




SpiNNaker

Universal Spiking Neural Network Architecture



SpiNNaker Application Programming Interface
version 0.0
09 December 2011

About this Document

Background

SpiNNaker was designed at the University of Manchester within an EPSRC-funded project in collaboration with the University of Southampton, ARM Limited and Silistix Limited. Subsequent development took place within a second EPSRC-funded project which added the universities of Cambridge and Sheffield to the collaboration. The work would not have been possible without EPSRC funding, and the support of the EPSRC and the industrial partners is gratefully acknowledged.

Intellectual Property rights

All rights to the SpiNNaker design and its associated software are the property of the University of Manchester with the exception of those rights that accrue to the project partners in accordance with the contract terms.

Disclaimer

The details in this design document are presented in good faith but no liability can be accepted for errors or inaccuracies. The design of a complex chip multiprocessor and its associated software is a research activity where there are many uncertainties to be faced, and there is no guarantee that a SpiNNaker system will perform in accordance with the specifications presented here.

The APT group in the School of Computer Science at the University of Manchester was responsible for all of the architectural and logic design of the SpiNNaker chip, with the exception of synthesizable components supplied by ARM Limited and interconnect components supplied by Silistix Limited. All design verification was also carried out by the APT group. As such the industrial project partners bear no responsibility for the correct functioning of the device.

Error notification and feedback

Please email details of any errors, omissions, or suggestions for improvement to Steve Furber <steve.furber@manchester.ac.uk>

Change history

version	date	changes
0.0	20/05/2011	Initial draft

Contents

About this Document

Background	2
Intellectual Property rights	2
Disclaimer	2
Error notification and feedback	2
Change history	2

Application programming interface (API)

0.1 Event-driven programming model	4
0.1.1 Design considerations	5
0.2 Programming interface	5
0.2.1 Events	5
0.2.2 Callback arguments	6
0.2.3 Pre-defined types and Constants	6
0.2.4 Kernel services	7
0.2.5 Application Programme Structure	16

Application programming interface (API)

0.1 Event-driven programming model

The SpiNNaker Programming Model (PM) 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. An Application Run-time Kernel (ARK) controls the flow of execution and schedules/dispatches application callback functions when appropriate.

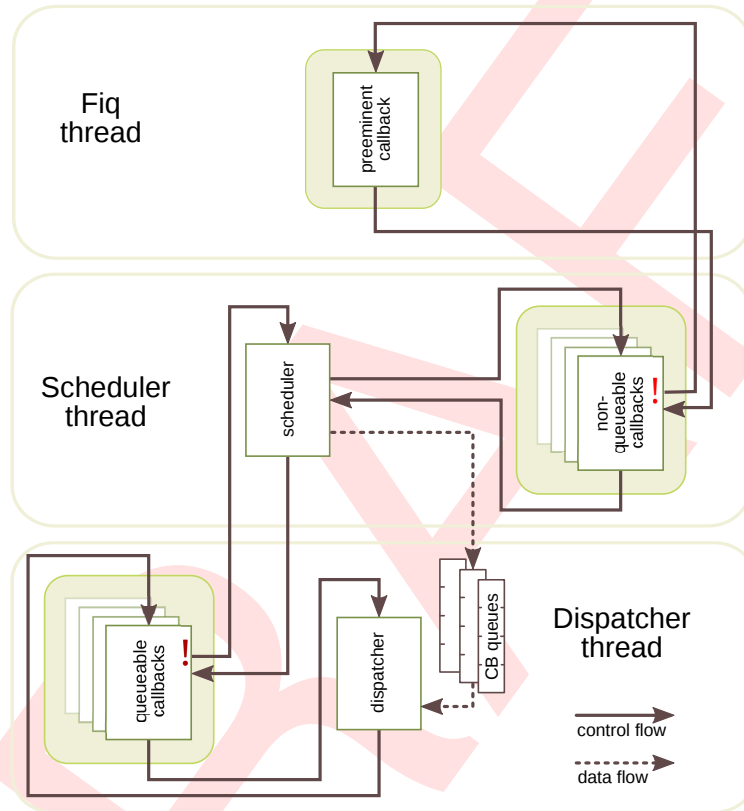


Figure 1: SpiNNaker event-driven programming framework.

