M2PGi – M2M

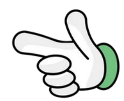
Pr. Olivier Gruber (olivier.gruber@imag.fr)

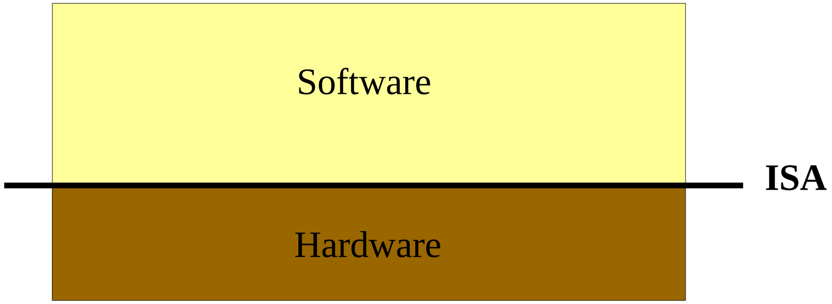
Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Today

- Hardware Support for Interrupts
  - Halt instruction, Processor modes, Interrupt Requests, Exception vectors, etc.
- Software Support for Interrupts
  - Handling interrupts
  - ***Race condition challenge***
    & Solutions...





Reminder – Coding with Interrupts


```
uint8_t button_get_status(void* bar);
void led_on(void* bar);
void led_off(void* bar);


void start() {

  button_init_regs(BUTTON_BAR);
  led_init_regs(LED_BAR);
  for (;;) {
     uint8_t status;
     status = button_get_status(BUTTON_BAR);
     if (status & 0x01)
       led_on(LED_BAR);
      else
       led_off(LED_BAR);
  }
}
```

```
void button_interrupt_handler(void* cookie) {
  uint8_t status;
  status = button_get_status(BUTTON_MMIO_BAR);
  if (status & 0x01)
    led_on(LED_MMIO_BAR);
  else
    led_off(LED_MMIO_BAR);
}

void _start() {
  init_interrupt_handlers();
  button_init_regs(BUTTON_MMIO_BAR);
  led_init_regs(LED_MMIO_BAR);
  vic_enable_interrupt(BUTTON_IRQ,
                       button_interrupt_handler,
                       null);
  button_enable_interrupt();
  core_enable_interrupts();
  for (;;) {
    halt();
  }
}
```

Safe transformation:

- the main loop only sleeps => ***no race condition***
- the ***handler is short*** and we are processing only one interrupt

Reminder – Analysis – Console & Shell


```
void shell(char* line, int length);

char line[80];
int offset = 0;

void _start() {

  uart_init(UART0);
  for (;;) {
    uint8_t code = uart_receive(UART0);
    while (code) {
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        offset=0;
      } else
        line[offset++]=(char)code;
      code = uart_receive(UART0);
    }
  }
}
```

```
void shell(char* line, int length);

char line[80];
int offset = 0;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line, offset);
      offset=0;
    } else
      line[offset++]=(char)code;
    code = uart_receive(UART0);
  }
}

void _start() {
  init_interrupt_handlers();
  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    core_halt();
  }
}
```

Transformation:

- No more spinning waiting for received bytes
- Requires a non-blocking uart receive
- Still potentially spinning, waiting for room to write bytes

Safe transformation?

A Solution – Using Interrupts


```
void shell(char* line, int length);

char line[80];
int offset = 0;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line, offset);
      offset=0;
    } else
      line[offset++]=(char)code;
    code = uart_receive(UART0);
  }
}

void _start() {

  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();


  for (;;) {
    core_halt();
  }
}
```

Not really! Handlers must be short!

This one may be long running:
- spinning waiting for room to transmit
- long-running shell calls
- NO INTERRUPTS DURING THE EXECUTION OF THE LINE BY THE SHELL!

So ***we may still loose bytes***, received by the UART...

Also, what if we had to manage several devices?
Long handlers delay the processing of other interrupts because interrupts are disabled!

Analysis – Need to Rework the Code...


```
void shell(char* line, int length);

char line[80];
int offset = 0;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line, offset);
      offset=0;
    } else
      line[offset++]=(char)code;
    code = uart_receive(UART0);
  }
}

void _start() {

  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();


  for (;;) {
    // the bulk of the work needs to be here
    core_halt();
  }
}
```

The handler must be short, just tending to the device.

