-match A)	22	H'002C to H'002D	
	CMI1B (Compare-match B)	23	H'002E to H'002F	
	OVI1 (Overflow)	24	H'0030 to H'0031	
Reserved		25	H'0032 to H'0033	
		26	H'0034 to H'0035	
Serial communication interface	ERI (Receive error)	27	H'0036 to H'0037	
	RXI (Receive end)	28	H'0038 to H'0039	
	TXI (TDR empty)	29	H'003A to H'003B	
	TEI (TSR empty)	30	H'003C to H'003D	
Reserved		31	H'003E to H'003F	
		32	H'0040 to H'0041	
		33	H'0042 to H'0043	
		34	H'0044 to H'0045	
A/D converter	ADI (Conversion end)	35	H'0046 to H'0047	
Watchdog timer	WOVF (WDT overflow)	36	H'0048 to H'0049	Low

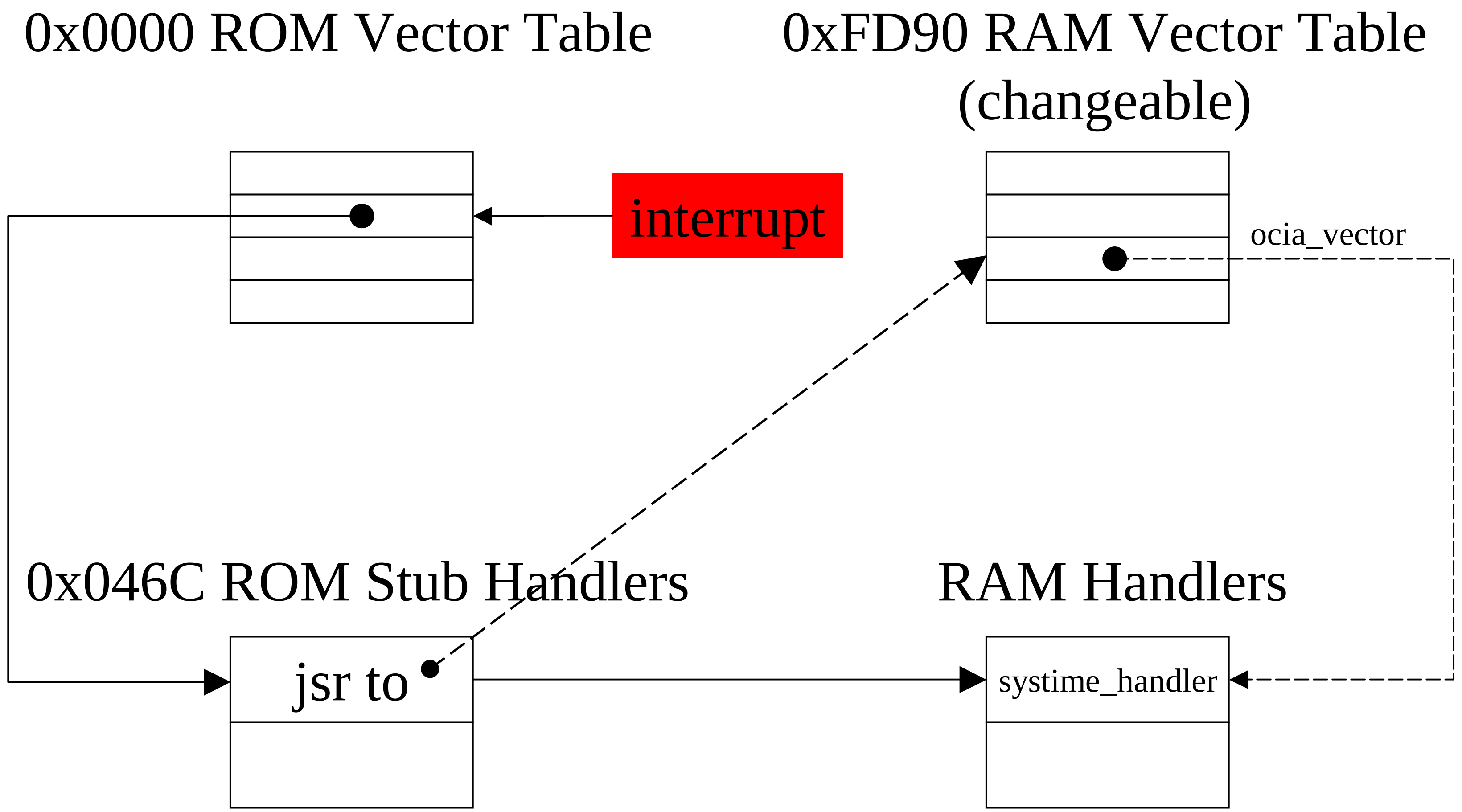
Timer Interrupt

Network

Sensors

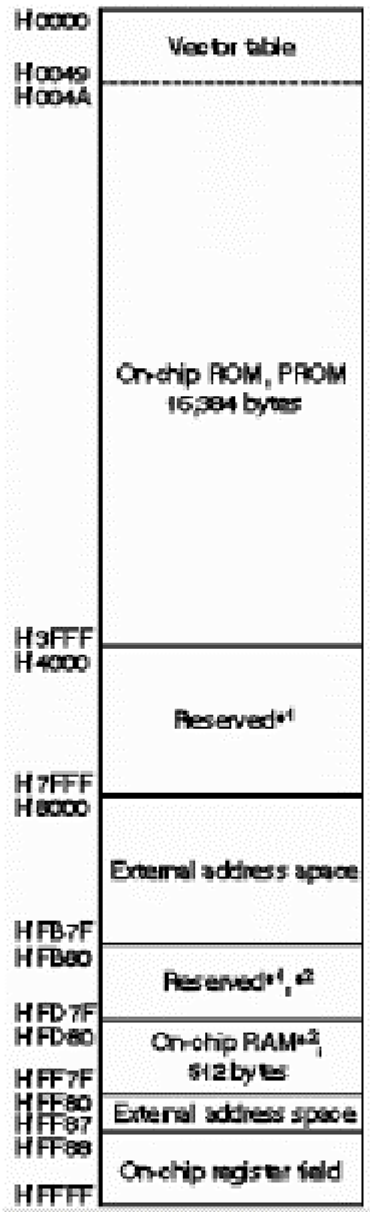
□ LegOS uses

Two Interrupt Vector Tables



Not efficient, but without this indirection LegOS wouldn't exist

Memory



Provided by LEGO:

- Sound Handlers
- LCD Handlers
- Mostly unused by LegOS

RAM:

- LegOS
- User Programs
- Thread Stacks ↑
- Heap

Memory mapped I/O:

