SpiNNaker Application Programming Interface (API)

Version 2.0.0

10 March 2016

Application programming interface (API)

Event-driven programming model

The SpiNNaker API programming model is a simple, event-driven model. Applications do not control execution flow, they can only indicate the functions, referred to as callbacks, to be executed when specific events occur, such as the arrival of a packet, the completion of a Direct Memory Access (DMA) transfer or the lapse of a periodic time interval. A dispatcher kernel controls the flow of execution and schedules/dispatches application callback functions when appropriate.

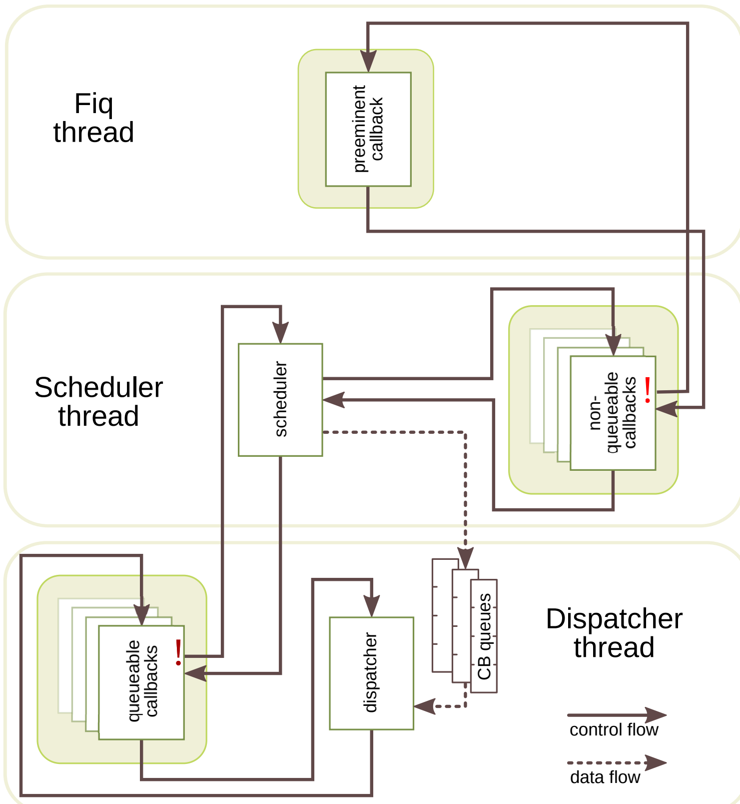

Figure 1: SpiNNaker event-driven programming framework.

Fig. **??** shows the basic architecture of the event-driven framework. Application developers write callback routines that are associated with events of interest and register them at a certain priority with the dispatcher. When the corresponding event occurs the dispatcher either executes the callback immediately and atomically (in the case of a non-queueable callback) or places it into a scheduling queue at a position according to its priority (in case of a queueable callback). When control is returned to the dispatcher (following the completion of a callback) the highest-priority queueable callback is executed. Queueable callbacks do not necessarily execute atomically: they may be pre-empted by non-queueable callbacks if a corresponding event occurs during their execution. The dispatcher goes to sleep (low-power consumption state) if the pending callback queues are empty and will be awakened by an event. Application developers can designate one non-queueable callback as the preeminent callback, which has the highest priority and can pre-empt other non-queueable callbacks as well as all queueable ones.

The preeminent callback is associated with a FIQ interrupt while other non-queueable callbacks are associated with IRQ interrupts.

Design considerations

- Non-queueable callbacks are available as a method of pre-empting long running tasks with short, high priority tasks. The allocation of application tasks to non-queueable callbacks must be carefully considered. The selection of the preeminent callback can be particularly important. Long-running operations should not be executed in non-queueable callbacks for fear of starving queueable callbacks.

- Queueable callbacks may require critical sections (*i.e.*, sections that are completed atomically) to prevent pre-emption during access to shared resources. Critical sections may be achieved by disabling interrupts before accessing the shared resource and re-enabling them afterwards. Applications are executed in a privileged mode to allow the callback programmer to insert these critical sections. This approach has the risk that it allows the programmer to modify peripherals, such as the system controller, unchecked.