The bulk of the work must go back to the main loop.

But how?

Let's try to do it...

Safe Transformation?


```
void shell(char* line, int length);

char line[80];
int offset = 0;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line, offset);
      offset=0;
    } else
      line[offset++]=(char)code;
    code = uart_receive(UART0);
  }
}

void _start() {

  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();

  for (;;) {
    core_halt();
  }
}
```

```
void shell(char* line, int length);

list_of_chars_t *chars;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    list_append(chars,(char)code);
    code = uart_receive(UART0);
  }
}

void _start() {
  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();

  char line[80];
  int  nchars = 0;
  for (;;) {
    while (!list_empty(line)) {
      uint8_t code = list_remove(chars,0);
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        nchars=0;
      } else
        line[nchars++] = code;
    }
    core_halt();
  }
}
```

What do we think? Is this a safe transformation?

Safe Transformation?

TOTALLY UNSAFE!

There is a race condition around the manipulation of the list...

We do need a communication channel between the handler and the main loop, but we need a ***correct solution***...

```
void shell(char* line, int length);

list_of_chars_t *chars;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    list_append(chars,(char)code);
    code = uart_receive(UART0);
  }
}

void _start() {
  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();

  char line[80];
  int  nchars = 0;
  for (;;) {
    while (!list_empty(line)) {
      uint8_t code = list_remove(chars,0);
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        nchars=0;
      } else
        line[nchars++] = code;
    }
    core_halt();
  }
}
```

Safe Transformation?


```
void shell(char* line, int length);

list_of_chars_t *chars;

void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    list_append(chars,(char)code);
    code = uart_receive(UART0);
  }
}

void _start() {
  uart_init(UART0,uart_rx_handler);
  core_enable_interrupts();

  char line[80];
  int  nchars = 0;
  for (;;) {
    while (!list_empty(line)) {
      core_disable_interrupts();
      uint8_t code = list_remove(chars,0);
      core_enable_interrupts();
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        nchars=0;
      } else
        line[nchars++] = code;
    }
    core_halt();
  }
}
```

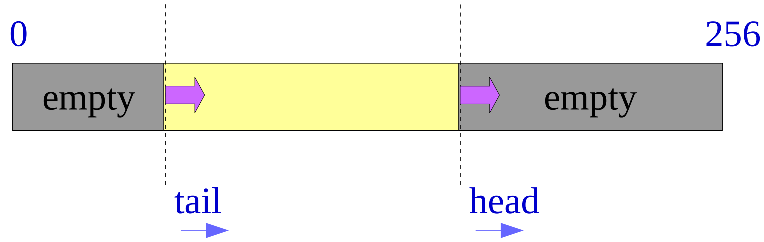
A safe solution, disabling interrupts around critical sections of code.

Remember: interrupts must be disable for short period of time.

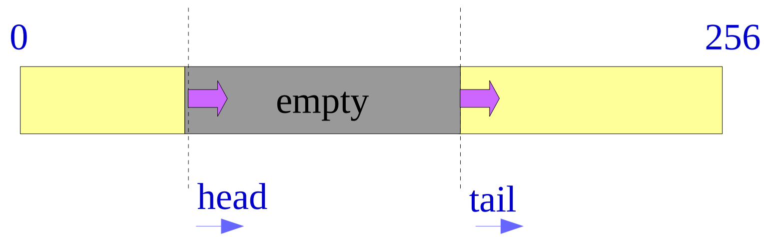
But there is also a data structure that can be used in this context, in a lock-free manner.

Lock-free Circular Buffer – Lock-Free Ring


```
#define MAX_CHARS 512
volatile uint32_t tail = 0;
volatile uint8_t buffer[MAX_CHARS];
volatile uint32_t head = 0;

bool_t ring_empty() {
  return (head==tail);
}

bool_t ring_full() {
  int next = (head + 1) % MAX_CHARS;
  return (next==tail);
}

void ring_put(uint8_t bits) {
  uint32_t next = (head + 1) % MAX_CHARS;
  buffer[head] = code;
  head = next;
}

uint8_t ring_get() {
  uint8_t bits;
  uint32_t next = (tail + 1) % MAX_CHARS;
  bits = buffer[tail];
  tail = next;
  return bits;
}
```