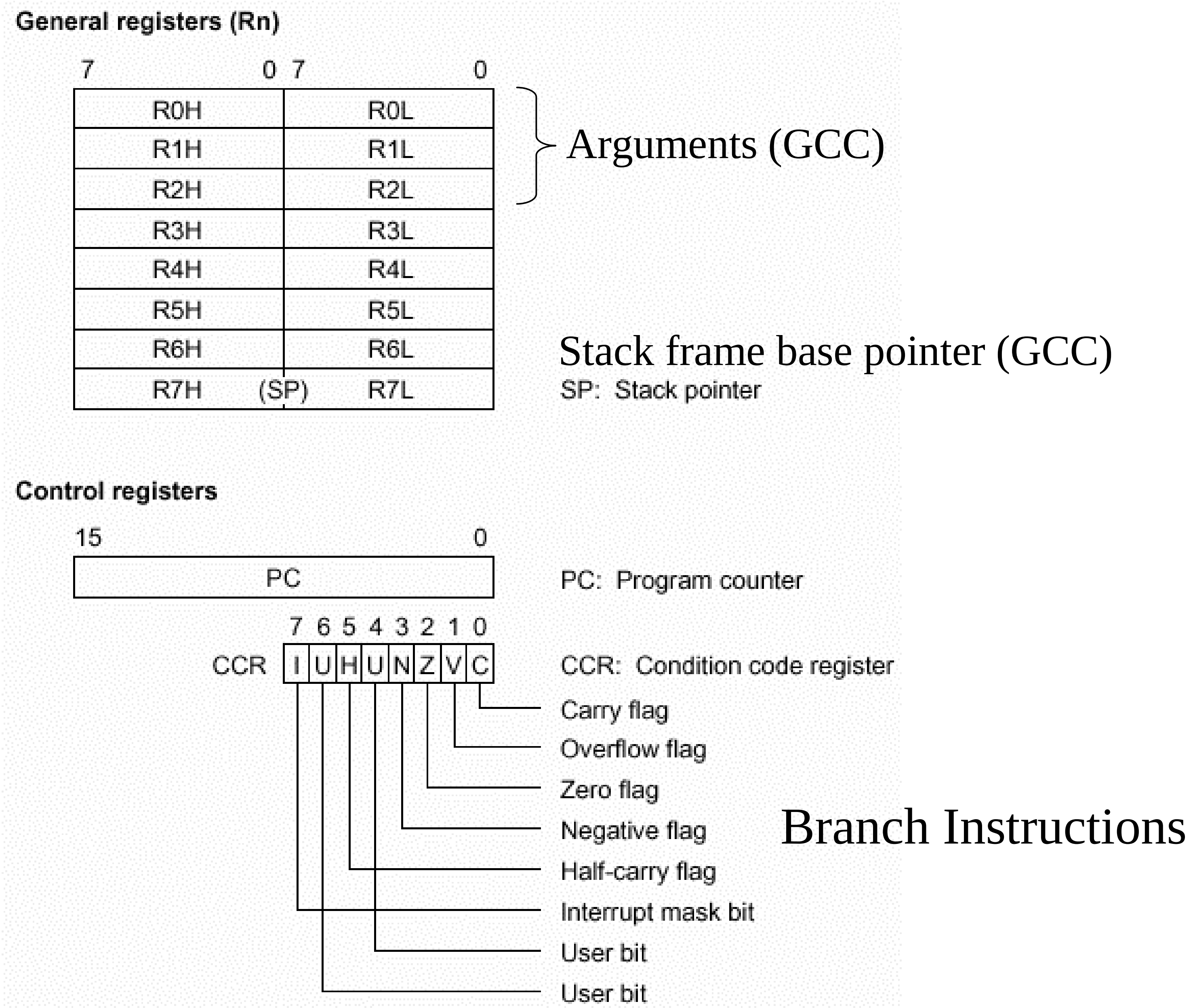
- Motors
- LCD

RAM interrupt vector table

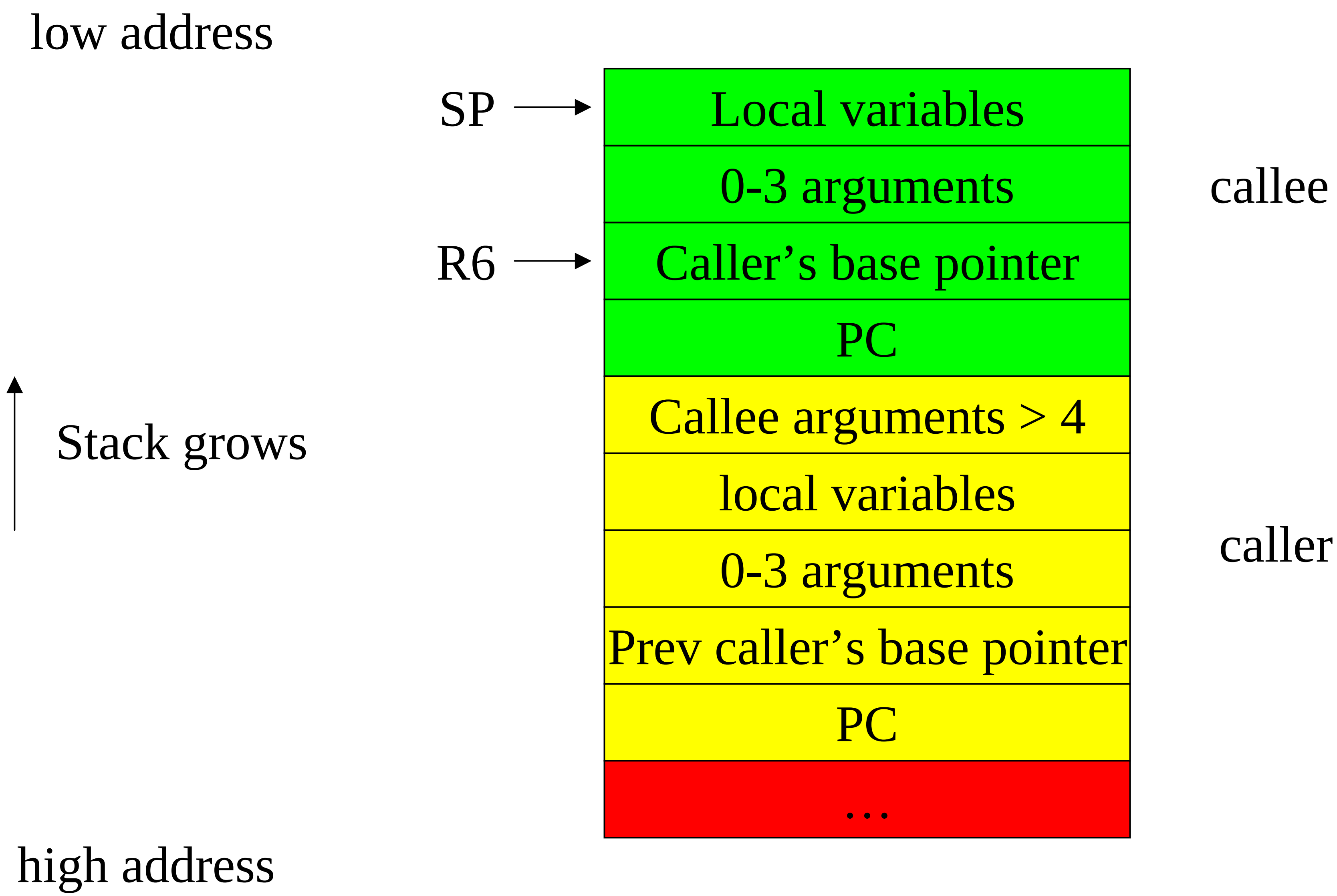
“Magic Numbers” Linker File: h8300.rcx

- Memory Map
  - `ram: o = 0x8000, l = 0x6f30`
- Used ROM Functions
  - `lcd_show = 0x1b62`
- RAM Interrupt Vectors
  - `ocia_vector  = 0x22`
- On-chip Module Registers
  - `T_OCRA    = 0x94`

CPU Registers



GCC Stack Frame



H8/3292 Assembly Example

```
_Increment:                       ;Increment a global variable and call DoNothing

    push r0                       ;Save 16-bit register on stack
    mov.w @_GlobalWord, r0        ;Copy global variable to register
    add.b #0x1, r0L               ;8-bit add 1 to r0L.  Result is in r0L
    mov.w r0, @_GlobalWord        ;Copy register to global variable
    jsr _DoNothing                ;Jump to subroutine.  Push PC, PC = DoNothing
    pop r0                        ;Restore register from stack before returning
    rts                           ;Return from subroutine. Pop PC

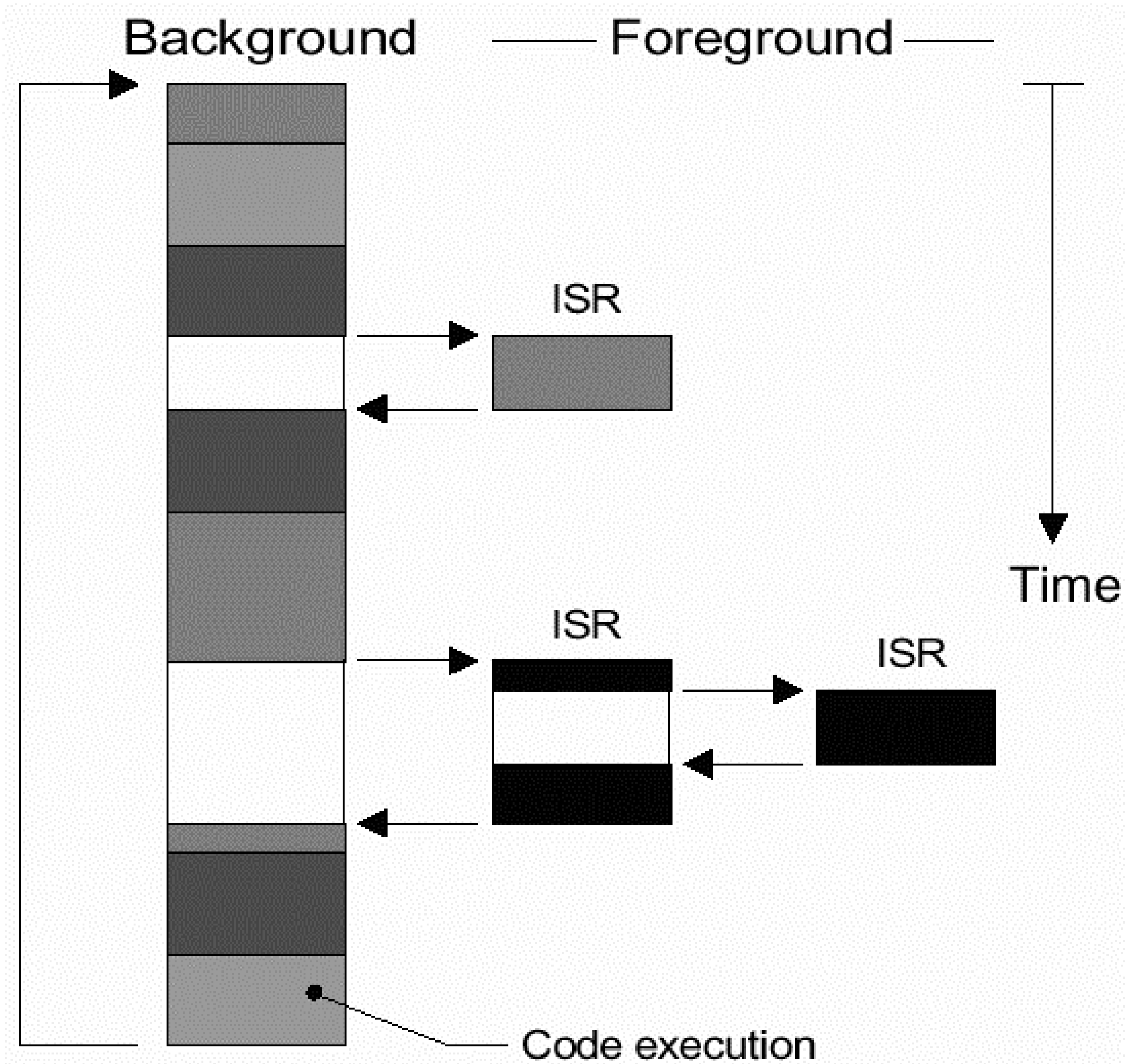


_DoNothing:                       ;Does nothing

    rts                           ;Return from subroutine. Pop PC
```

LegOS = Background + Foreground

Threads
- user
- kernel



Interrupts
- A/D
- timer
- UART

Sensors (A/D Interrupt)

