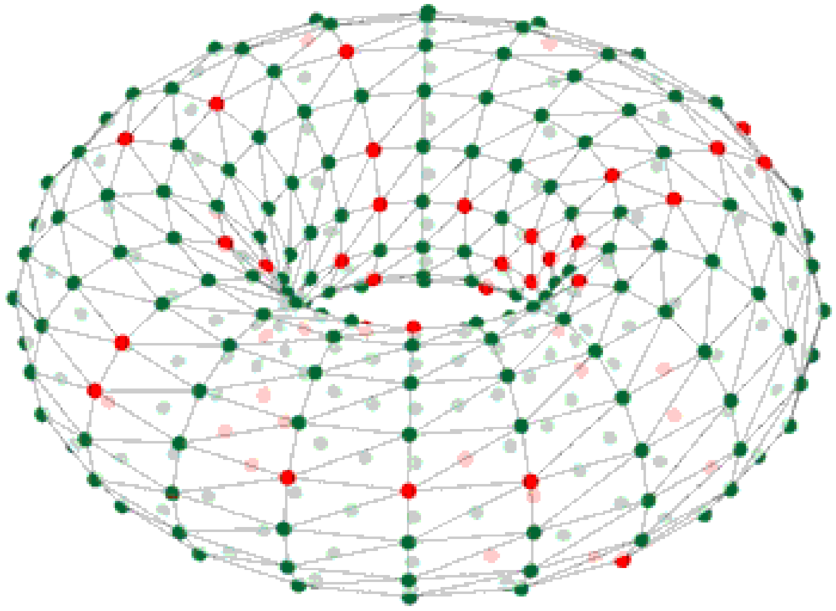
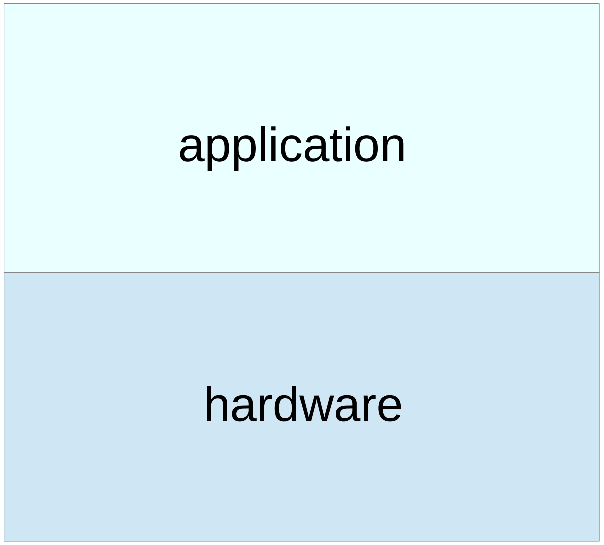
SpiNNaker API