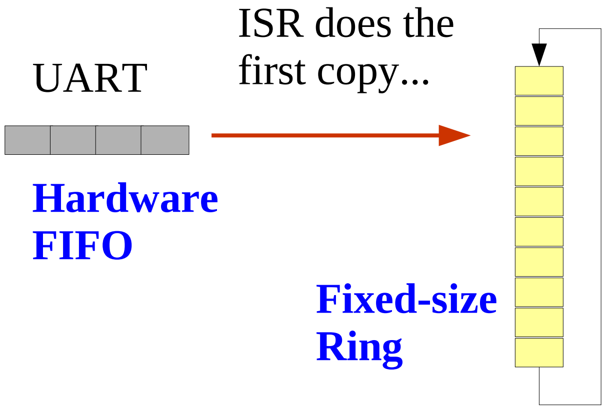
Circular Buffer:
Acts as a communication channel between the ISR and the main loop



Using a Ring – A nice solution...


```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}

void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
    core_halt();
  }
}

char line[MAX_CHARS];
uint32_t nchars = 0;

void process_ring() {
  uint8_t code;
  while (!ring_empty()) {
    code = ring_get();
    line[nchars++]=(char)code;
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line,nchars);
      nchars = 0;
    }
  }
}
```

Add bytes to the ring...



Most of the work is back on the main loop!

Read characters from the buffer, split them into lines, and passed them to the Shell for interpretation.

Notice that uart_send can spin now...

Using a Ring – A nice solution, but...


```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}

void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
 ------------------------------------------------
    core_halt();
  }
}

char line[MAX_CHARS];
uint32_t nchars = 0;

void process_ring() {
  uint8_t code;
  while (!ring_empty()) {
    code = ring_get();
    line[nchars++]=(char)code;
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line,nchars);
      nchars = 0;
    }
  }
}
```

Is this correct though?

Don’t we risk not processing the last received characters?

Last interrupt here...

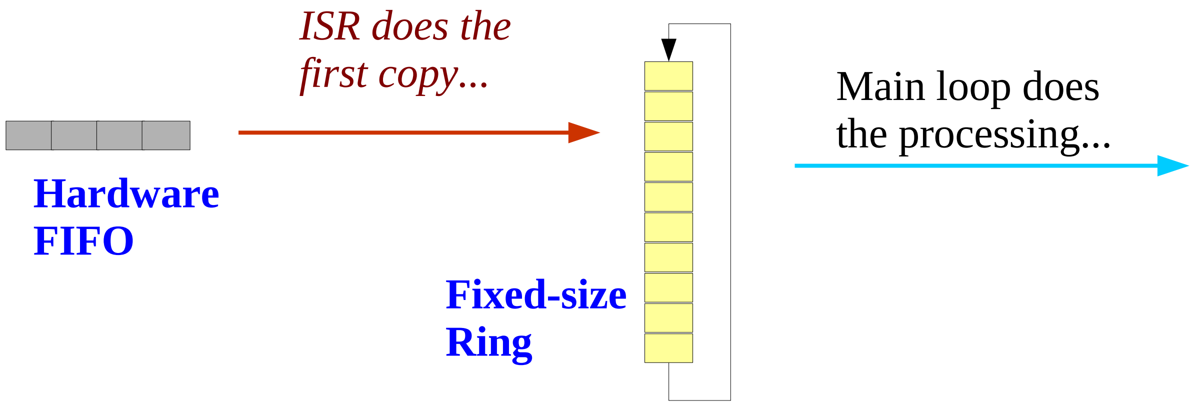
Using a Ring – A correct and simple solution


```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}

void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
    core_disable_interrupts();
    if (ring_empty())
      core_halt();
  }
}

char line[MAX_CHARS];
uint32_t nchars = 0;

void process_ring() {
  uint8_t code;
  while (!ring_empty()) {
    code = ring_get();
    line[nchars++]=(char)code;
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line,nchars);
      nchars = 0;
    }
  }
}
```

A correct and simple solution.