- Non-queueable callbacks may also require critical sections, as they can be pre-empted by the preeminent callback.

- Events, usually triggered by interrupts, have priority determined by the programming of the Vectored Interrupt Controller (VIC). This allows priority to be determined when multiple events corresponding to different non-queueable callbacks occur concurrently. It also affects the order in which queueable callbacks of the same priority are queued.

Programming interface

The following sections introduce the events and functions supported by the API.

Events

The SpiNNaker API programming model is event-driven: all computation follows from some event. The following events are available to the application:

event	**trigger**
MC packet received	reception of a multicast packet (no payload)
MCPL packet received	reception of a multicast packet (with payload)
FR packet received	reception of a fixed route packet (no payload)
FRPL packet received	reception of a fixed route packet (with payload)
DMA transfer done	successful completion of a DMA transfer
Timer tick	passage of specified period of time
SDP packet received	reception of a SpiNNaker Datagram Protocol packet
User event	software-triggered interrupt

In addition, errors can also generate events:

— events not yet supported —

event	**trigger**
MCP parity error	multicast packet received with wrong parity
MCP framing error	wrongly framed multicast packet received
DMA transfer error	unsuccessful completion of a DMA transfer
DMA transfer timeout	DMA transfer is taking too long

Each of these events is handled by a dispatcher routine which may schedule or execute an application callback, if one is registered by the application.

Callback arguments

Callbacks are functions with two unsigned integer arguments and no return value. The arguments may be cast into the appropriate types by the callback. The arguments provided to callbacks (where 'none' denotes a superfluous argument) by each event are:

event	**first argument**	**second argument**
MC packet received	uint key	(uint none)
MCPL packet received	uint key	uint payload
FR packet received	uint 'key'	(uint none)
FRPL packet received	uint 'key'	uint payload
DMA transfer done	uint transfer_ID	uint tag
Timer tick	uint simulation_time	(uint none)
SDP packet received	uint mailbox	uint destination_port
User event	uint arg0	uint arg1

Pre-defined constants

logic value	**value**	**keyword**
true	(0 == 0)	TRUE
false	(0 != 0)	FALSE

function result	**value**	**keyword**
failure	0	FAILURE
success	1	SUCCESS

transfer direction	**value**	**keyword**
read (system to TCM)	0	DMA_READ
write (TCM to system)	1	DMA_WRITE

packet payload	**value**	**keyword**
no payload	0	NO_PAYLOAD
payload present	1	WITH_PAYLOAD

event	**value**	**keyword**
MC packet received	0	MC_PACKET_RECEIVED
DMA transfer done	1	DMA_TRANSFER_DONE
Timer tick	2	TIMER_TICK
SDP packet received	3	SDP_PACKET_RX
User event	4	USER_EVENT
MCPL packet received	5	MCPL_PACKET_RECEIVED
FR packet received	6	FR_PACKET_RECEIVED
FRPL packet received	7	FRPL_PACKET_RECEIVED

Pre-defined types

type	value	size
uint	unsigned int	32 bits
ushort	unsigned short	16 bits
uchar	unsigned char	8 bits
callback_t	void (*callback_t) (uint, uint)	32 bits
sdp_msg_t	struct (see below)	292 bytes
diagnostics_t	struct (see below)	44 bytes

SDP message structure

```
typedef struct sdp_msg              // SDP message (=292 bytes)
{
 struct sdp_msg *next;              // Next in free list
 ushort length;                     // length
 ushort checksum;                   // checksum (if used)

 // sdp_hdr_t

 uchar flags;                       // SDP flag byte
 uchar tag;                         // SDP IPtag
 uchar dest_port;                   // SDP destination port
 uchar srce_port;                   // SDP source port
 ushort dest_addr;                  // SDP destination addr
 ushort srce_addr;                  // SDP source address

 // cmd_hdr_t (optional)

 ushort cmd_rc;                     // Command/Return Code
 ushort seq;                        // Sequence number
 uint arg1;                         // Arg 1
 uint arg2;                         // Arg 2
 uint arg3;                         // Arg 3

 // user data (optional)

 uchar data[SDP_BUF_SIZE];          // User data (256 bytes)

 uint _PAD;                         // Private padding
} sdp_msg_t;
```