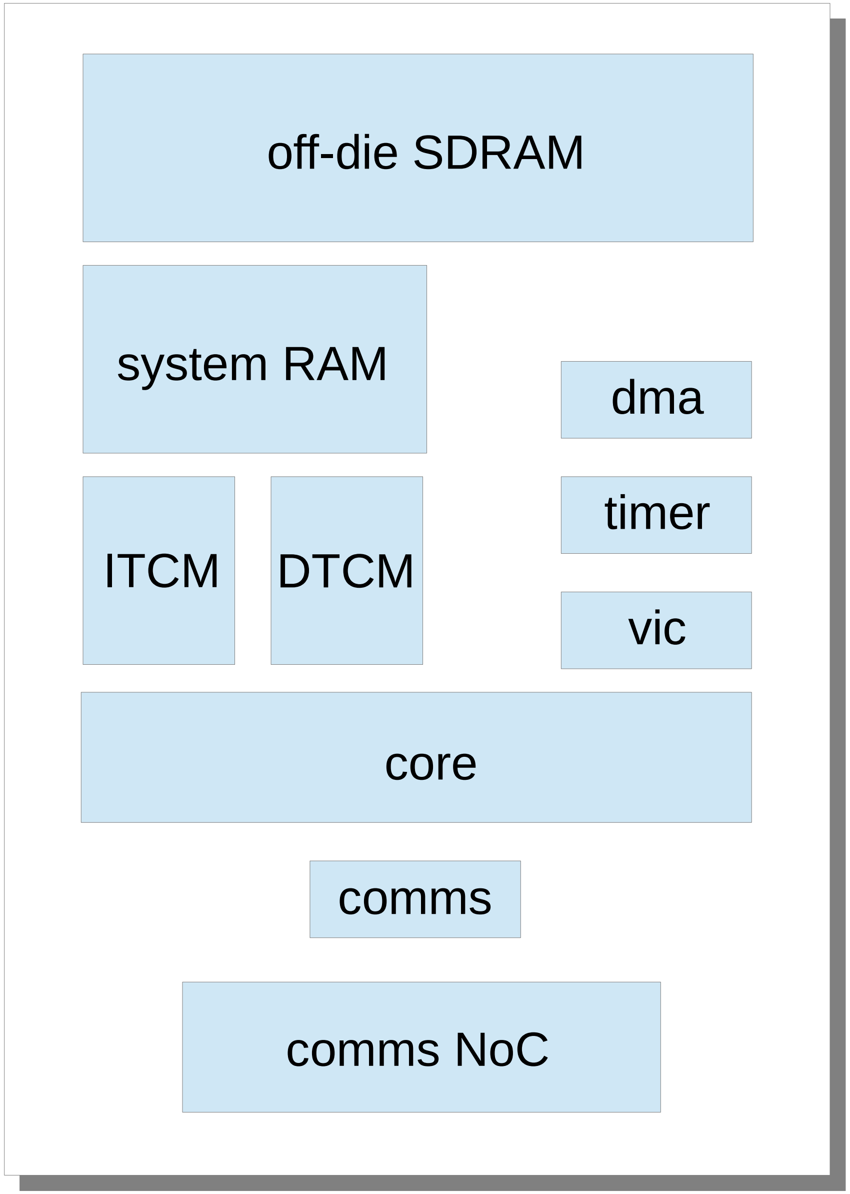


Luis Plana

SpiNNaker Workshop, September 2015

hardware resources





SARK: low-level software

application

SARK

hardware

application control

core control

memory management

peripheral management

event management

SDP messaging

API: run-time environment

application

API

SARK

hardware

event-driven
programming model

run-time
environment

SARK functionality
still available

event-driven model

applications do not control execution flow

applications indicate functions to be executed when events of interest occur

API controls execution and schedules application functions when appropriate

application functions are known as callbacks

events and callbacks

event	trigger
timer tick	periodic event has occurred
multicast packet received	multicast packet has arrived
DMA transfer done	scheduled DMA transfer completed successfully
SDP packet received	SDP packet has arrived
user event	application-triggered event has occurred

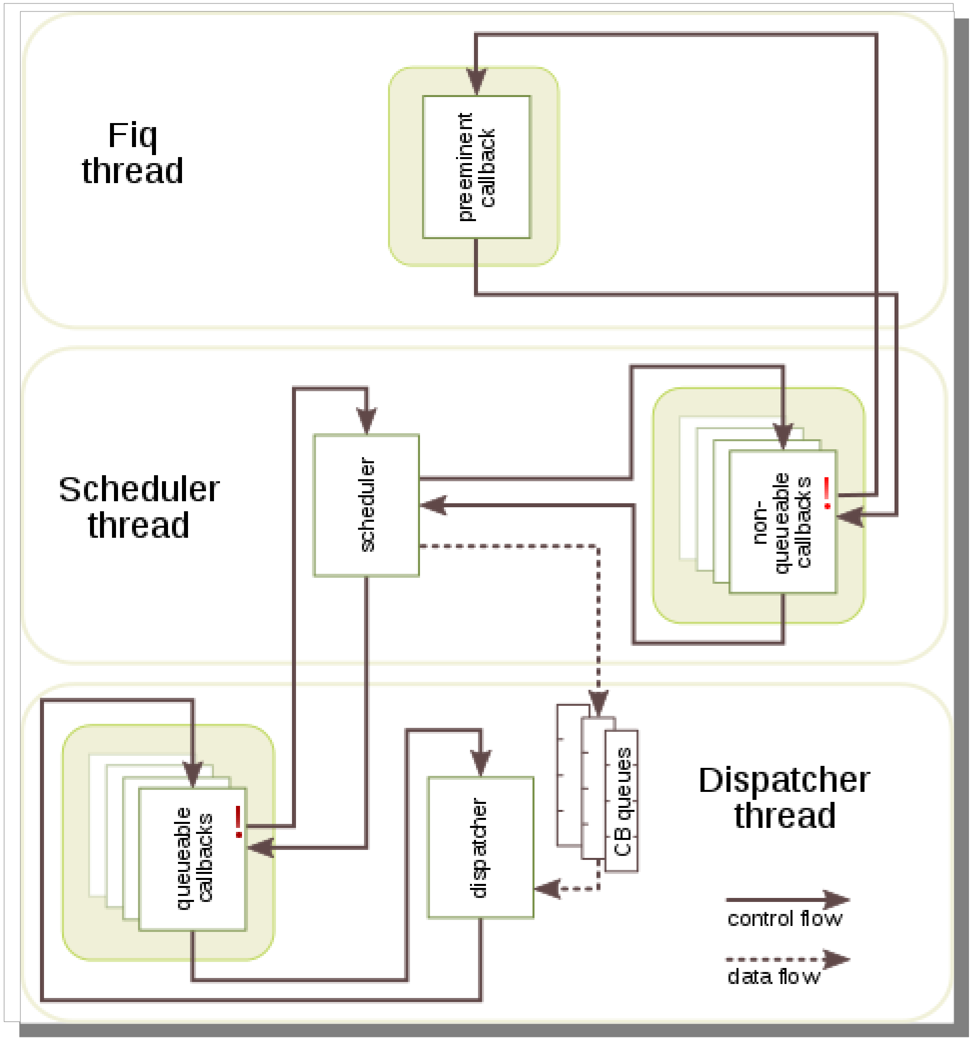
event	first argument	second argument
timer tick	simulation time (ticks)	null
MCP w/o payload received	key	0
MCP with payload received	key	payload
DMA transfer done	transfer ID	tag
SDP packet received	*mailbox	destination port
user event	arg0	arg1