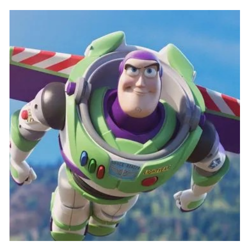
It relies on a property of the instruction to halt a core, like “wfi” on ARM. The instruction re-enables interrupts before going to sleep, and only if there are no pending interrupts

Overall architecture based on using rings




```c
void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
    core_disable_interrupts();
    if (ring_empty())
      core_halt();
  }
}
```

Mr. UART
(hardware)

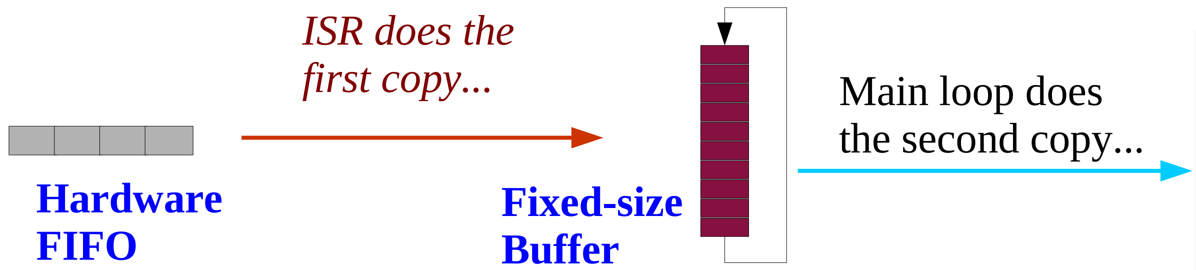


Mr. ISR
(interrupt service routine)

```c
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}
```

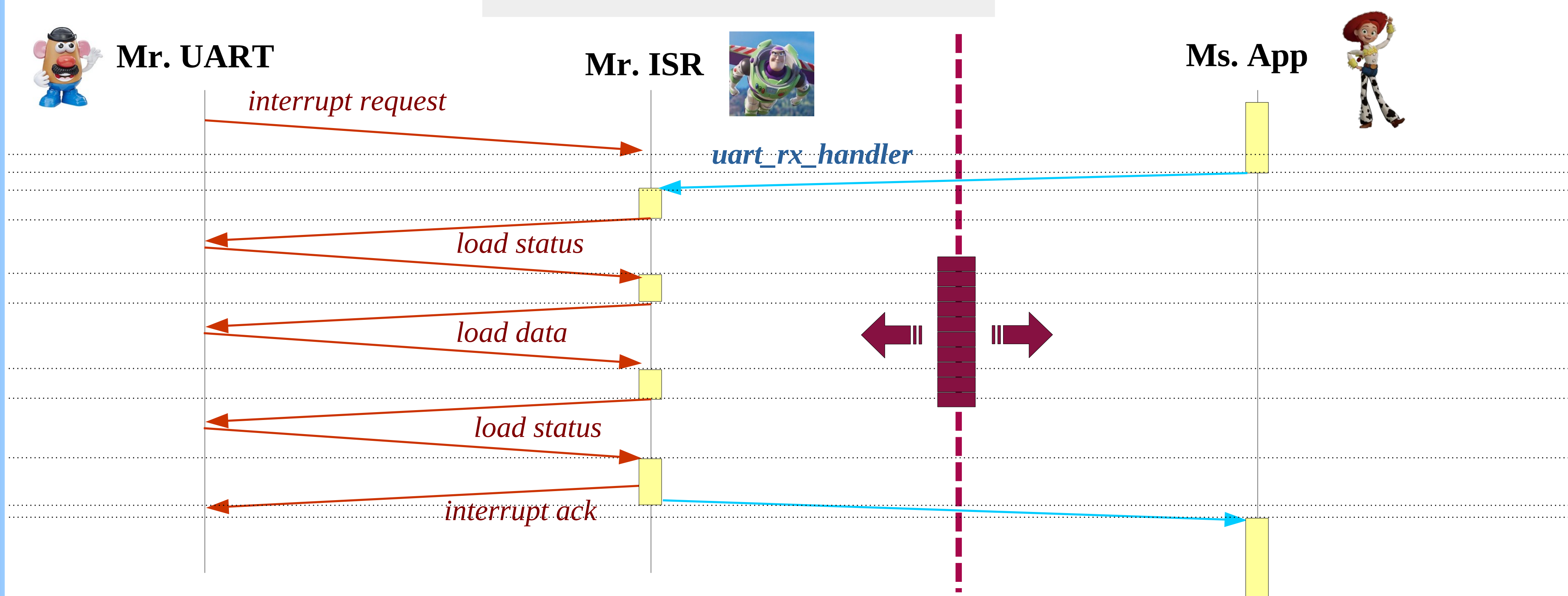
Ms. App
(application)