- Touch
- Light
- Rotation

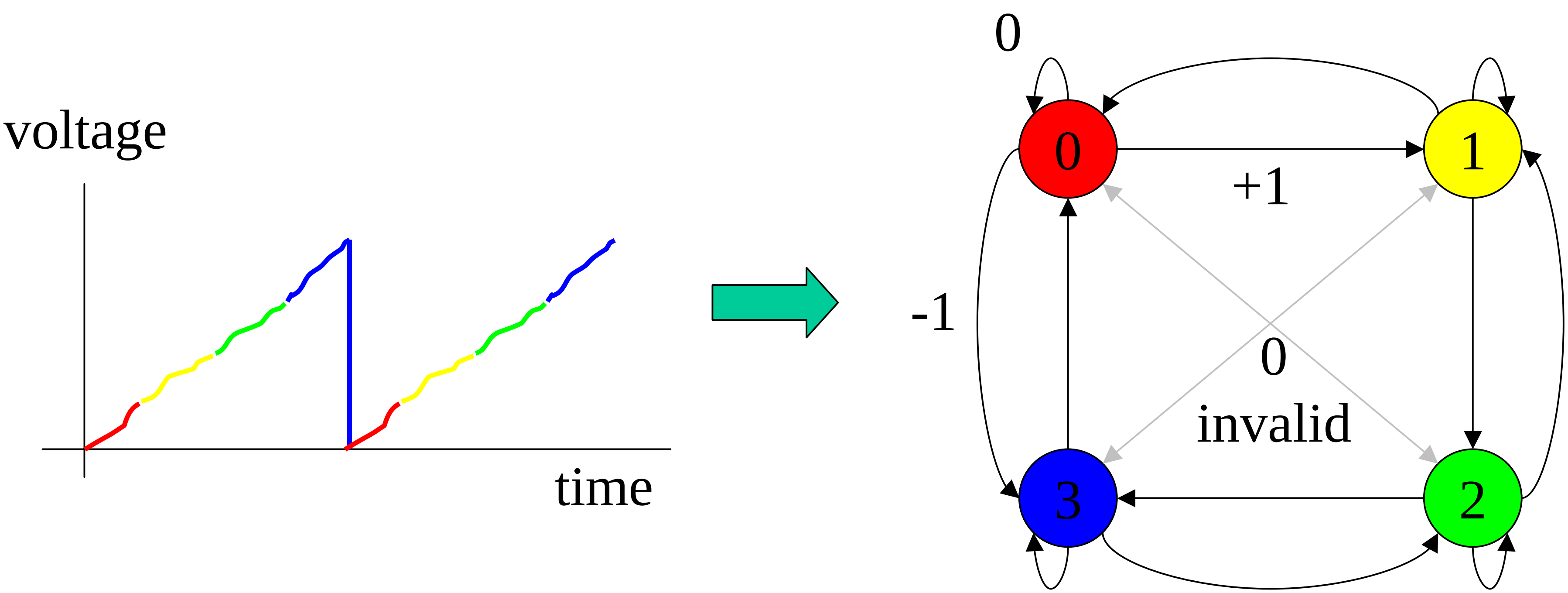
Conversion done →

```
if(rotation)
    run state machine( )
channel++
Start conversion
```

- LIGHT_X and TOUCH_X just "scale" A/D output registers
  => Can use Touch & Light on same input port!
- Rotation = special case: `ds_rotation_on(sensor)`
- ROTATION_X reads position from state machine…

Rotation Sensor State Machine

- Converts repeating analog waveform to absolute position



Works if we can sample fast enough to not miss a state

Timer Interrupt

1 ms →

Increment time

Check LNP inter-byte timeout

Motors

Sound

LCD Refresh

Buttons

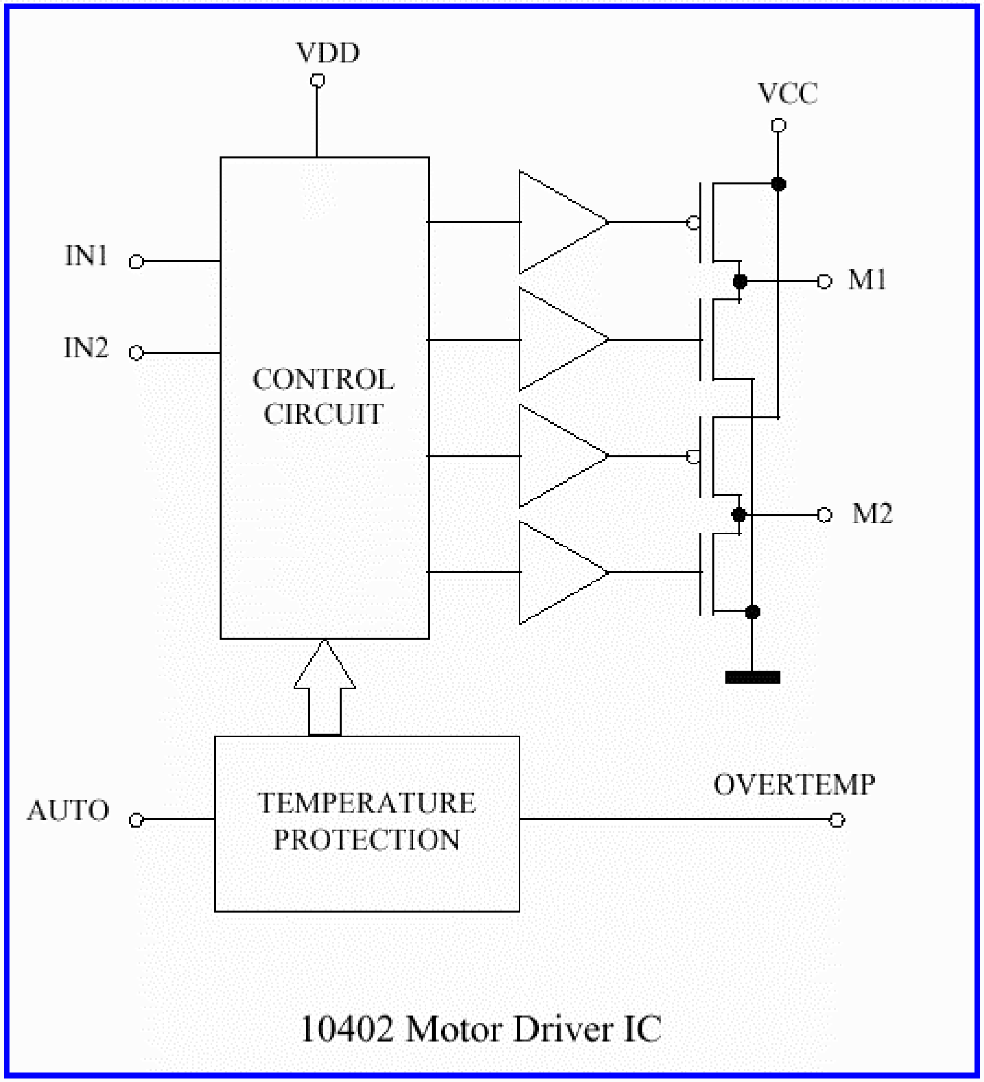
Battery indicator

Thread switching

output

Input
(Poll environment)

Three Monolithic H-Bridges



Memory mapped byte at 0xF000

IN1	IN2	M1	M2	Driving Mode
1	0	1	0	Forward
0	1	0	1	Reverse
1	1	0	0	Brake (Motor shorted)
0	0	Z	Z	Off (Motor disabled)

Melexis Microelectronic Integrated Systems

Motor Handler (Open-loop)

```
struct MotorState{
        char delta; //speed setting (actually torque)
        char sum;   //increment by delta every 1 ms
        char dir;   //2-bit output pattern when sum overflows
};
```

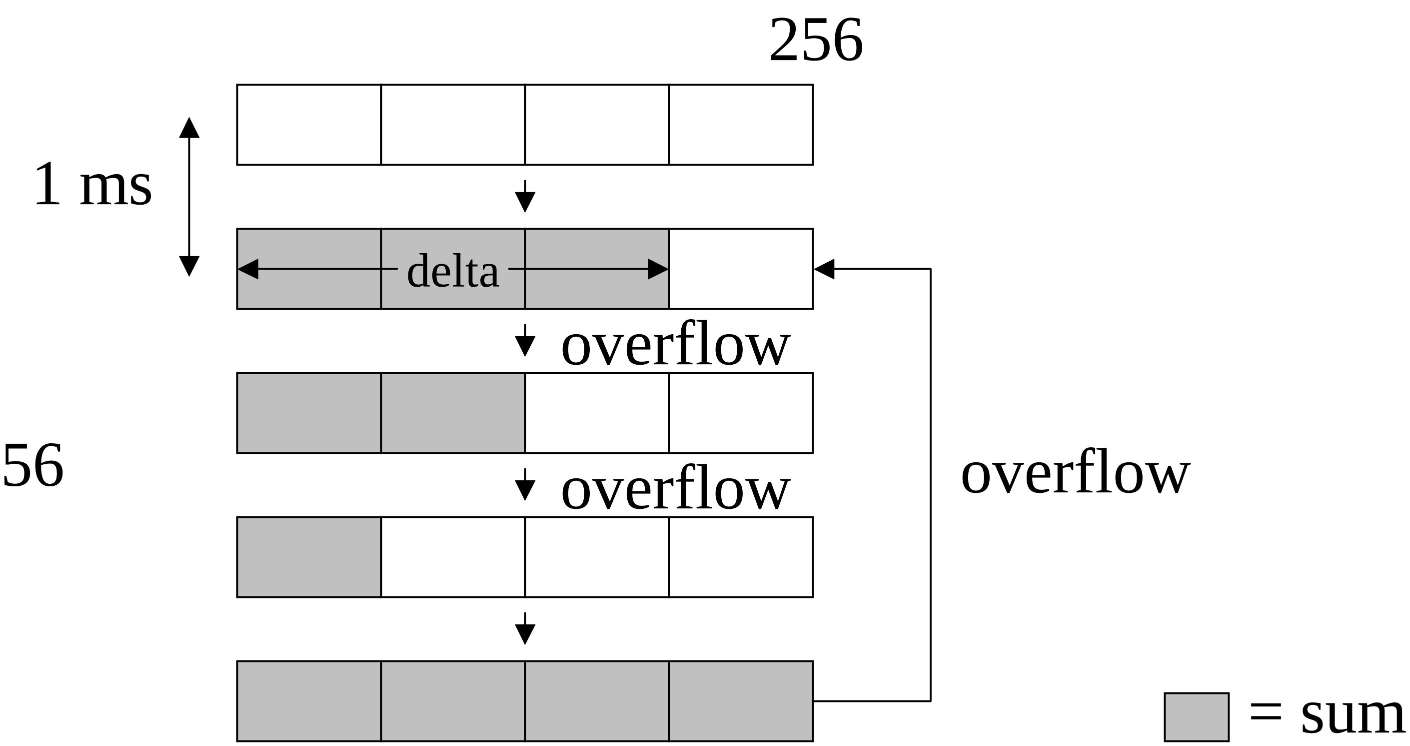
API:

`motor_a_speed()`

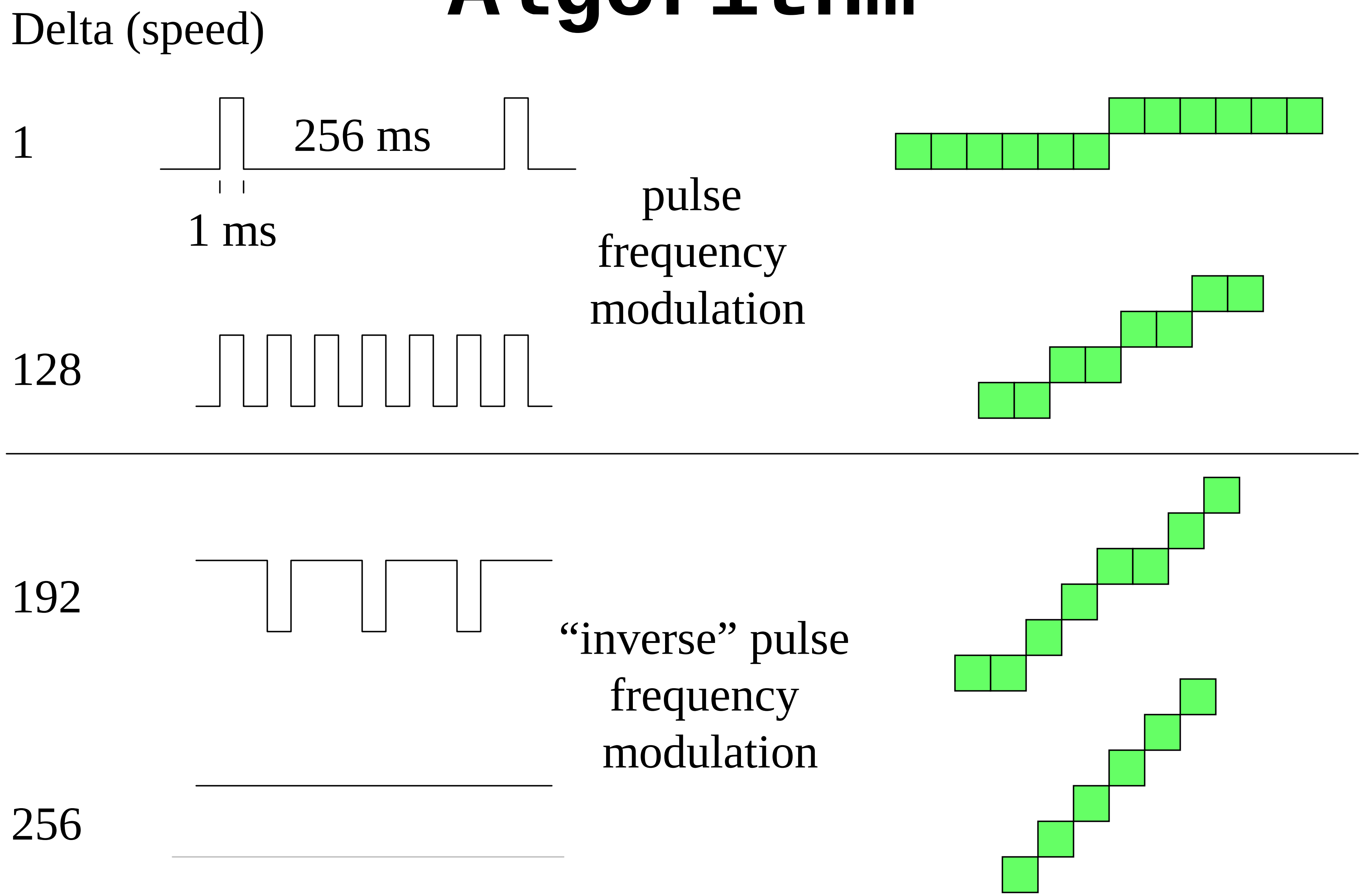
`motor_a_dir()`

Example:

delta = ¾ * 256
= 192



Bresenham's Line Drawing Algorithm



Motor Handler Implementation

(from 1 ms timer interrupt)

```
struct MotorState
{
        char delta; //0
        char sum;   //1
        char dir;   //2
};
```

```
; motor A
        ...
; motor B
        mov.w   @_MotorBState, r0        ; simultaneously load delta and sum
        add.b   r0h, r0l                 ; add delta (r0h) to sum (r0l)
        bcc     NoOvrFl                  ; branch if carry clear (no sum overflow)
        mov.b   @_MotorBState+2, r6h     ; overflow -> output drive pattern (dir)
        xor.b   r6h, r6l                 ; overlay b's output on top of a's
NoOvrFl: mov.b  r0l, @_MotorBState+1     ; save sum (clears overflow flag)

; motor C
        ...

        mov.b   r6l, @0xf000:16          ; output motor waveform
```

Task Management

- Paper Lingo: Task = Process = Thread
- Semaphores
- Structures
- Scheduling Tasks
- Creating New Tasks
- Ending Tasks
- The Life Of A Thread

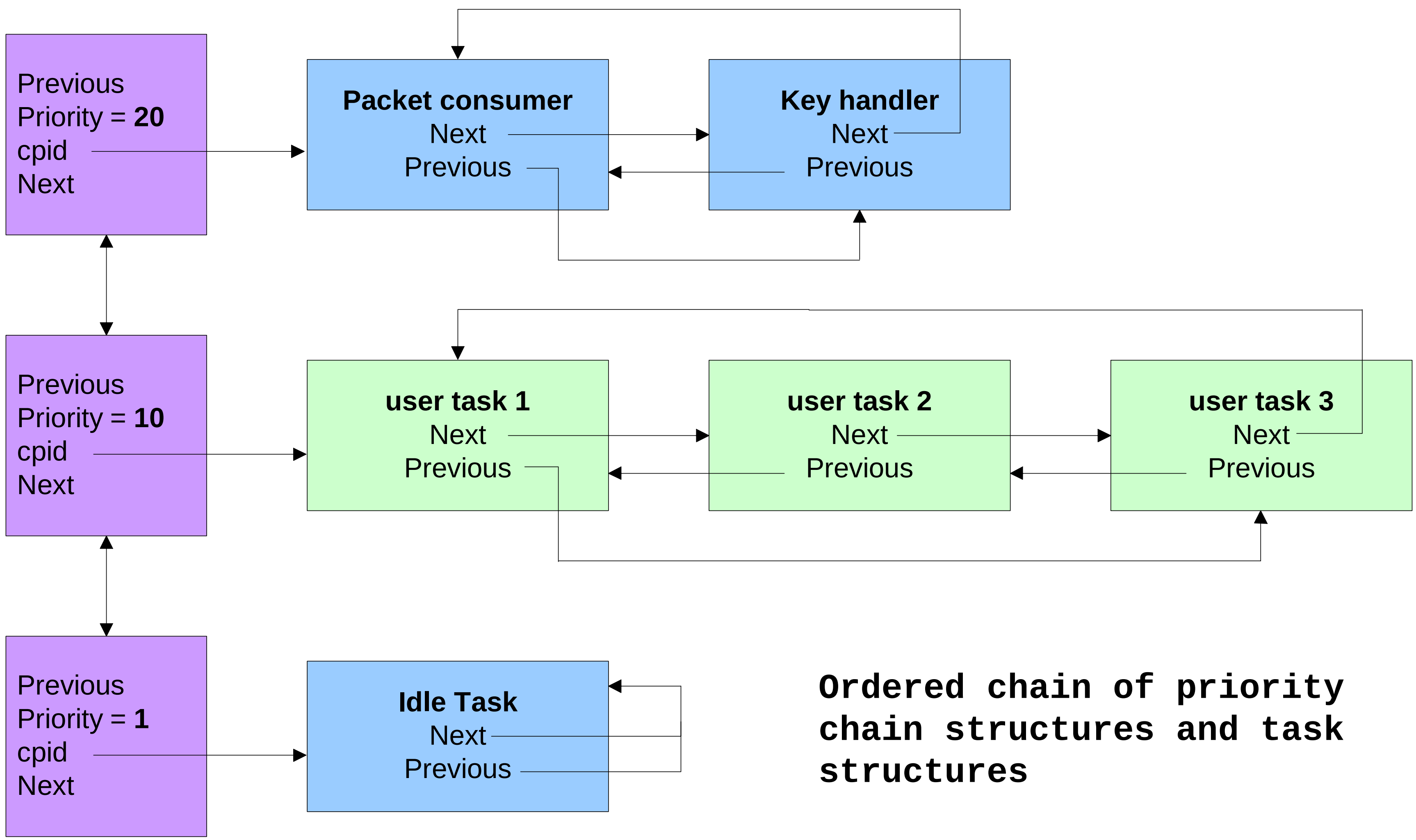
Semaphores-API

- semaphores are POSIX.
  - When count!=0, share info accessible
  - legOS semaphores init with count=1
- Sem_wait
  - suspends calling thread until count!=0, then automatically decreases count
- Sem_trywait
  - non blocking version of sem_wait for interrupt routines. Returns error if count==0
- Sem_post
  - increases count