priorities

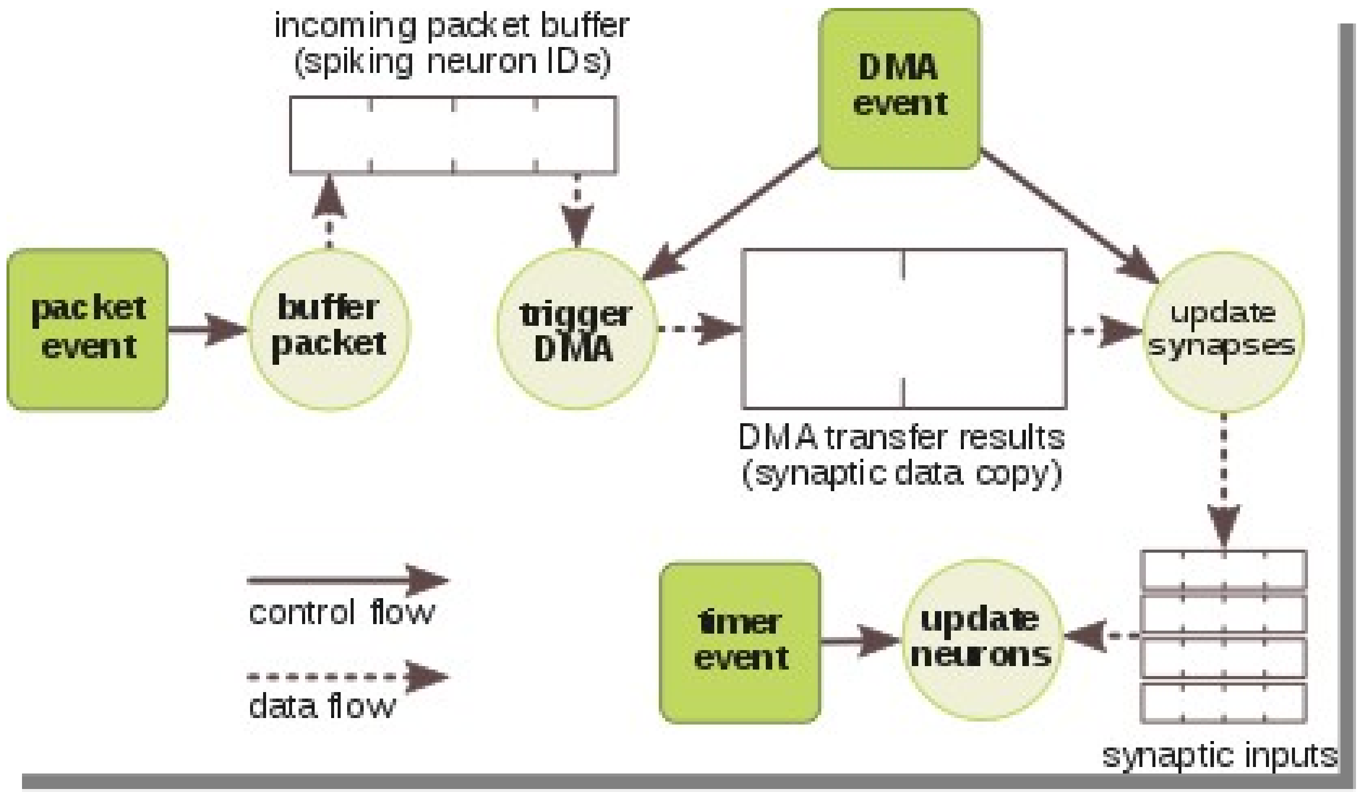
priority level = -1
only one callback
cannot be pre-empted