RACE CONDITION WARNING – ONLY SHARE RINGS




```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}
```

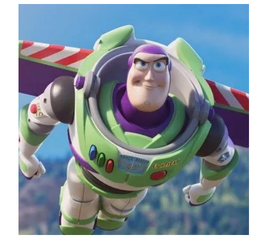
```
void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
    core_disable_interrupts();
    if (ring_empty())
      core_halt();
  }
}
```



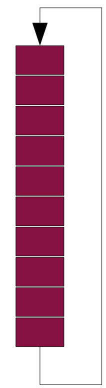
Using a Buffer – The Overall Idea


```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}

void _start() {
  uart_init(UART0, uart_rx_handler);
  core_enable_interrupts();
  for (;;) {
    process_ring();
    core_disable_interrupts();
    if (ring_empty())
      core_halt();
  }
}

char line[MAX_CHARS];
uint32_t nchars = 0;

void process_ring() {
  uint8_t code;
  while (!ring_empty()) {
    code = ring_get();
    line[nchars++]=(char)code;
    uart_send(UART0, code);
    if (code == '\n') {
      shell(line,nchars);
      nchars = 0;
    }
  }
}
```

***Append bytes** to the ring...*



Mr. ISR



Problems?

***Extract bytes** from the ring*
and does something with them...

Ms. App

Using a Buffer – The Overall Idea


```
void uart_rx_handler(void* cookie) {
  uint8_t code = uart_receive(UART0);
  while (code) {
    if (ring_full()) panic();
    ring_put(code);
    code = uart_receive(UART0);
  }
   uart_interrupt_ack();
}

void panic() {
  _reset();
}
```

Warning: <u>the circular buffer may be full</u>…

What should we do?

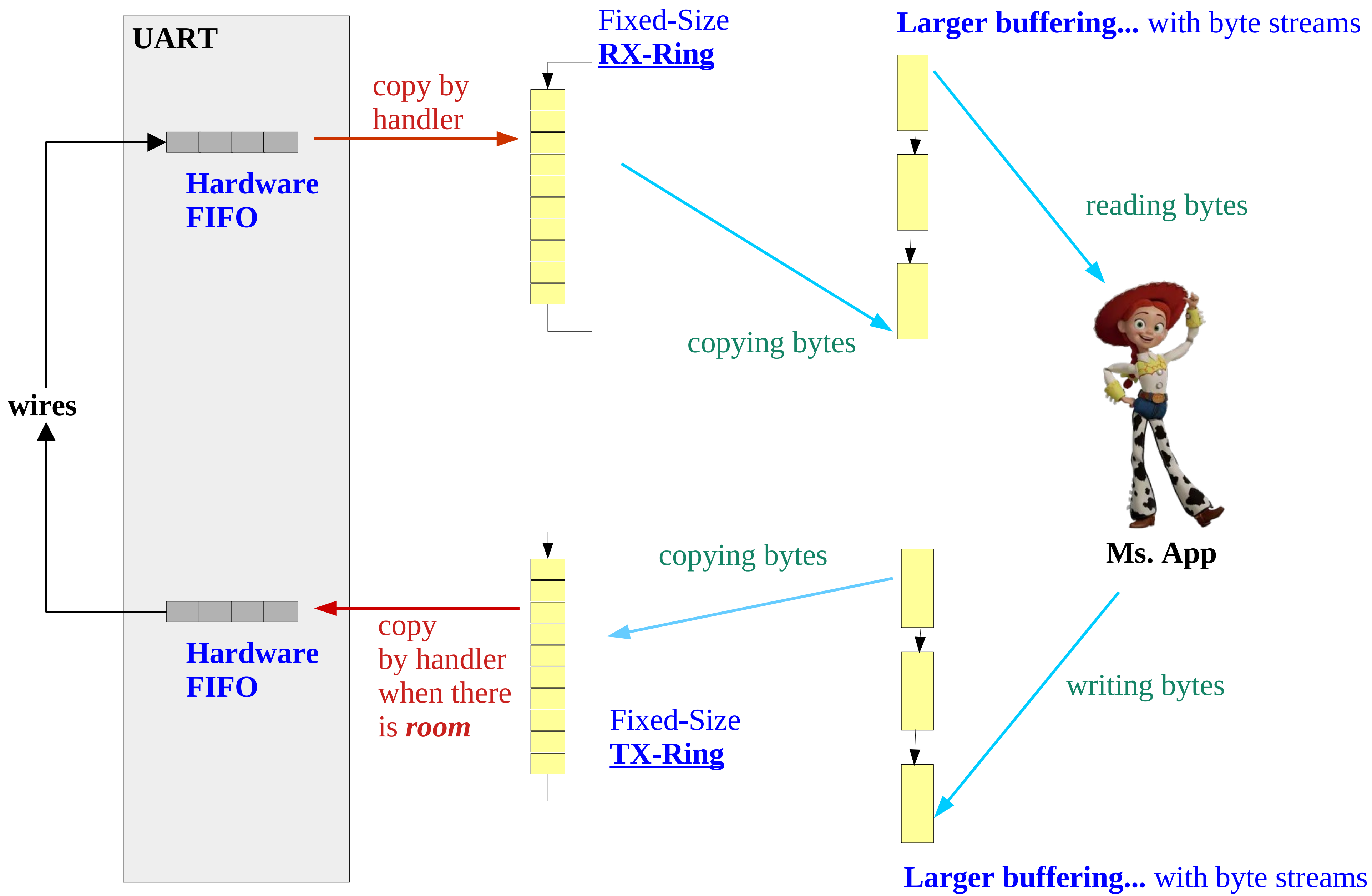
- ***Anything, but just no silent failures!***
- However, there is not much we can do... besides panicking... which usually means a ***reset*** of the board.