diagnostics variable structure

```
typedef struct
{
 uint exit_code;                // simulation exit code
 uint warnings;                 // warnings type bit map
 uint total_mc_packets;         // total routed MC packets during simulation
 uint dumped_mc_packets;        // total dumped MC packets by the router
 uint discarded_mc_packets;     // total discarded MC packets by API
 uint dma_transfers;            // total DMA transfers requested
 uint dma_bursts;               // total DMA bursts completed
 uint dma_queue_full;           // dma queue full count
 uint task_queue_full;          // task queue full count
 uint tx_packet_queue_full;     // transmitter packet queue full count
 uint writeBack_errors;         // write-back buffer errror count
} diagnostics_t;
```

Pre-declared variables

variable	type	function
leadAp	**uchar**	TRUE if appointed chip-wide application leader
diagnostics	**diagnostics_t**	returns diagnostic information (if turned on in compilation)

Dispatcher services

The dispatcher provides a number of services to the application programmer:

Simulation control functions

Start simulation		
function	arguments	description
uint spin1_start	sync_bool	synchronisation flag
returns:	EXIT_CODE (0 = NO ERRORS)	
notes:	• transfers control from the application to the dispatcher. • use spin1_exit to return with EXIT_CODE. • the argument should be `SYNC_NOWAIT` or `SYNC_WAIT`	

Stop simulation and report error		
function	arguments	description
void spin1_exit	uint rc	return code to report
returns:	no return value	
notes:	• transfers control from the dispatcher back to the application. • The argument is used as the return value for spin1_start.	

Set the timer tick period		
function	arguments	description
void spin1_set_timer_tick	uint period	timer tick period (in microseconds)
returns:	no return value	

Request simulation time		
function	arguments	description
uint spin1_get_simulation_time	void	no arguments
returns:	timer ticks since the start of simulation.	

Event management functions

Register **callback** to be executed when **event_id** occurs		
function	arguments	description
void spin1_callback_on	uint event_id	event that triggers callback
	callback_t callback	callback function pointer
	uint priority	priority <0 denotes preeminent
		priority 0 denotes non-queueable
		priorities >0 denote queueable
returns:	no return value	
notes:	• a callback registration overrides any previous ones for the same event. • only one callback can be registered as preeminent. • a second preeminent registration is demoted to non-queueable.	

Deregister **callback** from **event_id**		
function	arguments	description
void spin1_callback_off	uint event_id	event that triggers callback
returns:	no return value	

Schedule a **callback** for execution with given **priority**		
function	arguments	description
uint spin1_schedule_callback	callback_t callback	callback function pointer
	uint arg0	callback argument
	uint arg1	callback argument
	uint priority	callback priority
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• this function allows the application to schedule a callback without an event. • priority <= 0 must not be used (unpredictable results). • function arguments are not validated.	

Trigger a **user event**		
function	arguments	description
uint spin1_trigger_user_event	uint arg0	callback argument
	uint arg1	callback argument
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• FAILURE indicates a trigger attempt before a previous one has been serviced. • arg0 and arg1 will be passed as arguments to the registered callback. • function arguments are not validated.	

Data transfer functions

Request a DMA transfer		
function	arguments	description
uint spin1_dma_transfer	uint tag	for application use
	void *system_address	address in system NoC
	void *tcm_address	address in TCM
	uint direction	DMA_READ / DMA_WRITE
	uint length	transfer length (in bytes)
returns:	unique transfer identification number (TID)	
notes:	• completion of the transfer generates a DMA transfer done event.	
	• a registered callback can use TID and tag to identify the completed request.	
	• DMA transfers are completed in the order in which they are requested.	
	• TID = FAILURE (= 0) indicates failure to schedule the transfer.	
	• function arguments are not validated.	
	• may cause DMA error or DMA timeout events.	