priority level = 0
can only be pre-empted
by priority -1 callback

priority level > 0
can be pre-empted
by priority <= 0 callbacks
scheduled in priority order



example: spiking neural network



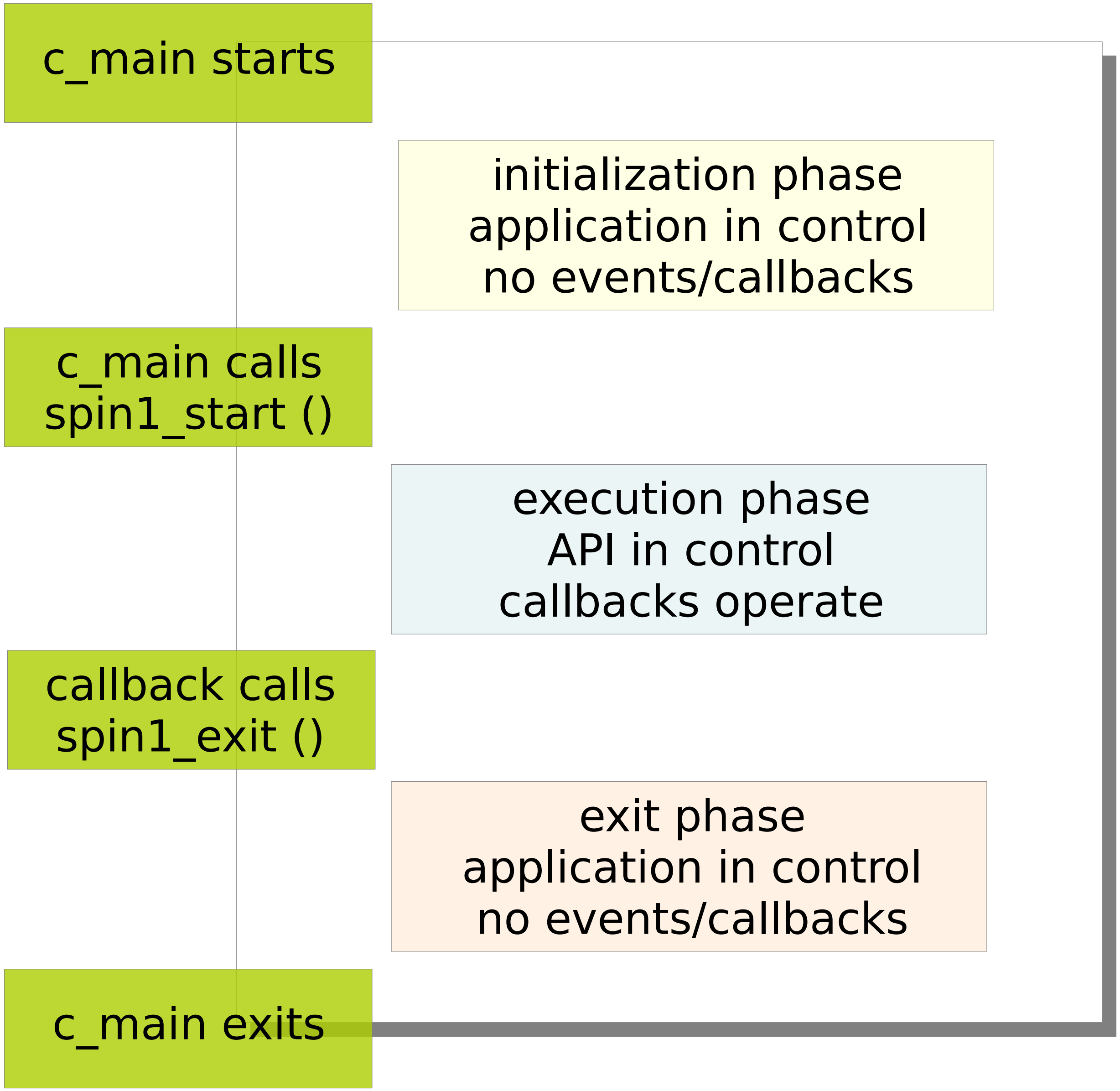
what is a sensible choice of priorities?