Kernel Semaphores

- tx_sem
  - transfer access for IR tower etc.
- mm_semaphore
  - memory management for malloc
- task_sem
  - task structure chain for task management

Task Structure Chain



Ordered chain of priority chain structures and task structures

Priority Chain Structure

```
struct _pchain_t
{ char priority;                  // numeric priority level
  struct _pchain_t *next;         // lower priority chain
  struct _pchain_t *prev;         // higher priority chain
  struct _pdata_t  *cpid;         // current process in chain
};
```

Process Data Structure

```
struct _pdata_t
{
  unsigned *sp_save;        // saved stack pointer
  char pstate;              // process state
  char *priority;           // pointer to priority chain
  struct _pdata_t *next;    // next process in queue
  struct _pdata_t *prev;    // previous process in queue
  struct _pdata_t *parent;  // parent process (e.g. main)
  unsigned *stack_base;     // lower stack boundary
  long(*wakeup)(long);      // event wakeup function
  long wakeup_data;          // user data for wakeup fn
};
```

Process States

- **Dead** - The process has terminated and its stack has been freed. Note: No task exists with pstate = dead.
- **Zombie** - The process has terminated, but its stack has not yet been freed.
- **Waiting** - The process is idle and waiting for an event.
- **Sleeping** - The process is idle but **ready** to run.
- **Running** - The process is running.

Wakeups: wait_event

how wakeup fn and data are added to task structure

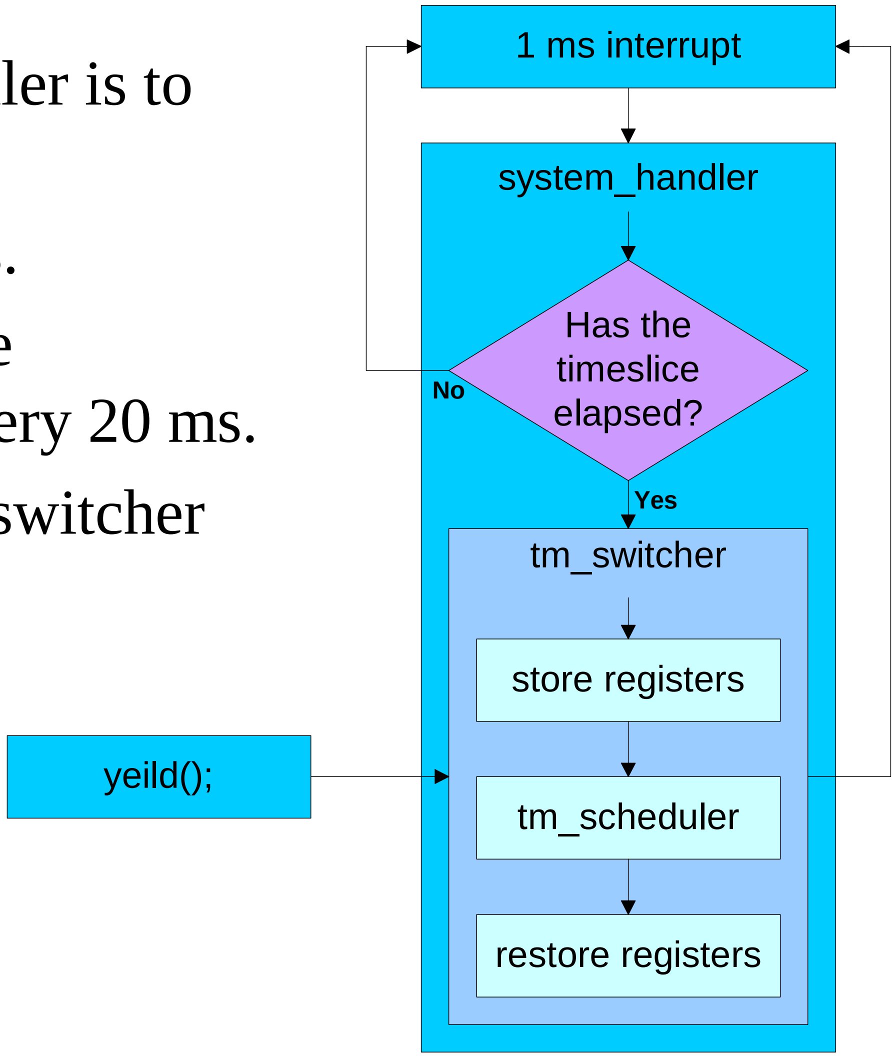
```
long wait_event
(long (*wakeup)(long),long data)
{
  cpid->wakeup      = wakeup;
  cpid->wakeup_data = data;
  cpid->pstate      = P_WAITING;
  yield(); //asm fn that calls tm_switcher
  return cpid->wakeup_data;
}
```

wait_event example: msleep

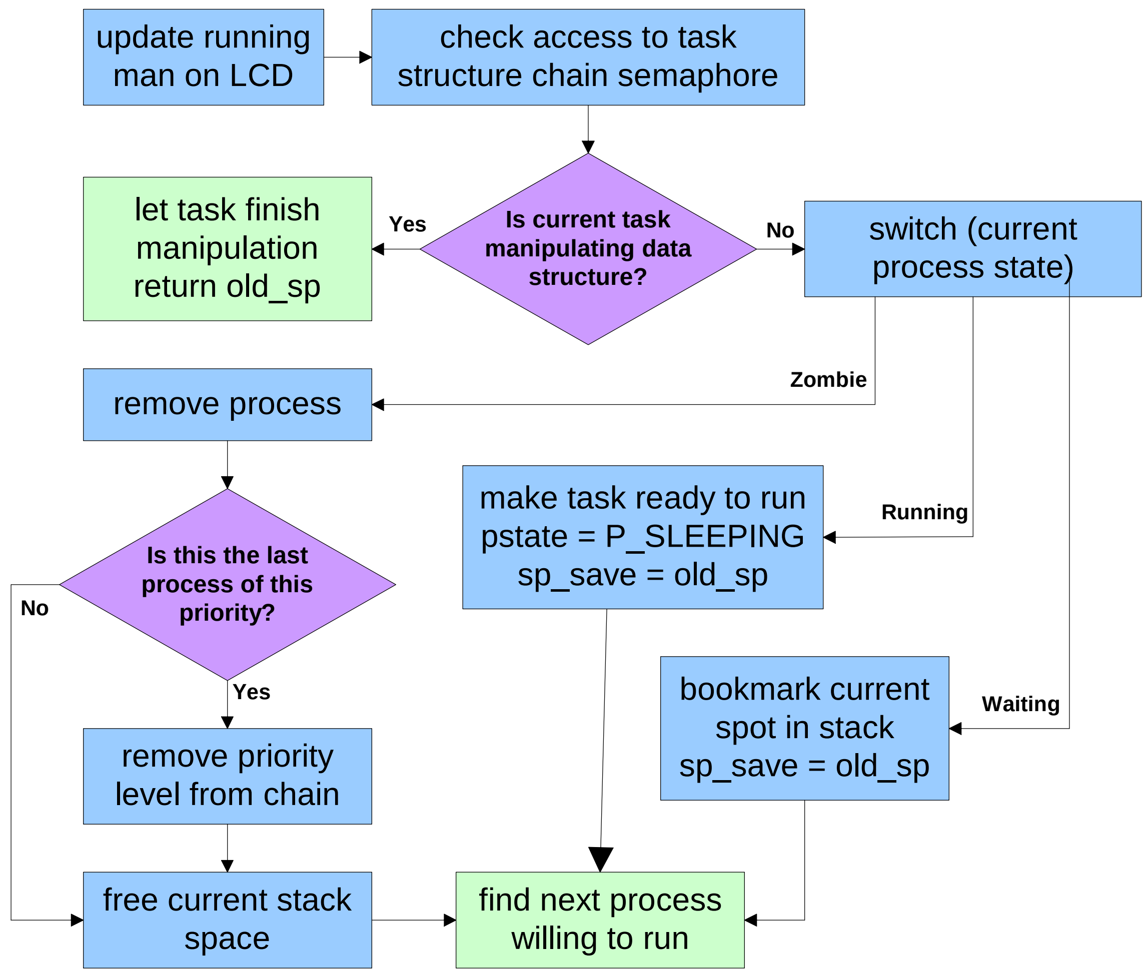
```
int msleep(int msec)
{                  //wait_event(*wakeup,data)
   (void) wait_event(&tm_sleep_wakeup, sys_time + msec);
   return 0;
}
                   //wakeup(data)
Static long tm_sleep_wakeup(long data)
{
   return ((long)data)<=sys_time;
}
```