Remember, using a ring, we just bought ourselves more time… ;-)

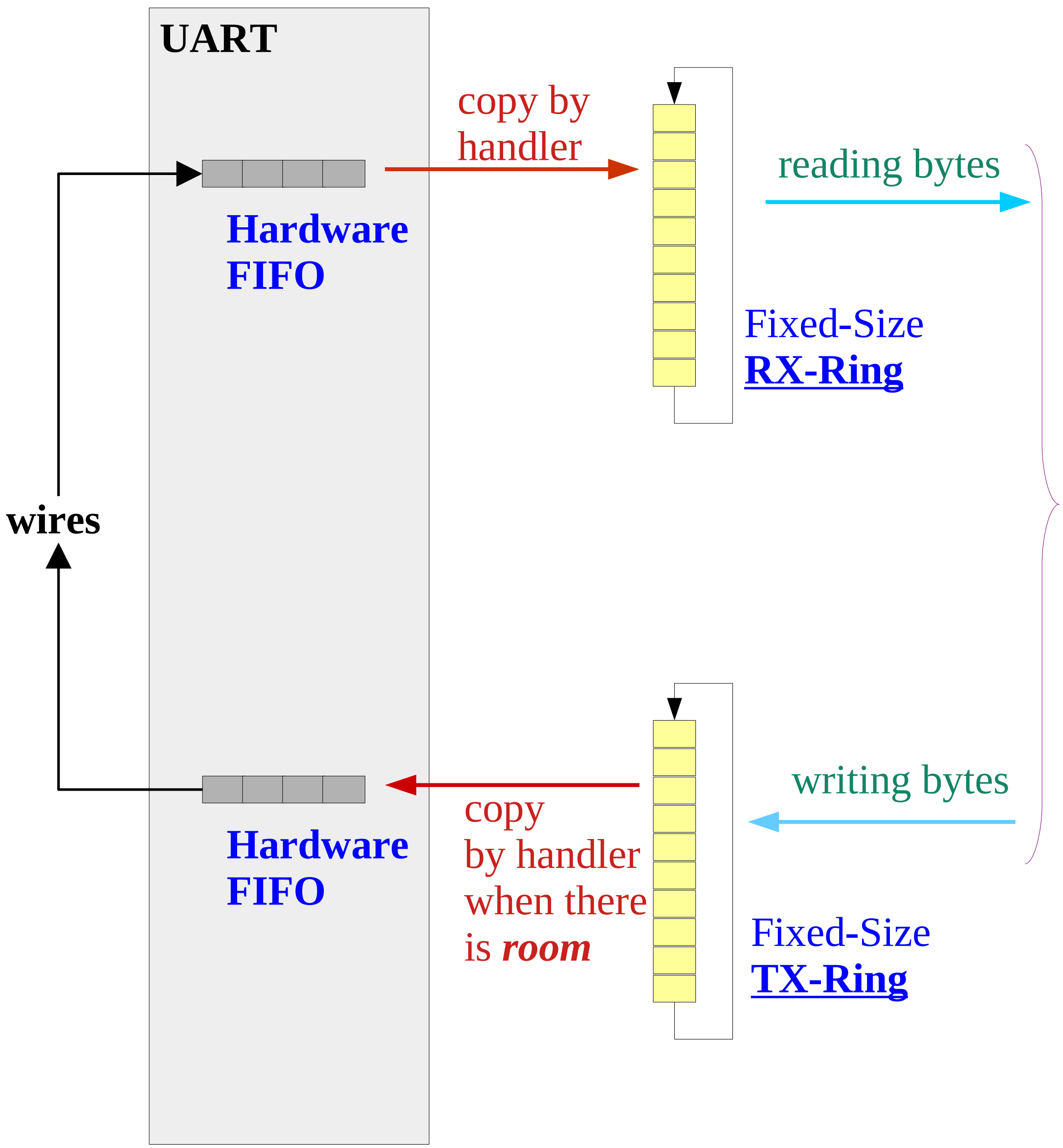
This means that the main loop must be able to keep up with incoming characters before the circular buffer fills up…