Copy a block of memory		
function	arguments	description
void spin1_memcpy	void *dst	destination address
	void const *src	source address
	uint len	transfer length (in bytes)
returns:	no return value	
notes:	• function arguments are not validated.	
	• may cause a data abort.	

Communications functions

Send a multicast packet		
function	arguments	description
uint spin1_send_mc_packet	uint key	packet key
	uint data	packet payload
	uint load	1 = payload present / 0 = no payload
returns:	SUCCESS (=1) / FAILURE (=0)	

Send a fixed route packet		
function	arguments	description
uint spin1_send_fr_packet	uint key	packet 'key'
	uint data	packet payload
	uint load	1 = payload present / 0 = no payload
returns:	SUCCESS (=1) / FAILURE (=0)	

Flush software outgoing multicast packet queue		
function	arguments	description
uint spin1_flush_tx_packet_queue	void	no arguments
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• queued packets are thrown away (not sent).	

Flush software incoming multicast packet queue		
function	arguments	description
uint spin1_flush_rx_packet_queue	void	no arguments
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• queued packets are thrown away.	

SpiNNaker Datagram Protocol (SDP)

Send an SDP message		
function	arguments	description
uint spin1_send_sdp_msg	sdp_msg_t * msg	pointer to message
	uint timeout	transmission timeout (ms)
returns:	SUCCESS (=1) / FAILURE (=0)	

Request a free SDP message container		
function	arguments	description
sdp_msg_t * spin1_msg_get	void	no arguments
returns:	pointer to message (NULL if unsuccessful)	

Free an SDP message container		
function	arguments	description
void spin1_msg_free	sdp_msg_t *msg	pointer to message
returns:	no return value	

Critical section support functions

Disable IRQ interrupts		
function	arguments	description
uint spin1_irq_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Disable FIQ interrupts		
function	arguments	description
uint spin1_fiq_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Disable ALL interrupts		
function	arguments	description
uint spin1_int_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Restore core mode and interrupt state		
function	arguments	description
void spin1_mode_restore	uint status	CPSR state to be restored
returns:	no return value.	

System resources access functions

Get core ID		
function	arguments	description
uint spin1_get_core_id	void	no arguments
returns:	core ID in bits [4:0].	

Get chip ID		
function	arguments	description
uint spin1_get_chip_id	void	no arguments
returns:	chip ID in bits [15:0].	
notes:	• chip ID contains x coordinate in bits [15:8], y coordinate in bits [7:0].	

Get ID		
function	arguments	description
uint spin1_get_id	void	no arguments
returns:	chip ID in bits [20:5] / core ID in bits [4:0].	

Control state of board LEDs		
function	arguments	description
void spin1_led_control	uint p	new state for board LEDs
returns:	no return value.	
notes:	• the number of LEDs and their colour varies according to board version. • to turn LEDs 0 and 1 on: spin1_led_control (LED_ON (0) + LED_ON (1)) • to invert LED 2: spin1_led_control (LED_INV (2)) • to turn LED 0 off: spin1_led_control (LED_OFF (0))	

Memory allocation

Allocate a new block of DTCM		
function	arguments	description
void * spin1_malloc	uint bytes	size of the memory block in bytes
returns:	pointer to the new memory block.	
notes:	• DEPRECATED - use sark_alloc, sark_free • memory blocks are word-aligned. • memory is allocated in DTCM. • there is no support for freeing a memory block.	

Miscellaneous

		Wait for a given time
function	arguments	description
void spin1_delay_us	uint time	wait time (in microseconds)
returns:	no return value	
notes:	• the function busy waits for the given time (in microseconds).	
	• prevents any queueable callbacks from executing (use with care).	

		Generate a 32-bit pseudo-random number
function	arguments	description
void spin1_rand	void	no arguments
returns:	32-bit pseudo-random number	
notes:	• Function based on example function in:	
	• "Programming Techniques", ARM document ARM DUI 0021A.	
	• Uses a 33-bit shift register with exclusive-or feedback taps at bits 33 and 20.	