Scheduling Tasks

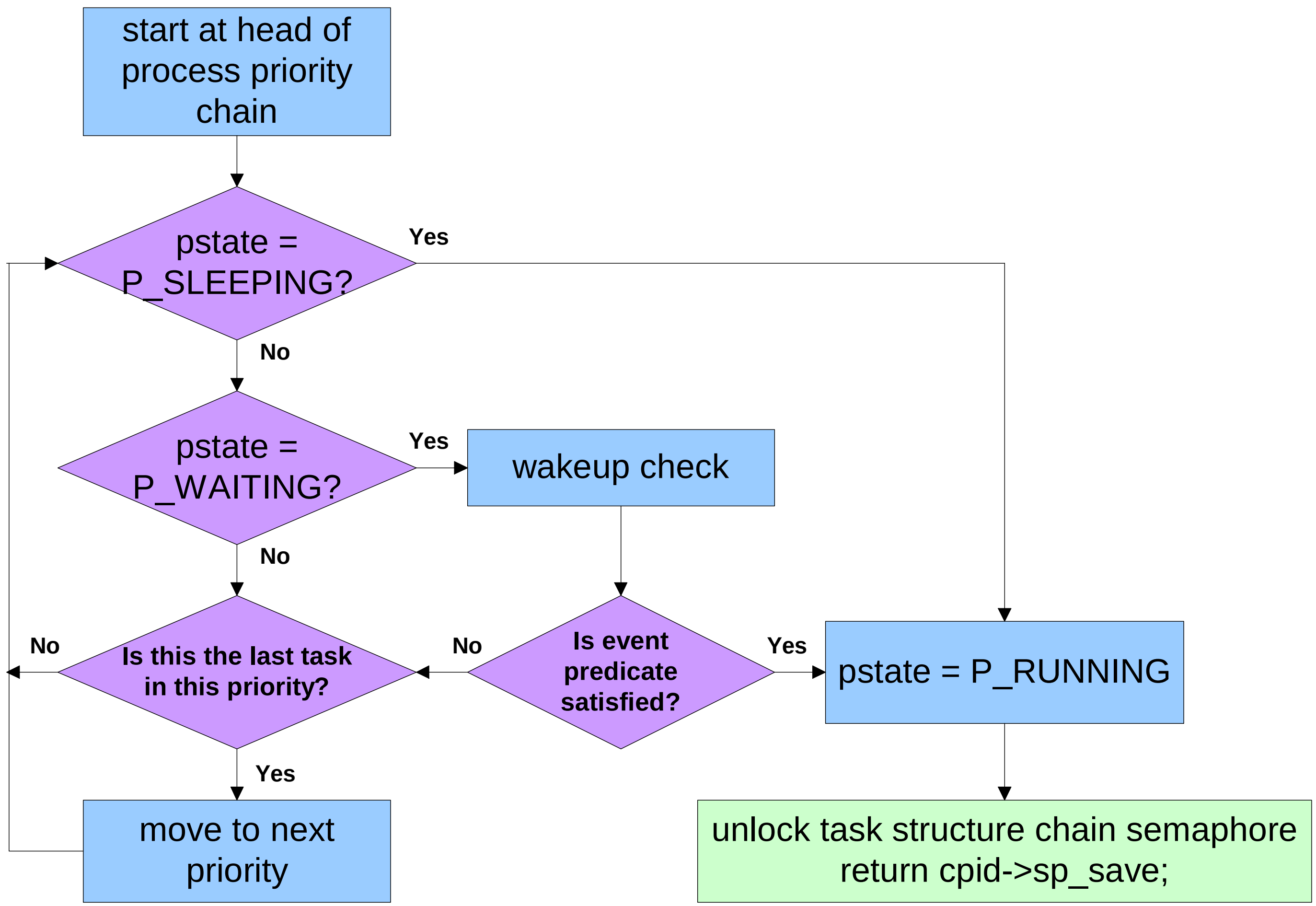
- Last duty of system_handler is to check the timeslice.
- Default timeslice = 20 ms.
- tm_switcher and therefore tm_scheduler is called every 20 ms.
- yeild(); will also call tm_switcher before the timeslice is up.