This suggests more buffering to buy more time... using <u>*dynamically allocated buffers*</u> and probably some form of <u>*control flow*</u>.

Fuller Buffering Picture – Typical of Larger Systems...



Fuller Buffering Picture – Keep it small, but...



Ms. App

She would love a nicer framework than rings… The idea of streams was nice… even if we cannot afford large buffers (small embedded systems).

something like a ***listener-based and non-blocking framework***, something like this:

```
void uart_init(uint8_t no,
               void (*rl)(void *cookie),
               void (*wl)(void *cookie),
               void *cookie);

bool_t uart_read(uint8_t no, uint8_t* bits);

bool_t uart_write(uint8_t no, uint8_t* bits);
```

Hum... how is this working?


```
void uart_init(uint8_t no,
               void (*read_listener)(void *cookie),
               void (*write_listener)(void *cookie),
               void *cookie);

bool_t uart_read(uint8_t no, uint8_t* bits);

bool_t uart_write(uint8_t no, uint8_t* bits);
```

The read listener will be called if there are bytes available to read.

The write listener will be called if there is room to write bytes, but only after a write failed because there was no room at the time.

Ms. App

How is this working actually?

Ah, right!
This is like event-oriented coding...

Init the application...

Ms. App, how do I initialize my app?

In this context,
it is about setting listeners.
That is it.

```
void read_listener(void *cookie);
void write_listener(void *cookie);

struct cookie {
  uint32_t uartno;
  char line[MAX_CHARS];
  uint32_t head = 0,tail = 0;
  bool_t processing=false;
};
struct cookie cookie;

void app_start() {
  uart_init(UART0, read_listener, write_listener,
            &cookie);
}
```

Coding listeners...


```
// Called whenever there are available bytes
void read_listener(void *addr) {
  struct cookie *cookie = addr;
  uint8_t code;
  while (!cookie->processing &&
         uart_receive(cookie->uartno,&code)) {
    cookie->line[cookie->head++]=(char)code;
    cookie->processing = (code == '\n');
    write_amap(cookie);
  }
  bool_t dropped=false;
  while (cookie->processing &&
         uart_receive(cookie->uartno,&code))
    dropped=true;
  if (dropped)
    beep(); // signal dropped bytes...
}
```

Note: *dropping bytes should be encoding aware,* ***dropping characters, not bytes***

```
struct cookie {
  uint32_t uartno;
  char line[MAX_CHARS];
  uint32_t head = 0,tail = 0;
  bool_t processing=false;
};
```

```
// Called when there are available bytes
void write_listener(void *addr) {
  struct cookie *cookie = addr;
  write_amap(cookie);
}
```

```
void write_amap(struct cookie *cookie) {
  while (cookie->tail < cookie->head) {
    uint8_t code = cookie->line[cookie->tail];
    if (!uart_write(cookie->uartno,code))
      return;
    cookie->tail++;
    if (code == '\n') {
      shell(cookie->line,cookie->head);
      cookie->tail= cookie->head = 0;
      cookie->processing=false;
    }
  }
}
```

amap: as much as possible

Nota Bene: the ownership transfer for the line buffer while processing. This essential here for correctness.