additional support

function	use
start/stop execution	start and stop simulation
set timer period	real-time or periodic callback
send multicast packet	inter-core communications
send SDP packet	host or I/O peripheral communications
start DMA transfer	software-managed cache
trigger user event	start a callback with priority <= 0
schedule callback	start a callback with priority > 0
enable/disable interrupts	critical section access (inter-thread control)
provide chip address and core ID	find out who you are
configure multicast routing table	setup routing entries

see API documentation for complete list

program structure



synchronization barrier

c_main starts

application initializes variables and may set up routing table

c_main calls spin1_start ()

core goes to sleep and waits for *sync0* signal

host can check that cores are waiting!

host sends *sync0* signal

core receives signal and starts execution

example: first.c

c_main

```
void c_main()
{
  // initialize variables and state
  // ------------------------------------------
  my_core = spin1_get_core_id();    // this core's id
  my_key  = ROUTING_KEY (my_core);  // this core's multicast routing key

  // initialize state in tubogrid
  // ------------------------------------------
  spin1_delay_us (100 * my_core);   // skew accesses to tubogrid!
  io_printf (IO_STD, state_colour[my_state]);

  // initialise routing tables: only one core needs to do it!
  // ------------------------------------------
  if (leadAp)
  {
    routing_table_init ();
  }

  // prepare for execution
  // ------------------------------------------
  // set timer tick value
  spin1_set_timer_tick (TIMER_TICK_PERIOD);

  // register callbacks
  spin1_callback_on (TIMER_TICK, update, 0);
  spin1_callback_on (MC_PACKET_RECEIVED, receive_packet, -1);

  // go
  // ------------------------------------------
  spin1_start(SYNC_WAIT);
}
```

example: first.c

definitions

```
// ------------------------------------------------
// simulation parameters and constants
// ------------------------------------------------
// set the execution speed and length
#define TIMER_TICK_PERIOD  125000   // 0.125s tick period (in microseconds)
#define TIMEOUT            128      // run for this many ticks

// indicate which core follows each other in the chain
uint next_core[] =
{
  2, 3, 4, 8, 6, 7, 11, 12, 5, 1, 10, 16, 9, 13, 14, 15
};

// enumerate possible core states
enum state_e {white = 0, red, green, blue};
typedef enum state_e state;

// an easy way to update to the next state
state next_state[] = {red, green, blue, white};

// prepare the strings to represent each state in tubogrid
char* state_colour[] =
{
  "#white;#fill;\n",
  "#red;#circle;\n",
  "#green;#circle;\n",
  "#blue;#circle;\n"
};
```

example: first.c

initialization

```
void routing_table_init ()
{
  // request table entries
  uint e = rtr_alloc (16);

  // write routing table entries to the router
  if (e != 0)
  {
    for (uint i = 0; i < 16; i++)
    {
      rtr_mc_set (e + i,                              // entry
                  ROUTING_KEY (i + 1),                // key
                  ROUTE_MASK,                         // mask
                  ROUTE_TO_CORE (next_core[i])        // route
                 );
    }
  }
}
```

example: first.c

packet callback

```
void receive_packet (uint pkt_key, uint pkt_payload)
{
  // update my state
    my_state = next_state[my_state];
}
```

example: first.c

timer callback

```
void update (uint ticks, uint b)
{
  // somebody has to get the changes started
  if ((ticks == 1) && (my_core == 1))
  {
    my_state = next_state[my_state];
  }

  // update if not finishing
  if (ticks <= TIMEOUT)
  {
    // check if state changed
    if (my_state != old_state)
    {
      // update tubogrid,
      io_printf (IO_STD, state_colour[my_state]);

      // send a packet to next core in the chain,
      spin1_send_mc_packet(my_key, 0, NO_PAYLOAD);

      // and remember state
      old_state = my_state;
    }
  }
  else
  {
    // finish
    spin1_exit (0);
  }
}
```

to think about: pitfalls

asynchronous operation and communications

multicast packets can be dropped due to congestion

UDP-based I/O
not guaranteed!

no floating-point support
use fixed-point arithmetic

no globally-shared resources
use message passing