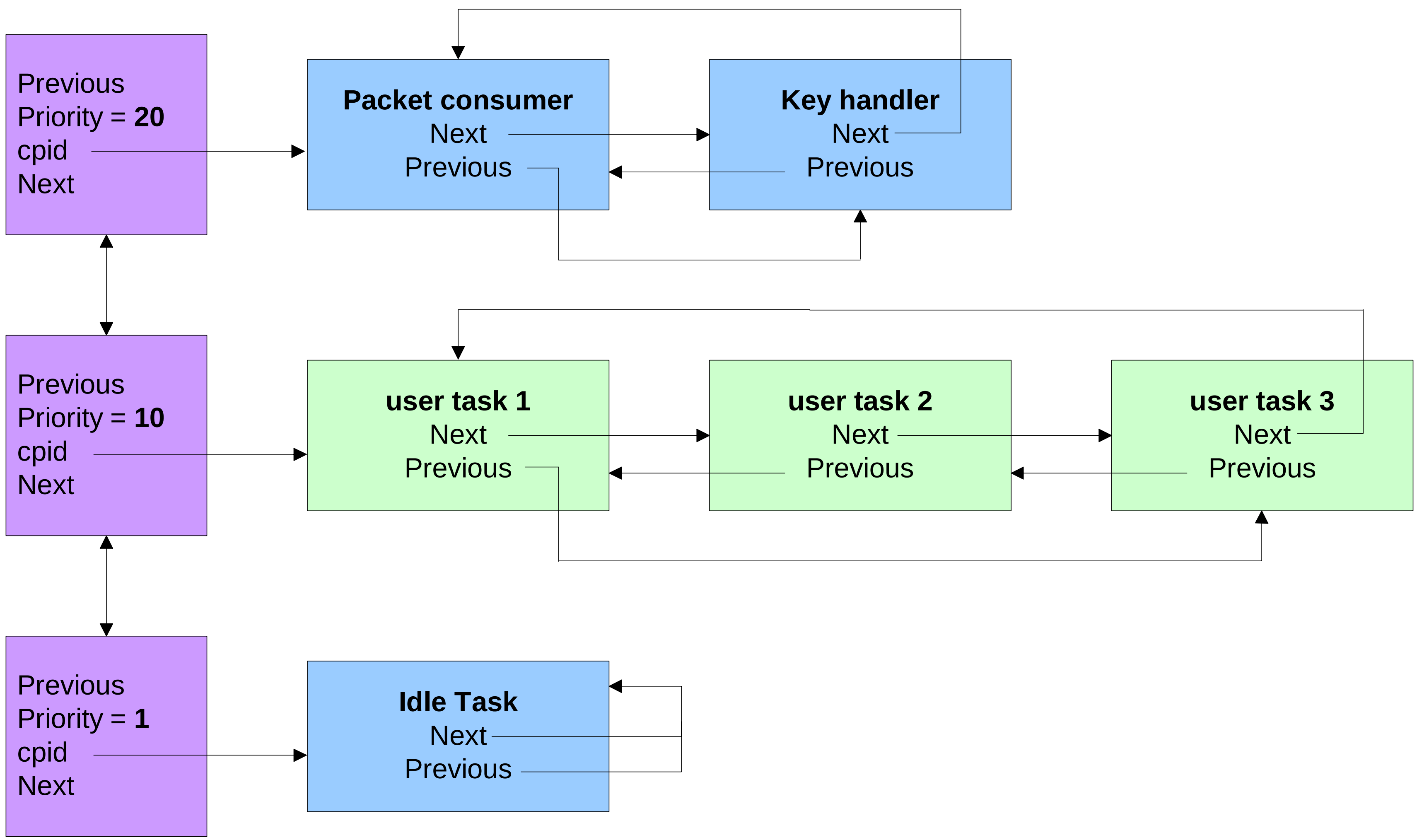
tm_scheduler: assessing current state



`tm_scheduler`: find next process

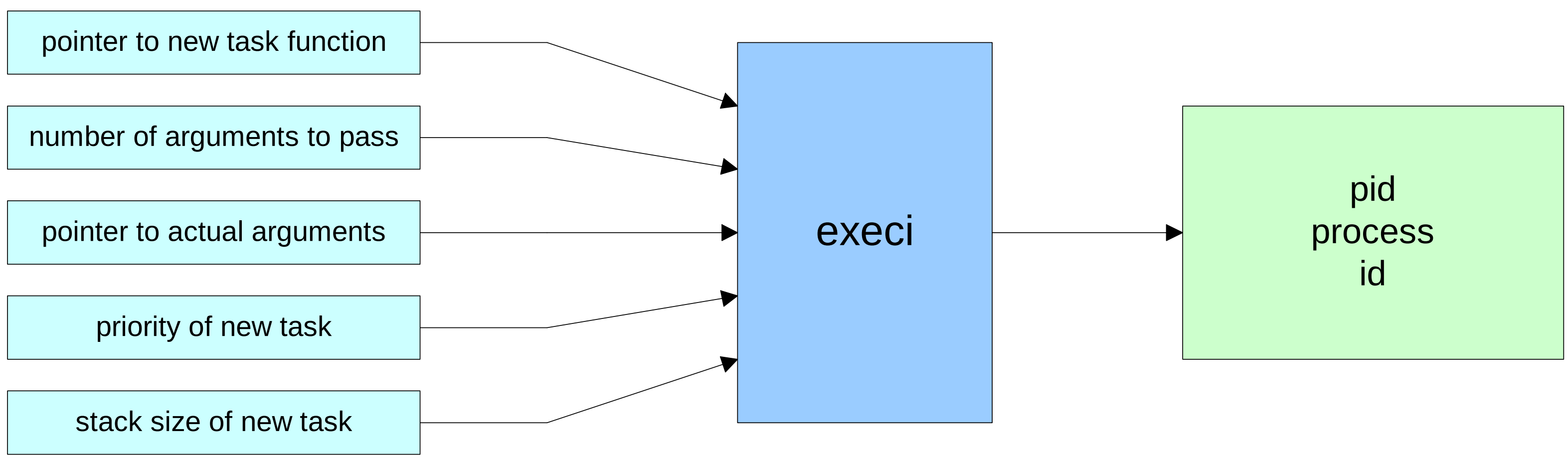


Prioritized Round-Robin

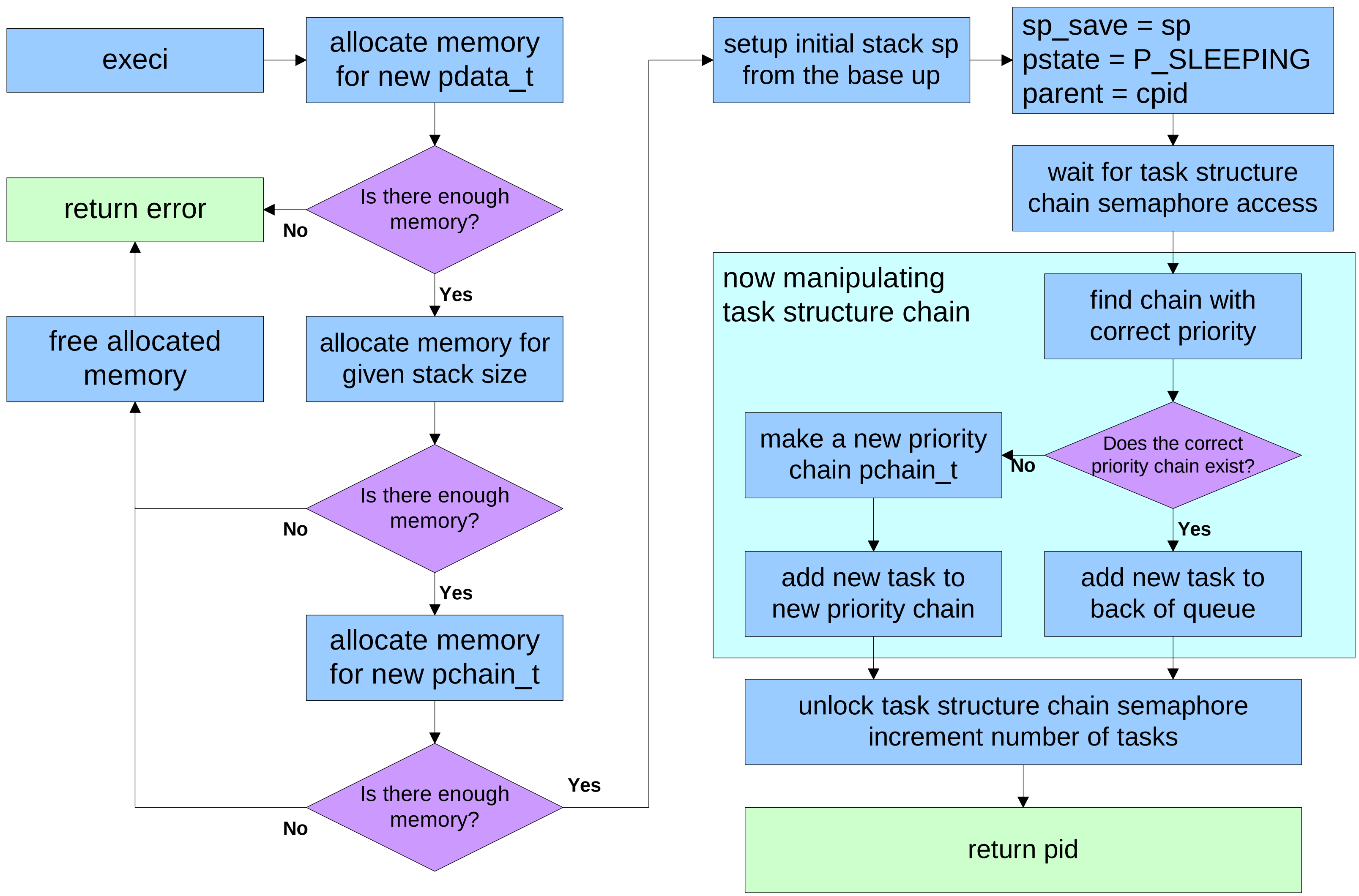


Creating New Tasks

```
unsigned execi
   (int (*code_start)(int,char**), // pointer to new task function
   int argc,                        // number of arguments to pass
   char **argv,                     // pointer to actual arguments
   char priority,                   // priority of new task
   unsigned stack_size)             // stack size of new task
```



The execi function



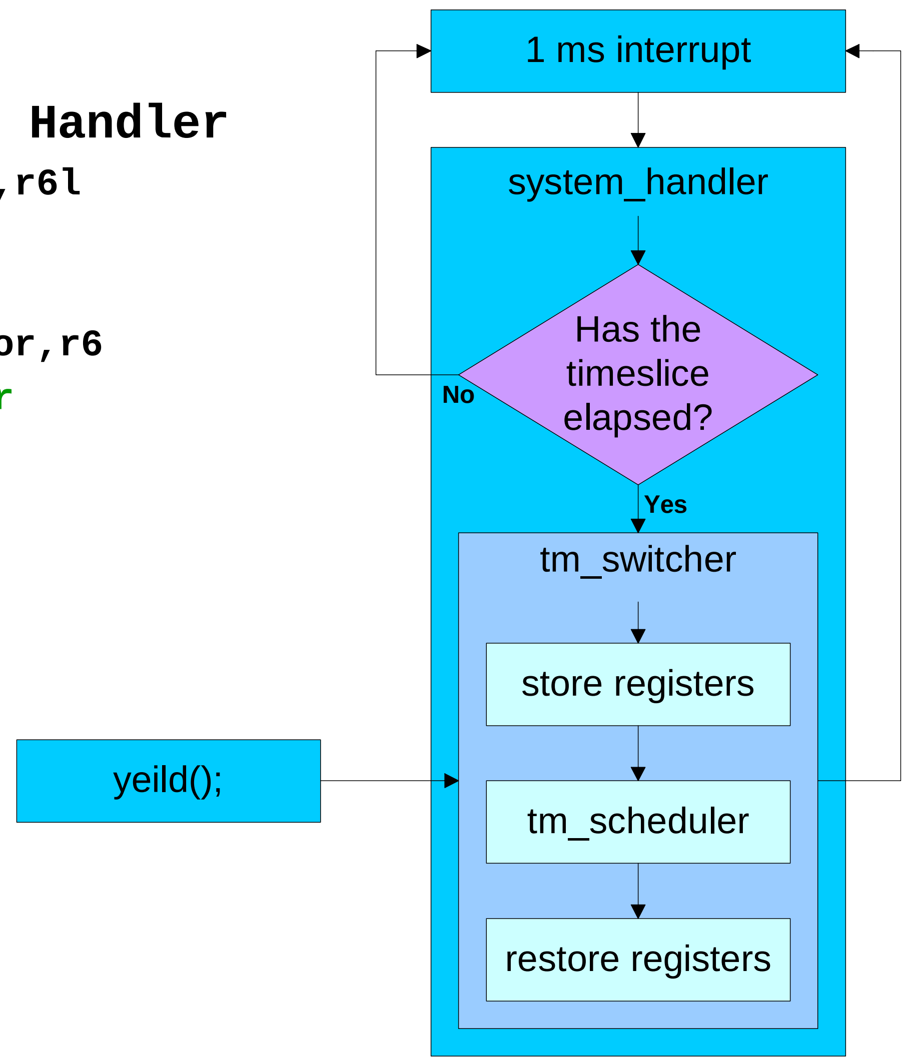
New Task Stack Frame

```
pd->stack_base=sp;                //setup initial stack
sp+=(stack_size>>1);              //from the bottom up to base
*(--sp)=&exit;                    //finish by calling exit
*(--sp)=code_start;               //entry point for task code
*(--sp)=0;                        //ccr for ROM timer interrupt
*(--sp)=0;                        //r6 for ROM timer interrupt
*(--sp)=&rom_ocia_return;         //ROM return of system_handler
*(--sp)=argc;                     //r0 used by system_handler
*(--sp)=&systime_tm_return;       //system return of tm_switcher
*(--sp)=argv;                     //r1
*(--sp)=0;                        //init r2 to 0    tm_switcher
*(--sp)=0;                        //init r3 to 0    registers
*(--sp)=0;                        //init r4 to 0
*(--sp)=0;                        //init r5 to 0
```

Exploring the Stack Frame

Last Part of System Handler

```
mov.b @_tm_current_slice,r6l
dec r6l
bne sys_noswitch
mov.w @_tm_switcher_vector,r6
jsr @r6 ;call tm_switcher
        .
        .
        .
        .
ret
```



Exploring the Stack Frame

tm_switcher (r7 = sp)

```
_tm_switcher:
mov.w    r1,@-r7  ; save registers
mov.w    r2,@-r7  ; from the current
mov.w    r3,@-r7  ; task stack frame
mov.w    r4,@-r7
mov.w    r5,@-r7
mov.w    r7,r0    ; arg for tm_scheduler
jsr_tm_scheduler ; call tm_scheduler
                  ; ret from tm_scheduler
_tm_switcher_return:
mov.w    r0,r7    ; set new sp
mov.w    @r7+,r5  ; restore registers
mov.w    @r7+,r4
mov.w    @r7+,r3
mov.w    @r7+,r2
mov.w    @r7+,r1

rts               ; return to new task
```

New Task Stack Frame

```
pd->stack_base=sp;
sp+=(stack_size>>1);
*(--sp)=&exit;
*(--sp)=code_start;
*(--sp)=0;
*(--sp)=0;
*(--sp)=&rom_ocia_return;
*(--sp)=argc;     //R0
*(--sp)=&systime_tm_return;
*(--sp)=argv;     //R1
*(--sp)=0;        //R2
*(--sp)=0;        //R3
*(--sp)=0;        //R4
*(--sp)=0;        //R5
```

Start of new sp is the stack base sp_save returned from tm_scheduler

Exploring the Stack Frame

Last Part of System Handler

```
mov.b @_tm_current_slice,r6l
dec r6l
bne sys_noswitch
mov.w @_tm_switcher_vector,r6
jsr @r6
;return from tm_switcher
_systime_tm_return:
   mov.b @_tm_timeslice,r6l
sys_noswitch:
   mov.b r6l,@_tm_current_slice
pop r0
; reset compare A IRQ flag
bclr    #3,@0x91:8
rts ;ret to rom_ocia_return
```