Fig. 1 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 kernel. When the corresponding event occurs the scheduler 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 preminent callback, which has the highest priority and can pre-empt other non-queueable callbacks as well as all queueable ones.

The preminent callback is associated with a FIQ interrupt while other non-queueable callbacks are associated with IRQ interrupts. The API provides different functions to

disable interrupts: `spin1_irq_disable` disables IRQs, `spin1_fiq_disable` disables FIQs while `spin1_int_disable` disables both FIQs and IRQs. The use of `spin1_fiq_disable` may lead to priority inversion.

0.1.1 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.

0.2 Programming interface

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

0.2.1 Events

The SpiNNaker PM 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
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 kernel routine which may schedule or execute an application callback, if one is registered by the application.

0.2.2 Callback arguments

Callbacks are functions with two unsigned integer arguments (which may be NULL) 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 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

0.2.3 Pre-defined types and Constants

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	292 bytes

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

0.2.4 Kernel services

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

Simulation control functions

Start simulation		
function	arguments	description
uint spin1_start	void	no arguments
returns: EXIT_CODE (0 = NO ERRORS)		
notes: <ul style="list-style-type: none"> • transfers control from the application to the ARK. • use spin1_kill to indicate a non-zero EXIT_CODE. 		
Stop simulation		
function	arguments	description
void spin1_stop	void	no arguments
returns: no return value		
notes: <ul style="list-style-type: none"> • transfers control from the ARK back to the application. 		
Stop simulation and report error		
function	arguments	description
void spin1_kill	uint error	error code to report
returns: no return value		
notes: <ul style="list-style-type: none"> • transfers control from the ARK 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.		
Indicate which cores are involved in the simulation		
function	arguments	description
void spin1_set_core_map	uint chips	number of chips
	uint * core_map	bit map array of cores
returns: no return value		
notes: <ul style="list-style-type: none"> • sets the map of the cores that need to synchronise to start the simulation. • the numbers of chips & cores default to 1, thus no synchronisation is attempted. 		

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:		
<ul style="list-style-type: none"> • 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:		
<ul style="list-style-type: none"> • 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 void *system_address void *tcm_address uint direction uint length	for application use address in system NoC address in TCM DMA_READ / DMA_WRITE transfer length (in bytes)
returns: unique transfer identification number (TID)		
notes: <ul style="list-style-type: none"> • 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 void const *src uint len	destination address source address transfer length (in bytes)
returns: no return value		
notes: <ul style="list-style-type: none"> • function arguments are not validated. • may cause a data abort. 		

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)	

Flush software outgoing multicast packet queue		
function	arguments	description
uint spin1_flush_tx_packet_queue	void	no arguments
returns: SUCCESS (=1) / FAILURE (=0)		
notes: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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 uint timeout	pointer to message transmission timeout
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		

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 address
    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;
  
```

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))		

Set up a multicast routing table entry		
function	arguments	description
uint spin1_set_mc_table_entry	uint entry	table entry
	uint key	entry routing key field
	uint mask	entry mask field
	uint route	entry route field
returns: SUCCESS (=1) / FAILURE (=0).		
notes: • see SpiNNaker datasheet for details of the MC table operation. • entries 0 to 999 are available to the application. • routing keys with bit[15] = 1 and bit[10] = 0 are reserved. • function arguments are not validated.		

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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none">• the function busy waits for the given time (in microseconds).• prevents any queueable callbacks from executing (use with care).		

0.2.5 Application Programme Structure

In general, an application programme 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 programme is shown on the next page. 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, NOPAYLOAD);
    . . .
}

syn_row_addr lookup_synapse_row(neuron_key key)
{
    . . .
}

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

/* ----- */
/* ----- application callbacks ----- */
/* ----- */
void update_neurons()
{
    . . .
    if (spin1_get_simulation_time() > 1000) // simulation time in "ticks"
        spin1_stop();
    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(MCPACKET_RECEIVED, process_spike, 0);
    spin1_callback_on(DMA_TRANSFER_DONE, schedule_spike, 0);
    . . .
    // transfer control to the run-time kernel
    spin1_start();
    // control returns here on execution of spin1_stop()
}

```