Idea: using a ring could help buffering while processing, but the ring might become full...

Interal Design – How?

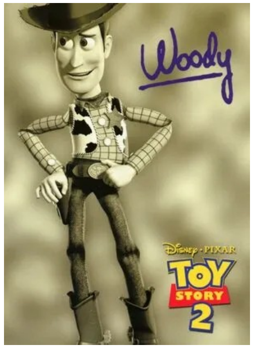

```c
void _start() {
  app_start();
  for (;;) {
    ???
    core_halt();
  }
}
```

How should we do that?

Mr. UART
(hardware)

Oh well, to infinity then!

Mr. ISR
(interrupt service routine)



Let's do it...

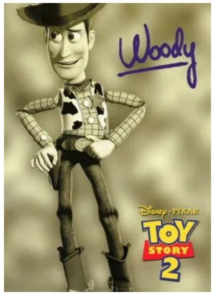
Mr. Event

Interal Design – A limited solution


```
struct uart {
  struct ring rx,tx;
  void (*rl)(uint8_t no);
  void (*wl)(uint8_t no));
  void *cookie;
};

struct uart uarts[NUARTS];
```

```
void _start() {
  uart_init(UART0,uart_rx_handler);
  for (;;) {
    core_disable_irqs();
    process_uart(UART0);
    core_halt();
  }
}

void process_uart(uint8_t no) {
  struct uart *uart = &uarts[no];
  process_uart_rx(uart);
  process_uart_tx(uart);
}

void process_tx_ring(struct *uart) { ... }

void process_rx_ring(struct *uart) {
  if (!ring_empty(&uart->rx)) {
    core_enable_interrupts();
    uart->rx(uart->cookie);
    core_disable_interrupts();
  }
}
```

```
void uart_init(uint8_t no,
               void (*read_listener)(void *cookie),
               void (*write_listener)(void *cookie),
               void *cookie);

bool_t uart_read(uint8_t no, uint8_t* bits);

bool_t uart_write(uint8_t no, uint8_t* bits);
```

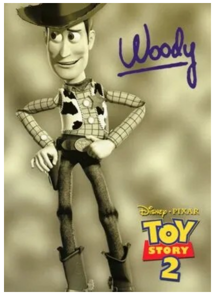


Not bad, it is a start...

But somewhat limited...

Towards a complete solution...


```
void _start() {
  uart_init(UART0,uart_rx_handler);
  core_enable_irqs();
  for (;;) {
    ...
    core_halt();
  }
}
```



As we have more and more things to do,
- more devices to attend to...
- also just more "***application tasks***" to carry out...

Two possible paradigms:
- ***Threads***
- ***Events***

Bare-metal systems[(1)] are reactive systems, using events.

(1) Some large systems make the same choice, like ***node.js*** or ***graphical toolkits***.

Event-oriented Programming

- Everything starts with an initial event
  - Posted by the initializing application
- Event reactions
  - Outside of handling interrupts
  - Any execution happens as executing a reaction
  - Each reactions run to completion
    - Reactions must be "***short***"
    - ***This means no blocking, no waiting, no spinning***
- Event pump
  - Two queues of events (ready, pending)
  - Reacting to events in the ready queue
  - Event reactions will post new events

```
// Post an new event on the FIFO queue
void event_post(void (*react)(void* cookie),
                void* cookie,
                uint32_t delay);
```

Event-oriented Programming – Partial Design


```
struct event {
  void* cookie;
  void (*react)(void* cookie);
  uint32_t eta; // Estimated Time of Arrival
}

struct queue {
  struct event events[];
  uint32_t head,tail;
}

struct queue ready;
struct queue pending;

void queue_init(); // pops the next event
struct event* queue_pop(); // pops the next event

int main(int argc, char** argv) {  // event pump,
  app_init();
  for (;;) {
     struct event* evt = event_pop();
     if (evt!=NULL)
       evt→react(evt→cookie);
      else
       core_halt();
  }
}
```

Ready queue is managed FIFO.

Pending queue is ordered by ETA.

Using a hardware timer and have the interrupt handler pop due events from the pending queue to the ready queue.

How would you model other interrupt handlers in this paradigm? User a handler and a ring, as before. The handler is called a "top" and needs a "bottom", the bottom being an event to consume from the ring.

Design a way for tops to request their bottom. Watch out for race condition and multipe bottoms...