New Task Stack Frame

```
pd->stack_base=sp;
sp+=(stack_size>>1);
*(--sp)=&exit;
*(--sp)=code_start;
*(--sp)=0;
*(--sp)=0;
*(--sp)=&rom_ocia_return;
*(--sp)=argc; //R0
*(--sp)=&systime_tm_return;
*(--sp)=argv;
*(--sp)=0;
*(--sp)=0;
*(--sp)=0;
*(--sp)=0;
```

Exit function

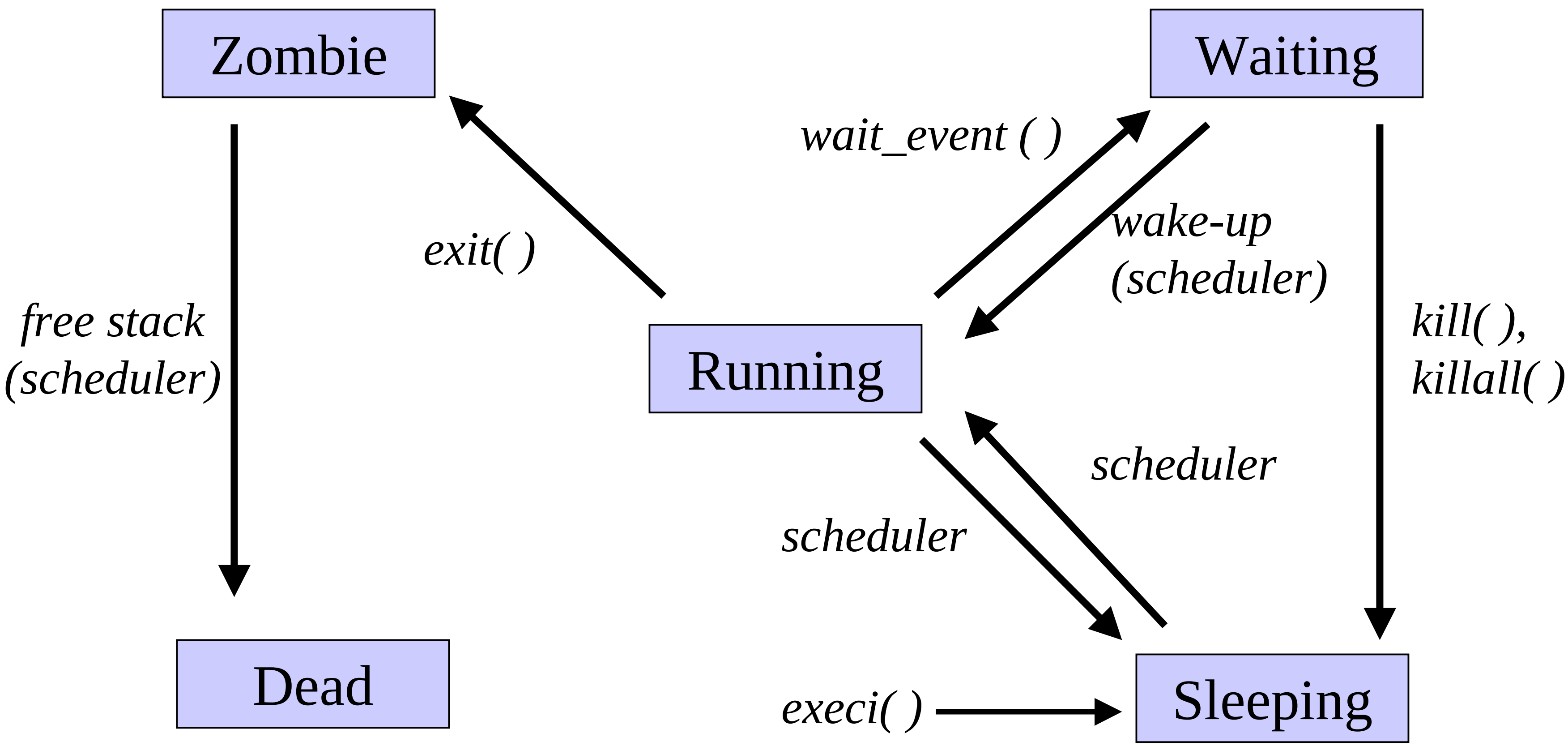
```
void exit(void)
{
  enable_irqs();          //enable interrupts just in
                          //case the task disabled them
  mm_reaper();            //free all blocks allocated
                          //by the current process
  cpid->pstate = P_ZOMBIE;//ready to be deallocated
  while(1)
      yield();            //call tm_switcher before
                          //timeslice is up
}
```

Kill Function

```
void kill(unsigned pid) //process ID
{
   pdata_t *pd=(pdata_t*) pid; //setup a pointer to task to kill
   if(pd==cpid)         //if task to kill is currently running
        exit(-1);       //exit immediately
   else                 //set up sp_save such that the next time
   {                    //the task runs it will exit immediately.
   sem_wait(&task_sem); //wait for semaphore access
   *( (pd->sp_save) + SP_RETURN_OFFSET )=&exit;
   pd->pstate=P_SLEEPING; //make ready to run in case waiting
   sem_post(&task_sem); //unlock semaphore access
   }
}
```

killall(priority) will kill all tasks in the specified priority

The Life of a LegOS Process

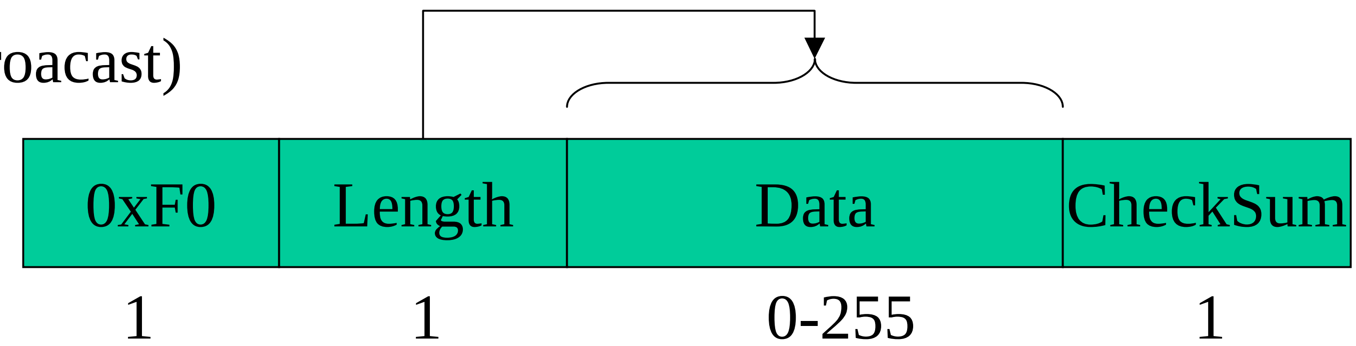


LNP (LegOS Network Protocol)

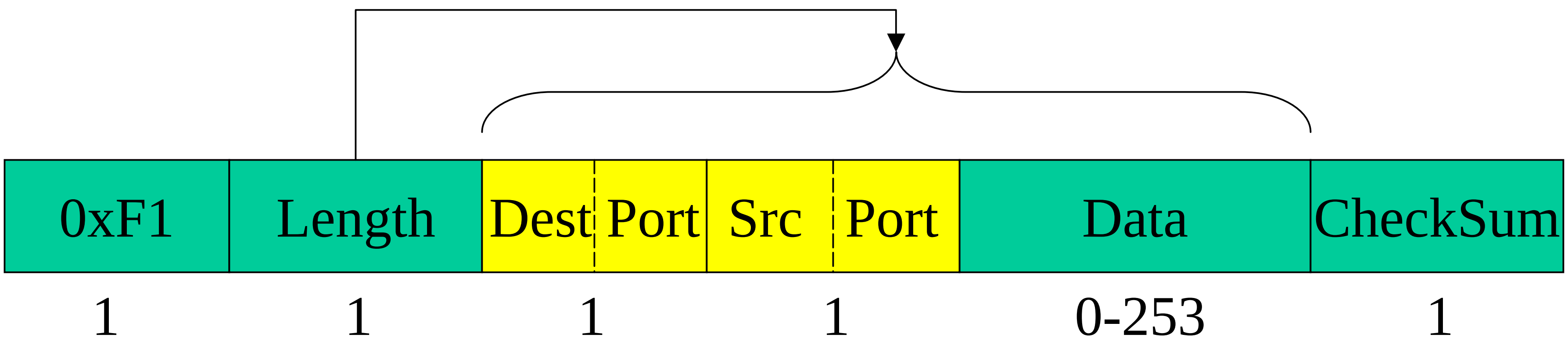
- UDP-like
  - No arrival guarantees (no replies/retries)
  - Packets that do arrive will be error-free
- Two packet types
  - “Integrity” = broadcast
  - “Addressing” = unicast
- Up to 16 nodes and 16 ports
- Port 0 reserved for program loading

Packet formats

Integrity (broacast)



Addressing (unicast)



LNP API

- Receive
  - `lnp_addressing_set_handler(MY_PORT, MyRxHandler)`
  - `MyRxHandler(char* Data, char Length, char Source)`
    - One per port (+1 for broadcast)
    - Will be called from an interrupt, so pass Data to thread
- Transmit
  - ```
    Collision = lnp_addressing_write(char *Data,
                     char Length, char DestAddrAndPort);
    ```
    - Blocks until entire packet is sent
- RCX Address: CONF_LNP_HOSTADDR = 0
  - Must recompile LegOS to change :(
  - PC Address = 8
    ```

Four LNP ISRs

- Received a byte
  - Reset inter-byte timeout
  - if(receiving)
    - Store incoming byte
    - if(end of packet) call handler
  - else //transmitting
    - Check for collisions
- Receive error (e.g. parity)
  - if(receiving)
    - Discard entire packet
  - else //transmitting
    - collision
- Transmit buffer available
  - Insert next byte (if there is one)
- Done transmitting

The End

- Hardware
- Assembly Language
- Motor and Sensor Handling
- Task Management: Threading
- Network