Introducing micro:bian

Mike Spivey

Hilary Term 2022

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

In this part

- Concurrent processes and messages between them as a way of structuring complex systems that respond to events (L12).
- Managing I/O devices with driver processes that receive interrupts as messages (L13).
- Implementing multiple processes (L14).
- Messages and scheduling (L15).
- Chasing down a bug (L16).

Why concurrency?

- Genuinely parallel machines
- Sharing one machine between several tasks
- Decomposing one task clearly
- Responding to several sources of events

In this lecture

- *Processes*: embedded programs are conveniently structured as a set of independent processes.
- *Messages*: processes can cooperate by exchanging messages in a way that synchronises their behaviour.
- *Shared variables* are best avoided by using messages instead.

Hearts again

```
static int row = 0;

void advance(void) {
    row++;
    if (row == 3) row = 0;
    GPIO_OUT = heart[row];
}
```

- Efficient but inflexible.
- Can’t pause inside subroutines or control structures.

But also primes

Use interrupts to overlap printing with the search, but ...

- When the serial buffer is full, wastes time waiting in a loop.
- Disables interrupts to protect the buffer from concurrent modification – hard to get right.

We're ready for to use an operating system: enter micro:bian!

Better: a process

```
static void heart_task(int arg) {
    while (1) {
        show(heart, 70);
        show(small, 10);
        show(heart, 10);

    }
}
```

```
static void show(int img[], int n) {
    while (n-- > 0) {
        for (int p = 0; p < 3; p++) {
            GPIO_OUT = img[p];
            timer_delay(5);
        }
    }
}
```

Another, independent process

```
static void prime_task(int arg) {
     int p = 2, n = 0;

     while (1) {
          if (prime(p)) {
               n++;
               printf("prime(%d) = %d\n", n, p);
          }
          p++;
     }
}
```

```
serial_putc(c);
```

Setting the ball rolling

```
void init(void) {
    start(SERIAL, "Serial", serial_task, 0, STACK);
    start(TIMER, "Timer", timer_task, 0, STACK);
    start(HEART, "Heart", heart_task, 0, STACK);
    start(PRIME, "Prime", prime_task, 0, STACK);
}
```

- a fixed collection of processes created before concurrent execution begins.
- our two processes, plus *device drivers* for the timer (`timer_delay`) and serial port (`serial_putc`).

Processes

Each a 'main program' in its own right

- It can call subroutines.
- It can pause (or be interrupted) at any point to give others a go.

Implementation

- Processes are interleaved.
- Each has its own stack.

micro:bian supports a fixed set of processes.

Other operating systems

- Processes with communication
- Memory management
- Drivers for I/O devices
- File system
- Networking

micro:bian supports processes and messages, and whatever device drivers we write.

No utility programs, shared libraries, GUI, ... either.

Sending messages

```
void prime_task(int arg) {
    int n = 2;
    message m;

    while (1) {
        if (prime(n)) {
            m.int1 = n;
            send(USEPRIME, PRIME, &m);
        }
        n++;
    }
}
```

Receiving messages

```
void summary_task(int arg) {
    int count = 0, limit = arg; message m;

    while (1) {
        receive(PRIME, &m);
        while (m.int1 >= limit) {
            report(count, limit);
            limit += arg;
        }
        count++;
    }
}
```

Rules for messages

Both sender and receiver have a message buffer (16 bytes).

- The sender assembles a message; then
- It is transferred from sender to receiver as an *atomic* action.
- No buffering, no queues of messages!

Alternatives to messages

Message passing:

- no “shared variables” between processes.
- all communication by messages

Shared variables with semaphores:

- like the serial output buffer.
- more efficient, but hard to get right.

Device drivers

Mike Spivey
Hilary Term 2022

UNIVERSITY OF OXFORD
Department of COMPUTER SCIENCE

In this lecture

- *Interrupts* can be tamed by turning them into 'messages from the hardware'.
- *Device drivers* look after hardware devices by serving requests one at a time in a loop.

(See wiki and Lab 4 for all details – many are omitted here for clarity.)

Implementing serial output

```
void serial_putc(char ch) {
    message m;
    m.int1 = ch;
    send(SERIAL, PUTC, &m);
}
```

- request message sent to the `SERIAL` driver.
- the caller *waits* if the driver is not ready.

Implementing the driver process

```
void serial_task(int arg) {
    static char txbuf[NBUF];
    int bufin, bufout, bufcount;
    message m; char ch;

    serial_setup();

    while (1) {
        receive(ANY, &m);
        switch (m.m_type) {
            ...
        }
    }
}
```

State is local to the driver

A server loop accepts requests

Handling PUTC messages

```
while (1) {
    receive(ANY, &m);
    switch (m.m_type) {
    case PUTC:
        ch = m.int1;
        txbuf[bufin] = ch; ...
        break;

        ...
    }
}
```

- Buffer variables are local, so no other process can interfere.

Handling interrupts

Key insight:

an interrupt is a message from the hardware.

```
while (1) {
    receive(ANY, &m);
    switch (m.m_type) {
    case INTERRUPT:
        if (UART_TXDRDY) {
            txidle = 1;
            UART_TXDRDY = 0;
        }
        break;

        ...
```

Responding to events

```
while (1) {
    receive(ANY, &m);
    switch (m.m_type) {
            ...
    }

    if (txidle && bufcount > 0) {
        UART.TXD = txbuf[bufout]; ...
        txidle = 0;
    }
}
```

When the buffer is full

Let's replace

```
receive(ANY, &m);
```

with

```
if (bufcount < NBUF)
    receive(ANY, &m);
else
    receive(INTERRUPT, &m);
```

When the buffer is full, we just stop accepting requests until it has emptied a bit.