		Provide a seed to the pseudo-random number generator
function	arguments	description
void spin1_srand	uint seed	32-bit seed
returns:	no return value	

Application Program Structure

In general, an application program contains three basic sections:

- **Application Functions**: General application functions to support the callbacks.
- **Application Callbacks**: Functions to be associated with run-time events.
- **Application Main Function**: Variable initialisation, callback registration and transfer of control to main loop.

The structure of a simple application program is shown below Many details are left out for brevity.

```
// declare application types and variables
neuron_state state[1000];
spike_bin bins[1000][16];
. . .

/* ------------------------------------------------------------------------ */
/* ---------------------- application functions ----------------------- */
/* ------------------------------------------------------------------------ */
void izhikevich_update(neuron_state *state){
    . . .
    spin1_send_mc_packet(key, 0, NO_PAYLOAD);
    . . .
}

syn_row_addr lookup_synapse_row(neuron_key key)
{
    . . .
}

void bin_spike(neuron_key key, axn_delay delay, syn_weigth weight)
{
    . . .
}

/* ------------------------------------------------------------------------ */
/* ---------------------- application callbacks ----------------------- */
/* ------------------------------------------------------------------------ */
void update_neurons()
{
    . . .
    if (spin1_get_simulation_time() > 1000) // simulation time in "ticks"
        spin1_exit(0);
    else
        for (i=0; i < 1000; i++) izhikevich_update(state[i]);
    . . .
}

void process_spike(uint key, uint payload)
{
    . . .
    row_addr = lookup_synapses(key);
    tid = spin1_dma_transfer(tag, row_addr, syn_buffer, READ, row_len);
    . . .
}

void schedule_spike()
{
    . . .
    bin_spike(key, delay, weight);
    . . .
}

/* ------------------------------------------------------------------------ */
/* ------------------------- application main ------------------------ */
/* ------------------------------------------------------------------------ */
void c_main()
{
    // initialise variables and timer tick
    . . .
    spin1_set_timer_tick(1000); // timer tick period in microseconds
    . . .
    // register callbacks
    spin1_callback_on(TIMER_TICK, update_neurons, 1);
    spin1_callback_on(MC_PACKET_RECEIVED, process_spike, 0);
    spin1_callback_on(DMA_TRANSFER_DONE, schedule_spike, 0);
    . . .
    // transfer control to the dispatcher
    spin1_start(SYNC_WAIT);
    // control returns here on execution of spin1_exit()
}
```

Changes in Version 1.3

The following changes were made in version 1.3 of the API.

- The function `spin1_set_mc_table_entry` was removed. The SARK functions `rtr_alloc` and `rtr_mc_set` should be used instead.
- The functions `spin1_stop` and `spin1_kill` have been removed and replaced by `spin1_exit` which provides a unified way to stop the API dispatcher and pass back a return code.
- The functions `spin1_set_core_map` and `spin1_application_core_map` have been removed. They were used to synchronise application start-up and this is now done by an argument passed to `spin1_start`.
- The function `spin1_start` now takes a single argument `SYNC_WAIT` or `SYNC_NOWAIT` which indicates if the application should synchronise with applications on other cores before entering the API dispatcher. This was previously indicated by the presence of a core map.
- There is a new event `MCPL_PACKET_RECEIVED` which allows (and requires) separate callbacks to be provided for received multicast packets with and without payloads.
- The use of `spin1_malloc` is deprecated. The SARK routines `sark_alloc` and `sark_free` provide access to a more flexible heap which allows blocks to be freed.

Changes in Version 2.0.0

The following changes were made in version 2.0.0 of the API.

- There are two new events `FR_PACKET_RECEIVED` and `FRPL_PACKET_RECEIVED` which allow reception of fixed route packets (without and with payload). There is a corresponding function `spin1_send_fr_packet` to transmit fixed route packets. The SARK functions `rtr_fr_set` and `rtr_fr_get` can be used to set up and query the fixed routes.