SpiNNaker Application Programming Interface

version 0.0

08 June 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

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

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 DMA memory 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.

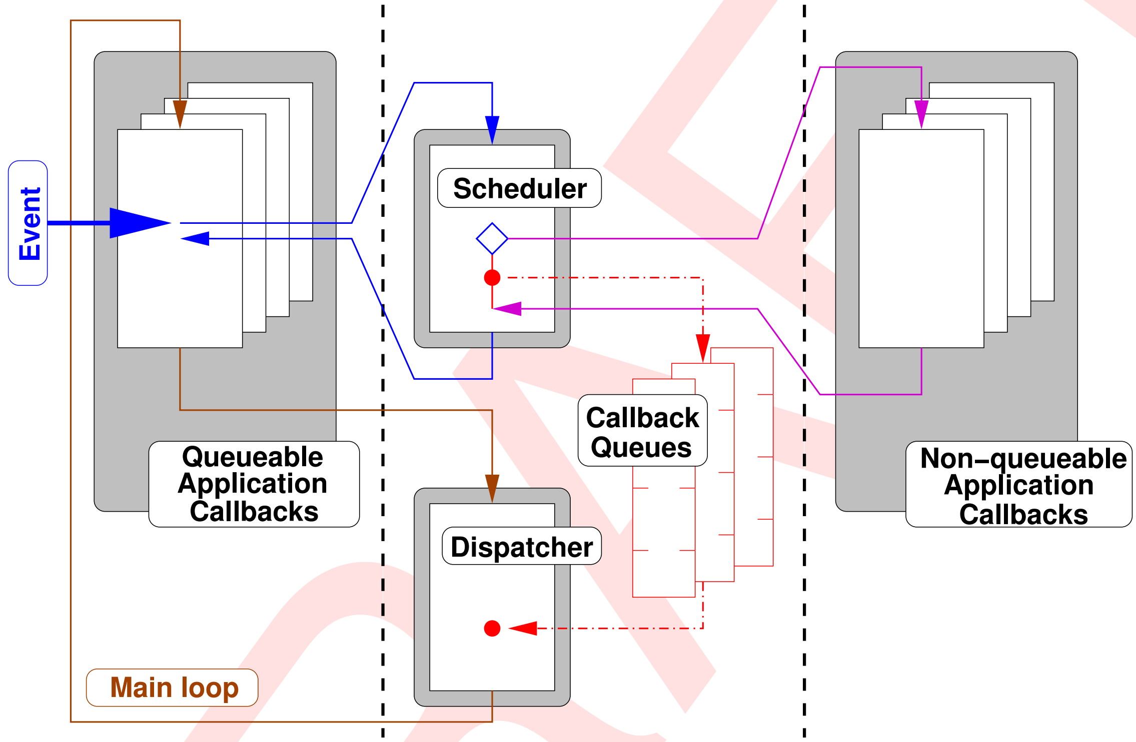

Figure 1: SpiNNaker event-driven programming framework.

Fig. 1 shows the basic architecture of the event-driven framework. The application space is shown on the left and right segments and the kernel space is in the center. 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.

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. 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 do not require explicit management of critical sections, as they are completed atomically by the event handler.
- 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
— events not yet supported —	
Host communication	raised by the host via the local monitor processor

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.

Events under consideration:	• application-triggered event. • MC packet without payload received event. • MC packet with payload received event.

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
Host communication	uint *mailbox	uint none

0.2.3 Pre-defined types and Constants

type	value	length
uint	unsigned int	32 bits
ushort	unsigned short	16 bits
uchar	unsigned char	8 bits
callback_t	void (*callback_t) (uint, uint)	32 bits

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
Host communication	3	HOST_COMM

0.2.4 Kernel services

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

Simulation control functions

		Start simulation
function	arguments	description
void start	void	no arguments
returns:	no return value	
notes:	• transfers control from the application to the ARK.	

		Stop simulation
function	arguments	description
void stop	void	no arguments
returns:	no return value	
notes:	• transfers control from the ARK back to the application.	

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

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

		Set number of cores in the simulation
function	arguments	description
void set_number_of_cores	uint ncores	number of cores in simulation
returns:	no return value	
notes:	• sets the number of cores that need to synchronise to start the simulation. • the number of cores defaults to 1, thus no synchronisation is attempted.	

		Wait for a given time
function	arguments	description
void 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).	

Event management functions

Register **callback** to be executed when **event_id** occurs		
function	arguments	description
void callback_on	uint event_id	event that triggers callback
	callback_t callback	callback function pointer
	uint priority	priority 0 denotes non-queueable
		priorities 1–4 denote queueable
returns:	no return value	
notes:	• a callback registration overrides any previous ones for the same event.	

Deregister **callback** from **event_id**		
function	arguments	description
void 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 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.	

Data transfer functions

		Request a DMA transfer
function	arguments	description
uint 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 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 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 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 flush_rx_packet_queue	void	no arguments
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• queued packets are thrown away.	

Critical section support functions

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

		Enable interrupts
function	arguments	description
uint irq_enable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

		Restore interrupt state
function	arguments	description
void irq_restore	uint status	CPSR state to be restored
returns:	no return value.	

System resources access functions

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

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

		Get current value of the LEDs 0 and 1
function	arguments	description
uchar get_leds	void	no arguments
returns:	value of LEDs in bottom 2 bits.	

		Set LEDs 0 and 1 to new value
function	arguments	description
void set_leds	uchar leds	new value for LEDs 0 and 1
returns:	no return value.	

		Set up a multicast routing table entry
function	arguments	description
uint 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. • function arguments are not validated.	

Host communication functions

— function not yet implemented —

Send data to host

function	arguments	description
void host_put	uint stream	stream handle
	void *data	data to transfer
	uint length	length of data (in bytes)
	returns:	no return value

notes:
- completion or failure of the put generates a host communication event.
- a registered callback can use resource_id to identify the completed request.
- data is sent to the host transparently via the monitor processor.

— function not yet implemented —

Request data from host

function	arguments	description
void host_get	uint stream	stream handle
	void *data	data to transfer
	uint length	length of data (in bytes)
	returns:	no return value

notes:
- completion or failure of the get generates a host communication event.
- a registered callback can use resource_id to identify the completed request.
- data is requested from the host transparently via the monitor processor.

— function not yet implemented —

Open a communication stream with the host

function	arguments	description
uint host_open	uint resource_id	host resource
	returns:	stream handle (= 0 represents FAILURE)

— function not yet implemented —

Close a communication stream with the host

function	arguments	description
uint host_close	uint stream	stream handle
	returns:	SUCCESS (=1) / FAILURE (=0)

Memory allocation

Allocate a new block of DTCM		
function	arguments	description
void * malloc	uint bytes	size of the memory block in bytes
returns:	pointer to the new memory block.	

notes:
- memory blocks are word-aligned.
- memory is allocated in DTCM.
- there is no support for freeing a memory block.

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){
    . . .
    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 (get_simulation_time() > 1000) // simulation time in "ticks"
        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 = 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
    . . .
    host_get(strm0, synapes, syn_len);
    set_timer_tick(1000); // timer tick period in microseconds
    . . .
    // register callbacks
    callback_on(TIMER_TICK, update_neurons, 1);
    callback_on(MC_PACKET_RECEIVED, process_spike, 0);
    callback_on(DMA_TRANSFER_DONE, schedule_spike, 0);
    . . .
    start();
    // control returns here on execution of stop()
    host_put(strm1, state, neuron_len);
}
```